SCALABLE AND DISTRIBUTED CONSTRAINED LOW RANK APPROXIMATIONS

A Thesis
Presented to
The Academic Faculty

by

Ramakrishnan Kannan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2016

SCALABLE AND DISTRIBUTED CONSTRAINED LOW RANK APPROXIMATIONS

Approved by:

Professor Haesun Park, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Duen Horng (Polo) Chau
School of Computation Science and Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computation Science and Engineering
Georgia Institute of Technology

Grey Malone Ballard
Truman Fellow
Sandia National Laboratory

Charu Aggarwal
Research Staff Member
IBM Research

Date Approved: 22 February 2016

Dedicated to the

lotus feet of my gurus

ACKNOWLEDGEMENTS

A wise adage says, "There is a woman behind every successful man". But my success is attributable to three wonderful ladies – my Wife, my Mother and Prof. Haesun Park. This thesis is a result of the motivation of my wife who patiently stood behind me on every challenging occasion over the past five years. Her ordeals during this period is worth a book and this thesis could not have been successful without her support. My mother, despite the need for financial and moral support for caring my ailing father, encouraged me to pursue my ambition several thousand miles away. My advisor Prof. Haesun Park, who spent significant amount of time with me explaining foundational scientific and numerical computing. Her encouragement to explore risky research directions, has broadened the scope of my research. Her relentless support was the sole reason for me to survive through the Ph.D. program and her advices will certainly act as the guiding light for rest of my research career.

I would like to thank all my professors and staff in School of Computational Science and Engineering at Georgia Tech. I owe my special thanks to Prof. Bader, Prof. Fujimoto, Prof. Polo, Prof. Song and Prof. Vuduc for their valuable collaborations. I also take this opportunity to thank Prof. Aluru, Prof. Dilkina, Prof. Poulson and Prof. Jimeng Sun for their time and career guidance. The support of administrative staff Michael Terrell, Mimi Haley, Deanna Richards, Carolyn Young, Rush Holly, Lometa Mitchell and Arlene Washington-Capers need a special mention for they making my life at the school an enjoyable and memorable one.

One of the major driving force of any research is the environment. The lab mates and collaborators play a major role in defining this environment. I would like to specially thank all my lab mates and collaborators Jingu Kim, Da Kuang, Jaegul Choo,

Rundong Du, Daniel Lee, Hannah Kim, Mariya Ishteva, Barry Drake, Richard Boyd, Seungyeon Kim, Longtran Quoc, Yunlong He, Richard Boyd, Hyenkyun Woo and Ashley Beavers for enabling my research with an encouraging environment. I enjoyed working on the summer challenges and some of the interesting research problems with this team. I look forward for opportunities to work with them in future. The servers "jacobi" and"ramanujan" that I assembled with Jaegul need a special mention and many experimental results on this thesis were run on these servers.

The external guides and collaborators play an equal role in this thesis. The research with Dr. Charu Aggarwal and Dr. Grey Malone Ballard were particularly very motivating. During their conversations, I learnt the art of questioning. I also thank Dr. Karthik Subbian and Dr. Dinesh Garg who were always available for long telephonic and thoughtful discussions about building a meaningful research career. I sincerely thank my master thesis advisor Prof. Narahari from Indian Institute of Science (IISc) and all my managers, mentors, friends and collaborators from IISc and IBM who were the primary catalysts for my Ph.D.

I also would like to recollect with gratitude, the support and encouraging words from my brothers and my in-laws, who helped me overcome difficult times and focus on my Ph.D. Last but not the least a big thanks to all my IDH, DAS, Computational Biology and HPC lab friends. I will miss the social fun events and the coffees in break rooms with them. If I can take someone's time always for granted, it would be Piyush and wish this stays forever. Those who want to know this big list of friends, visit the link `https://www.facebook.com/ramakrishnan.kannan/friends`. Finally, lots of love and kisses my little ones – Sakethram and Bhuvaneshi.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Low rank approximation is the problem of finding two low rank factors **W** and **H** such that the $rank(\mathbf{WH}) << rank(\mathbf{A})$ and $\mathbf{A} \approx \mathbf{WH}$. These low rank factors **W** and **H** can be constrained for meaningful physical interpretation and referred as Constrained Low Rank Approximation (CLRA). Like most of the constrained optimization problem, performing CLRA can be computationally expensive than its unconstrained counterpart. A widely used CLRA is the Non-negative Matrix Factorization (NMF) which enforces non-negativity constraints in each of its low rank factors **W** and **H**. In this thesis, we focus on scalable/distributed CLRA algorithms for constraints such as boundedness and non-negativity for large real world matrices that includes text, High Definition (HD) video, social networks and recommender systems.

First, we begin with the Bounded Matrix Low Rank Approximation (BMA) which imposes a lower and an upper bound on every element of the lower rank matrix. BMA is more challenging than NMF as it imposes bounds on the product **WH** rather than on each of the low rank factors **W** and **H**. For very large input matrices, we extend our BMA algorithm to Block BMA that can scale to a large number of processors. In applications, such as HD video, where the input matrix to be factored is extremely large, distributed computation is inevitable and the network communication becomes a major performance bottleneck. Towards this end, we propose a novel distributed Communication Avoiding NMF (CANMF) algorithm that communicates only the right low rank factor to its neighboring machine. Finally, a general distributed HPC-NMF framework that uses HPC techniques in communication intensive NMF operations and suitable for broader class of NMF algorithms.

CHAPTER I

INTRODUCTION

1.1 Motivation

Given a matrix $\mathbf{A}$, low rank approximation is the problem of finding another matrix $\hat{\mathbf{A}}$ such that the $rank(\hat{\mathbf{A}}) << rank(\mathbf{A})$ and $\mathbf{A} \approx \hat{\mathbf{A}}$. Generally, $\hat{\mathbf{A}}$ is defined as a product of two more low rank matrices called low rank factors. For example, the best low rank k approximation for a given input matrix $\mathbf{A}$ is a truncated Singular Value Decomposition (SVD) [19] which is defined as a product of three matrices $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$, such that the $rank(\mathbf{U\Sigma V}^\intercal) = k$. An image can be compressed by taking the low row rank approximation of its matrix representation using SVD. Similarly, in text data – latent semantic indexing, is a dimensionality reduction technique of a term-document matrix using SVD [11]. The other applications include event detection in streaming data, visualization of a document corpus and many more.

With the advent of the internet scale data, many online services such as recommender systems and topic modelling started using low rank approximation techniques. However, towards meaningful physical interpretation, there was a scope for improving classical low rank approximation SVD by introducing additional constraints on low rank factors called as Constrained Low Rank Approximations(CLRA). For different constraints, we need different low rank approximation algorithm to handle the huge volume of data. A widely used CLRA is Non-negative matrix factorization (NMF) where the low rank factors are constrained to be non-negative. That is, given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $k << rank(\mathbf{A})$, NMF is the problem of finding two low rank factors $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ such that $\mathbf{A} \approx \mathbf{WH}$. In the Data Mining and

Machine Learning literature, because of its name, as opposed to low rank approximation, the community liberally calls Non-negative"Matrix Factorization" a matrix factorization. Hence, there is an overlap between low rank approximations and matrix factorizations. We interchangeably use low rank approximation and matrix factorization that is appropriate to understand the application. Towards this end, we formally define Constrained Low Rank Approximations problem without and with any missing elements in the input matrix as below.

$$\begin{aligned} \min_{\mathbf{W},\mathbf{H}} \quad & \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{WH})\|_F^2, \\ \texttt{subject to} \quad & \\ & \texttt{constraints on } \mathbf{W} \\ & \texttt{constraints on } \mathbf{H} \end{aligned} \tag{1}$$

where, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{W}$ is a matrix of size $m \times k$ and $\mathbf{H}$ of size $k \times n$ and $\cdot*$ is hadamard product that is., element-wise matrix multiplication. The matrix $\mathbf{M} \in \{0,1\}^{n \times m}$ is an indicator matrix with missing entries as zero. For example., in the case of recommender systems, the matrix $\mathbf{A}$ represent the row user i's rating for column item j and not all users rate all the items. Hence the entry m_{ij} will be zero if the rating is not observed. The above definition is a general framework that covers the cases with and without any missing elements.

In this thesis, we focus on scalable and distributed CLRA algorithms for constraints such as boundedness and non-negativity for large real world matrices that includes text, High Definition (HD) video, social networks and recommender systems.

1.1.1 Scalable BMA

Recommender system is the problem of estimating the rating of a user for an item, given all the users ratings for a subset of their consumed items and A successful application of low rank approximation for internet data is recommender system called

as matrix factorization. The matrix factorization for recommender system is solved using Stochastic Gradient Descent (SGD) [16] and Alternating Least Squares with Regularization (ALSWR) [59]. However, these approaches have not leveraged the fact that the rating a_{ij} is always bounded within $[r_{min}, r_{max}]$. All the existing algorithms just artificially truncate their final solution to fit within the bounds. We use this signal to find low rank factors $\mathbf{W}$ and $\mathbf{H}$, such that the elements in the product $\mathbf{WH}$ are within a given range. This will guarantee that every estimated rating is within the bounds $[r_{min}, r_{max}]$. We call this low rank approximation as Bounded Matrix Low Rank Approximation (BMA). BMA is different from NMF that it enforces bounding constraints on the product of the low rank factors $\mathbf{WH}$. Whereas, NMF constrains every low rank factors $\mathbf{W}$ and $\mathbf{H}$ to be non-negative. This makes the BMA problem more challenging than NMF.

Formally, the BMA problem for an input matrix $\mathbf{A}$ is defined as[1]

$$\begin{aligned} \min_{\mathbf{W},\mathbf{H}} \quad & \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{WH})\|_F^2 \\ \text{subject to} \quad & \\ & r_{min} \leqslant \mathbf{WH} \leqslant r_{max}. \end{aligned} \tag{2}$$

Recently, there has been many innovations introduced to the naive low rank approximation technique such as considering only neighboring entries during the factorization process, time of the ratings, and implicit ratings such as "user watched but did not rate". Hence, we extended our previous work by designing a bounding framework [24] [23], that scientifically bounds the existing sophisticated recommender systems algorithms. Independently, we investigated using NMF for detecting outliers in text and network data and called it Outlier NMF. In both of these problems, the major bottleneck for scaling to very large matrices was computation of NMF. It was

[1]In our notation, if the inequality is between a vector/matrix and a scalar, every element in the vector/matrix should satisfy the inequality against the scalar.

very difficult to run NMF for very large sparse and dense matrices in shared memory multi core systems with limited memory and compute capacity. Hence, we address this problem by designing a distributed NMF.

1.1.2 Communication Avoiding NMF

The distributed NMF for defacto algorithms such as Multiplicative Update (MU) [37, 39] have been designed by parallelizing every individual operations of the algorithm. For eg., the building blocks of the MU algorithm are matrix multiplication, element-wise multiplication/division and these operations had been optimized for hadoop considering sparse matrix representations. However, distributed NMF algorithm is expensive in communication and the state of the art algorithms do not discuss about communication reduction prohibiting these algorithms to run for large dense matrices such as video data. In this work, we proposed a distributed MPI-based communication avoiding NMF (CANMF) algorithm inspired by parallel Jacobi method and Block Principal Pivoting(BPP) that is well suited for both large sparse and dense matrices. The proposed CANMF algorithm reduced the communication cost from $O((m+n)klogp)$ of the baseline algorithm to $O(\frac{nk}{p})$ for p processes. We compare the performance of our algorithm with commonly used NMF algorithms such as multiplicative update, HALS on sparse/dense synthetic and real world datasets. We also present the scalability results of our CANMF algorithm.

While conducting experiments for the above algorithm, we were exploring HPC based techniques to improve the baseline communication cost. In the case of NMF algorithm the communication intensive operation is matrix multiplication and distributing the matrix for performing NNLS computation. Also, we observed that collective MPI calls were much more efficient during communication over point to point MPI calls.

1.1.3 High Performance Computing Non-negative Matrix Factorization (HPC-NMF)

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 algorithm to solve the problem for big datasets. The current state-of-the-art approaches for NMF with large-scale data have primarily focused on the Map-Reduce programming model with implementations in systems like Hadoop. These implementations are much too slow because they perform more data movement than necessary and because each step of computation involves reading and writing data from disk. Furthermore, because the organization of computations is handled by the run-time system, users cannot control data movement in order to guarantee privacy.

Efficient NMF algorithms must prioritize both data distribution and data movement (i.e., communication). The current trend for high-performance computing (HPC) is that available parallelism (and therefore aggregate computational rate) is increasing much more quickly than improvements in the speed at which data can be transferred between processors and throughout the memory hierarchy. This trend implies that the relative cost of communication (compared to computation) is increasing.

Existing distributed-memory algorithms are limited in terms of performance and applicability, as they are implemented using Hadoop and are designed only for sparse matrices. We carefully designed this parallel algorithm which avoids communication overheads and scales well to modest numbers of cores. CANMFwas particularly performing well only with BPP algorithm and could not be extended to HALS and MU. Using these observations, we propose a new NMF algorithm using HPC based techniques called HPC-NMF that can be useful for distributed implementation of wider

class NMF algorithm such as HALS and MU. In the case that the input matrix $\mathbf{A}$ is dense, HPC-NMF provably minimizes communication costs under mild assumptions.

The overall thesis has the following chapters. In Chapter 2, we discuss relevant foundations that are required to understand this thesis. We start presenting my three key research contributions (1) Bounded Matrix Low Rank Approximation (2) Communication Avoiding NMF and (3) HPC-NMF as the next three chapters. Finally, we conclude the thesis with detailed explanation about our proposed NMF software and other future directions.

CHAPTER II

FOUNDATIONS

In this chapter, we would like to present relevant optimization techniques that are required to rest of the chapters. To begin with we would like to start with the notations used in this Thesis.

A lowercase/uppercase letter such as x or X, is used to denote a scalar; a boldface lowercase letter, such as $\mathbf{x}$, is used to denote a vector; a boldface uppercase letter, such as $\mathbf{X}$, is used to denote a matrix. Indices typically start from 1. When a matrix $\mathbf{X}$ is given, $\mathbf{x}_i$ denotes its i^{th} column, $\mathbf{x}_j^\intercal$ denotes its j^{th} row and x_{ij} or $X(i,j)$ denote its $(i,j)^{th}$ element. For a vector $\mathbf{i}$, $\mathbf{x}(\mathbf{i})$ means that vector $\mathbf{i}$ indexes into the elements of vector $\mathbf{x}$. That is, for $\mathbf{x} = [1,4,7,8,10]$ and $\mathbf{i} = [1,3,5]$, $\mathbf{x}(\mathbf{i}) = [1,7,10]$. We have also borrowed certain notations from matrix manipulation scripts such as Matlab/Octave. For example, the $max(\mathbf{x})$ is the maximal element $x \in \mathbf{x}$ and $max(\mathbf{X})$ is a vector of maximal elements from each column $\mathbf{x} \in \mathbf{X}$.

For the reader's convenience, the notations used in this chapter are summarized in Table 1.

2.1 NMF and Block Coordinate Descent(BCD)

Consider a constrained non-linear optimization problem as follows:

$$\min f(x) \text{ subject to } x \in \mathcal{X}, \tag{3}$$

Table 1: Notations

$\mathbf{A} \in \mathbb{R}^{n \times m}$	Ratings matrix. The missing ratings are indicated by 0, and the given ratings are bounded within $[r_{min}, r_{max}]$.
$\mathbf{M} \in \{0,1\}^{n \times m}$	Indicator matrix. The positions of the missing ratings are indicated by 0, and the positions of the given ratings are indicated by 1.
n	Number of users
m	Number of items
k	Value of the reduced rank
$\mathbf{W} \in \mathbb{R}^{n \times k}$	User-feature matrix. Also called as a low rank factor.
$\mathbf{H} \in \mathbb{R}^{k \times m}$	Feature-item matrix. Also called as a low rank factor.
$\mathbf{w}_x \in \mathbb{R}^{n \times 1}$	x-th column vector of $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_k]$
$\mathbf{h}_x^\intercal \in \mathbb{R}^{1 \times m}$	x-th row vector of $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_k]^\intercal$
$r_{max} > 1$	Maximal rating or upper bound
r_{min}	Minimal rating or lower bound
$\cdot*$	Hadamard product – Element wise matrix multiplication.
$\cdot/$	Element wise matrix division
$\mathbf{A}(:, i)$	$i-th$ column of the matrix $\mathbf{A}$
$\mathbf{A}(i, :)$	$i-th$ row of the matrix $\mathbf{A}$
β	Data structure in memory factor
$memsize(v)$	The approximate memory of a variable v=the product of (the number of elements in v, size of each element, and β)
μ	Mean of all known ratings in $\mathbf{A}$
$\mathbf{p} \in \mathbb{R}^n$	Bias of all users u
$\mathbf{q} \in \mathbb{R}^m$	Bias of all items i
$\mathbf{M}_i$	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

where $\mathcal{X}$ is a closed convex subset of $\mathbb{R}^n$. An important assumption to be exploited in the BCD method is that the set $\mathcal{X}$ is represented by a Cartesian product:

$$\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_l, \tag{4}$$

where $\mathcal{X}_j$, $j = 1, \cdots, l$, is a closed convex subset of $\mathbb{R}^{N_j}$, satisfying $n = \sum_{j=1}^{l} N_j$. Accordingly, vector $\mathbf{x}$ is partitioned as $\mathbf{x} = (\mathbf{x}_1, \cdots, \mathbf{x}_l)$ so that $\mathbf{x}_j \in \mathcal{X}_j$ for $j = 1, \cdots, l$. The BCD method solves for $\mathbf{x}_j$ fixing all other subvectors of $\mathbf{x}$ in a cyclic manner. That is, if $\mathbf{x}^{(i)} = (\mathbf{x}_1^{(i)}, \cdots, \mathbf{x}_l^{(i)})$ is given as the current iterate at the i^{th} step, the algorithm generates the next iterate $\mathbf{x}^{(i+1)} = (\mathbf{x}_1^{(i+1)}, \cdots, \mathbf{x}_l^{(i+1)})$ block by block, according to the solution of the following subproblem:

$$\mathbf{x}_j^{(i+1)} \leftarrow \underset{\xi \in \mathcal{X}_j}{\operatorname{argmin}} f(\mathbf{x}_1^{(i+1)}, \cdots, \mathbf{x}_{j-1}^{(i+1)}, \xi, \mathbf{x}_{j+1}^{(i)}, \cdots, \mathbf{x}_l^{(i)}). \tag{5}$$

Also known as a *non-linear Gauss-Seidel* method [3], this algorithm updates one block each time, always using the most recently updated values of other blocks $\mathbf{x}_{\tilde{j}}, \tilde{j} \neq j$. This is important since it ensures that after each update the objective function value does not increase. For a sequence $\left\{\mathbf{x}^{(i)}\right\}$ where each $\mathbf{x}^{(i)}$ is generated by the BCD method, the following property holds.

Theorem 1. *Suppose f is continuously differentiable in $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_l$, where $\mathcal{X}_j$, $j = 1, \cdots, l$, are closed convex sets. Furthermore, suppose that for all j and i, the minimum of*

$$\min_{\xi \in \mathcal{X}_j} f(\mathbf{x}_1^{(i+1)}, \cdots, \mathbf{x}_{j-1}^{(i+1)}, \xi, \mathbf{x}_{j+1}^{(i)}, \cdots, \mathbf{x}_l^{(i)})$$

is uniquely attained. Let $\left\{\mathbf{x}^{(i)}\right\}$ be the sequence generated by the block coordinate descent method as in Eq. (5). Then, every limit point of $\left\{\mathbf{x}^{(i)}\right\}$ is a stationary point. The uniqueness of the minimum is not required when $l = 2$ [20].

The proof of this theorem for an arbitrary number of blocks is shown in Bertsekas [3]. For a non-convex optimization problem, often we can expect the stationarity of a limit point [38] from a good algorithm.

The two key questions of the BCD method are (a) Determining the the partition of $\mathcal{X}$ such that its cartesian product constitutes $\mathcal{X}$. Typically the partition that is computationally efficient is preferred. For eg., the partition that can result in a closed form solution for the subproblem. (b) Also, the BCD method requires the most recent values be used for each problem in Eq. (5). Thus, the partition strategies that are independent from each others are preferred so that we can parallelize the computations. In case, there are dependency among the partitions, the update order of the partition is preserved to guarantee the most recent values are always used.

2.1.1 BCD with Two Matrix Blocks - ANLS Method

For convenience, we first assume all the elements of the input matrix are known and hence we ignore $\mathbf{M}$ from the discussion. The most natural partitioning of the variables is to have two big blocks, $\mathbf{W}$ and $\mathbf{H}$. In this case, following the BCD method in Eq. (5), we take turns solving

$$\mathbf{W} \leftarrow \text{argmin}_{\mathbf{W} \geq 0} f(\mathbf{W}, \mathbf{H}) \text{ and } \mathbf{H} \leftarrow \text{argmin}_{\mathbf{H} \geq 0} f(\mathbf{W}, \mathbf{H}). \tag{6}$$

Since the subproblems are non-negativity constrained least squares (NLS) problems, the two-block BCD method has been called the alternating non-negative least square (ANLS) framework [38, 26, 29].

2.1.2 BCD with 2k Vector Blocks - HALS/RRI Method

Let us now partition the unknowns into 2k blocks in which each block is a column of $\mathbf{W}$ or a row of $\mathbf{H}$, as explained in Figure 1. In this case, it is easier to consider the

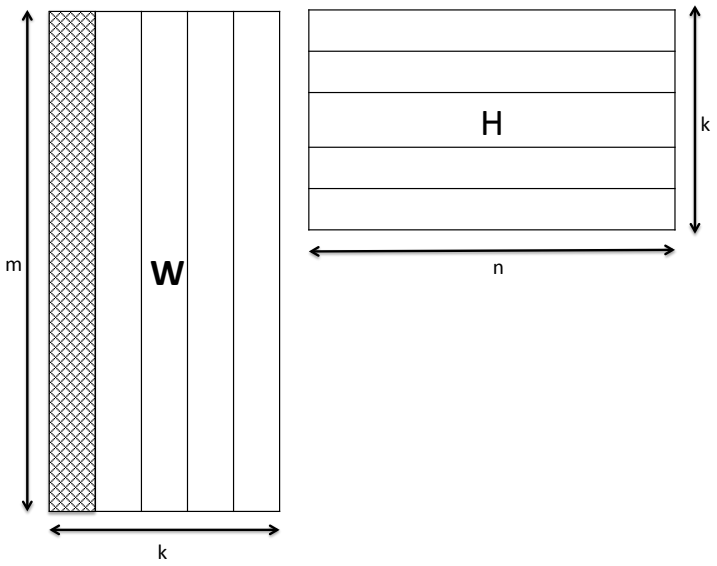


Figure 1: BCD with 2k Vector Blocks

objective function in the following form:

$$f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal) = \|\mathbf{A} - \sum_{j=1}^{k} \mathbf{w}_j \mathbf{h}_j^T\|_F^2, \tag{7}$$

where $\mathbf{W} = [\mathbf{w}_1, \cdots \mathbf{w}_k] \in \mathbb{R}_+^{n \times k}$ and $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_k]^\intercal \in \mathbb{R}_+^{k \times m}$. The form in Eq. (7) represents that $\mathbf{A}$ is approximated by the sum of k rank-one matrices.

Following the BCD scheme, we can minimize f by iteratively solving

$$\mathbf{w}_i \leftarrow \text{argmin}_{\mathbf{w}_i \geq 0} f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal)$$

for $i = 1, \cdots, k$, and

$$\mathbf{h}_i^\intercal \leftarrow \text{argmin}_{\mathbf{h}_i^\intercal \geq 0} f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal)$$

for $i = 1, \cdots, k$.

The $2k$-block BCD algorithm has been studied as Hierarchical Alternating Least

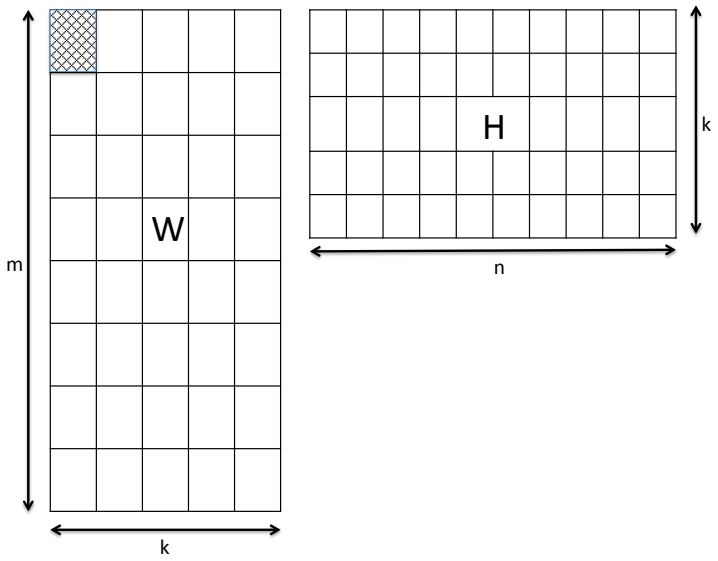


Figure 2: BCD with k(n + m) Scalar Blocks

Squares (HALS) proposed by Cichocki et al. [9, 8] and independently by Ho et al. [22] as rank-one residue iteration (RRI).

2.1.3 BCD with k(n + m) Scalar Blocks

We can also partition the variables with the smallest $k(n + m)$ element blocks of scalars as in Figure 2, where every element of $\mathbf{W}$ and $\mathbf{H}$ is considered as a block in the context of Theorem 1. To this end, it helps to write the objective function as a quadratic function of scalar w_{ij} or h_{ij} assuming all other elements in $\mathbf{W}$ and $\mathbf{H}$ are fixed:

$$f(w_{ij}) = \|(\mathbf{a}_i^\intercal - \sum_{\tilde{k}\neq j} w_{i\tilde{k}}\mathbf{h}_{\tilde{k}}^\intercal) - w_{ij}\mathbf{h}_j^\intercal\|_2^2 + \text{const}, \tag{8a}$$

$$f(h_{ij}) = \|(\mathbf{a}_j - \sum_{\tilde{k}\neq i} \mathbf{w}_{\tilde{k}}h_{\tilde{k}j}) - \mathbf{w}_i h_{ij}\|_2^2 + \text{const}, \tag{8b}$$

where $\mathbf{a}_i^\intercal$ and $\mathbf{a}_j$ denote the i^{th} row and the j^{th} column of $\mathbf{A}$, respectively.

Kim et al. [27] discuss about NMF using BCD method.

2.2 NMF Algorithms

We will see three reference algorithms for each of these different common matrix partitions. To address the 2 Block BCD method, we begin an active set based method called ANLS-BPP (Alternating Non-negative Least Square-Block Principal Pivoting) proposed by Kim and Park in [26, 29].

2.2.1 Alternating Non-negative Least Squares-Block Principal Pivoting

According to the ANLS framework, we first partition the variables of the NMF problem into two blocks $\mathbf{W}$ and $\mathbf{H}$. Then we solve the following equations iteratively until a stopping criteria is satisfied.

$$\begin{aligned} \mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \tilde{\mathbf{W}}\mathbf{H} \right\|_F^2, \\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \mathbf{W}\tilde{\mathbf{H}} \right\|_F^2. \end{aligned} \tag{9}$$

The optimizations sub-problem for $\mathbf{W}$ and $\mathbf{H}$ are NLS problems which can be solved by a number of methods from generic constrained convex optimization to active-set methods. Typical approaches use form the normal equations of the least squares problem (and then solve them enforcing the non-negativity constraint), which involves matrix multiplications of the factor matrices with the data matrix. We focus on and use the block principal pivoting [30] method to solve the non-negative least squares problem, as it is the fastest algorithm (in terms of number of iterations).

BPP is the state-of-the-art method for solving the NLS subproblems in Eq. (40). The main sub-routine of BPP is the single right-hand side NLS problem

$$\min_{\mathbf{x} \geqslant 0} \|\mathbf{C}\mathbf{x} - \mathbf{b}\|_2^2. \tag{10}$$

The Karush-Kuhn-Tucker (KKT) optimality condition for Eq. (10) is as follows

$$\mathbf{y} = \mathbf{C}^T\mathbf{C}\mathbf{x} - \mathbf{C}^T\mathbf{b} \tag{11a}$$

$$\mathbf{y} \geqslant 0 \tag{11b}$$

$$\mathbf{x} \geqslant 0 \tag{11c}$$

$$\mathbf{x}^T\mathbf{y} = 0. \tag{11d}$$

The KKT condition (11) 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. (11) 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 it 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 active set and passive set. Since the above problem is convex, the correct partition of the optimal solution will satisfy the KKT condition (Eq. (11)). The BPP algorithm greedily swaps indices between the 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 [30], discuss the BPP algorithm in further detail.

Apart from BPP there are many other methods that can be used such as projected gradient, conjugate gradient, interior point methods etc. Chen and Plemmons provide

a detailed survey of NLS methods in [7]. One computational advantage of BPP method in [30], is its ability to solve multiple right hand sides. In the case of NMF problem, the size of m and n will be in the order of millions. Hence, for such very large scale ANLS problems with many right hand sides, it is preferable to choose an NLS method similar to BPP that can solve multiple right hand sides.

2.2.2 Hierarchical Alternating Least Squares(HALS)

In the case of HALS algorithm, we find every row of $\mathbf{W}$ and column of $\mathbf{H}$ keeping the other row/column blocks fixed by solving objective function (7). Consider $\mathbf{E}$ as the difference of the input matrix $\mathbf{A}$ with sum of $k-1$ rank-one matrices except x. That is.,

$$\mathbf{E} = \mathbf{A} - \sum_{j=1, j\neq x}^{k} \mathbf{w}_j \mathbf{h}_j^T$$

and solve for $\mathbf{w}_x$ given $\mathbf{h}_x$ and vice-versa. This can be formally represented as

$$\mathbf{w}_x = \underset{\mathbf{w}_x \geqslant 0}{\operatorname{argmin}} \|\mathbf{E} - \mathbf{w}_x \mathbf{h}_x^T\|_F^2. \tag{12}$$

$$\mathbf{h}_x = \underset{\mathbf{h}_x \geqslant 0}{\operatorname{argmin}} \|\mathbf{E} - \mathbf{w}_x \mathbf{h}_x^T\|_F^2. \tag{13}$$

Unlike the active set based NNLS solution explained in ANLS-BPP, The above problem has a closed form equation.

Theorem 2. *For each of the minimization problem in* (12)*, the solutions are*

$$\mathbf{w}_x = \frac{[\mathbf{E}\mathbf{h}_x^T]_+}{\mathbf{h}_x \mathbf{h}_x^T} \tag{14}$$

$$\mathbf{h}_x = \frac{[\mathbf{E}^T \mathbf{w}_x]_+}{\mathbf{w}^T \mathbf{w}_x} \tag{15}$$

The proof for this theorem is discussed in [27]. HALS algorithm computes the k columns vectors of $\mathbf{w} \in \mathbb{R}_+^m$ and k row vectors $\mathbf{h}^T \in \mathbb{R}_+^n$ using the closed form

equation from the above Theorem 2.

It is imperative to differentiate the HALS and ANLS-BPP from the algorithmic perspective. In the case of ANLS-BPP, the algorithm determines m or n independent vectors of size $\mathbb{R}_+^k$. Whereas, HALS finds k vectors of size $\mathbb{R}_+^m$ or $\mathbb{R}_+^n$ one-by-one as it is dependent to use the most recent blocks. Remember $k << min(m, n)$ and typically in the order of 100's for the input matrices of dimensions in order of millions. Hence, from the implementation stand point, ANLS-BPP is embarrassingly parallelizable over HALS.

2.2.3 Multiplicative Update(MU)

Multiplicative Update (MU) is the popular algorithm for NMF. The elements of the w_{ij} and h_{ij} are updated only based on the multiplication of the matrices and hence the name Multiplicative Update. It was first proposed by Lee and Seung [36] to learn parts of objects that could not be addressed by LSI which produced the basis vectors with mixed signs. The update equations for $\mathbf{W}$ and $\mathbf{H}$ are

$$w_{mk} = \frac{(\mathbf{AH})_{mk}}{(\mathbf{WHH}^T)_{mk}} \tag{16}$$

$$h_{kn} = \frac{(\mathbf{A}^T\mathbf{W})_{nk}}{(\mathbf{W}^T\mathbf{WH})_{nk}} \tag{17}$$

With this necessary introduction of three different NMF algorithms, we compare and contrast the performance of these algorithms over number of iterations and by sweeping the low rank k on different nature of datasets as described in 2. In all the cases above, consistently ANLS-BPP outperforms the other popular NMF algorithms. A detailed comparison with more algorithm is performed by Kim et.al in [29] and [27]. Also, The convergence properties of these different algorithms are discussed in detail by Kim, He and Park [28]. The observation can be attributed to the reason that

ANLS-BPP solves the subproblem to optimality as opposed to other algorithms.

Table 2: Realworld and Synthetic Datasets

Datasets	Type	Size	NNZ
Synthetic	Dense	100000x50000	
Video	Dense	1.3 millionx2400	
Synthetic	Sparse	200000x100000	2×10^6
Wiki	Sparse	387086x736048	55821923

At this juncture, we would like to establish the computational connection among these three algorithms.

2.3 Alternating Updating NMF

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 $\mathbf{W}$. We restrict attention to the class of NMF algorithms that use the Gram matrix – a matrix that is formed by the inner products of the individual vectors; 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.

input : $\mathbf{A}$ is an $m \times n$ matrix, k is rank of approximation
output: $\mathbf{W} \in \mathbb{R}_+^{m \times k}, \mathbf{H} \in \mathbb{R}_+^{k \times n}$
Initialize $\mathbf{H}$ with a non-negative matrix in $\mathbb{R}_+^{n \times k}$;
while *stopping criteria not satisfied* **do**
    Update $\mathbf{W}$ using $\mathbf{H}\mathbf{H}^T$ and $\mathbf{A}\mathbf{H}^T$;
    Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$;

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

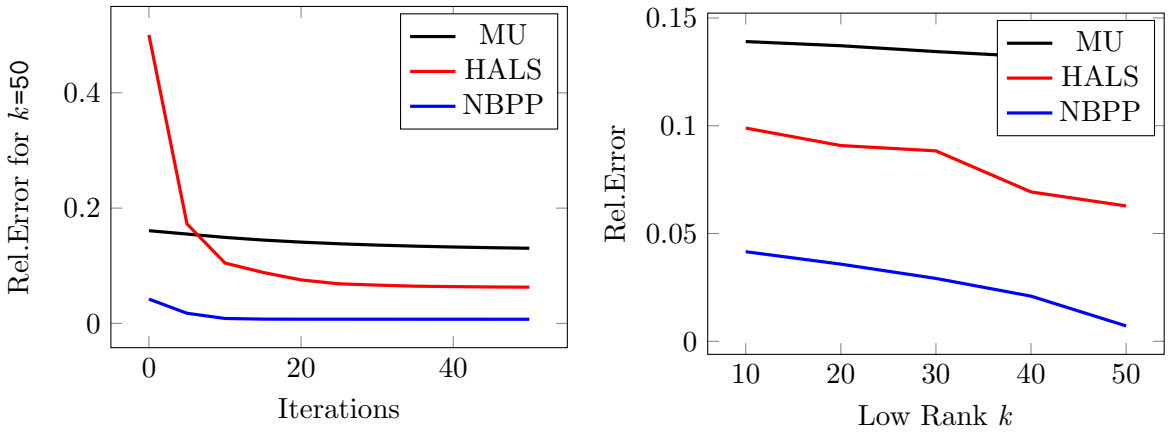


Figure 3: Comparison of NMF Algorithms – Dense Synthetic

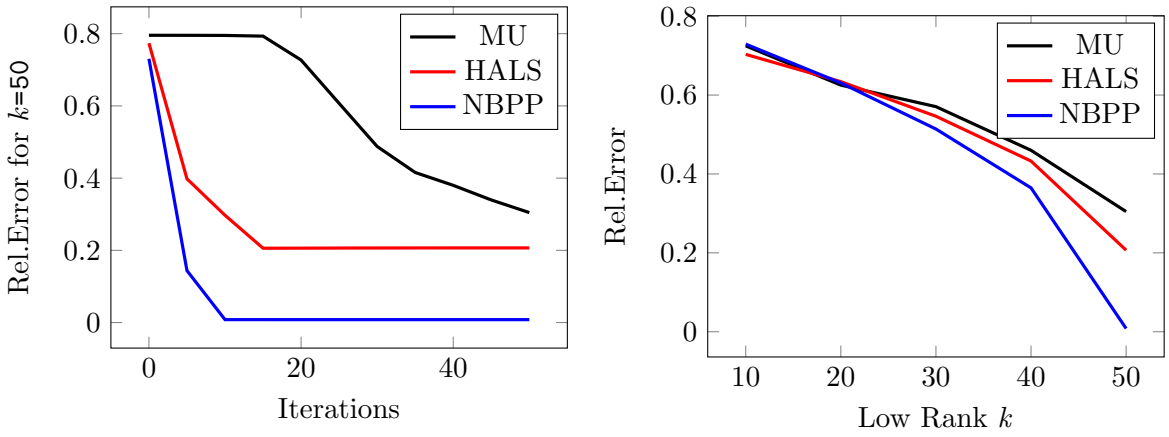


Figure 4: Comparison of NMF Algorithms – Sparse Synthetic

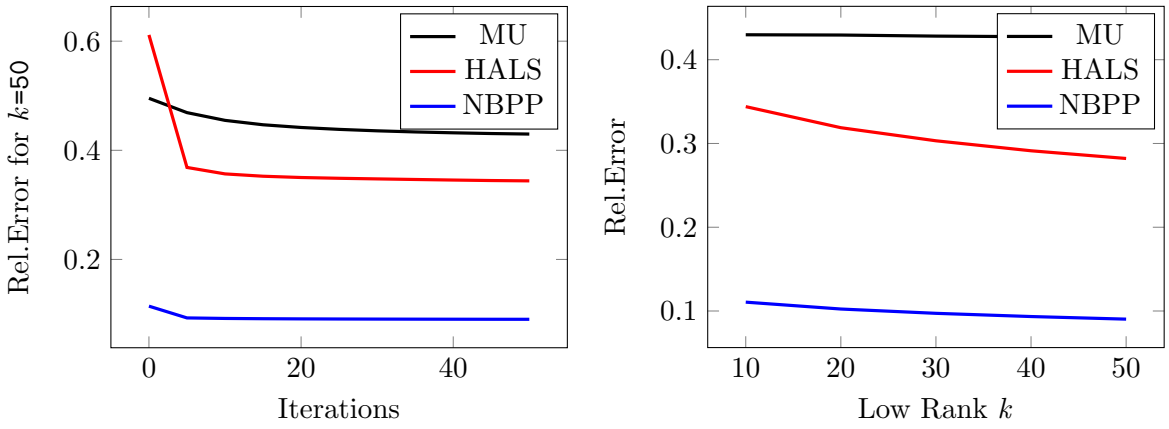


Figure 5: Comparison of NMF Algorithms – Video Realworld

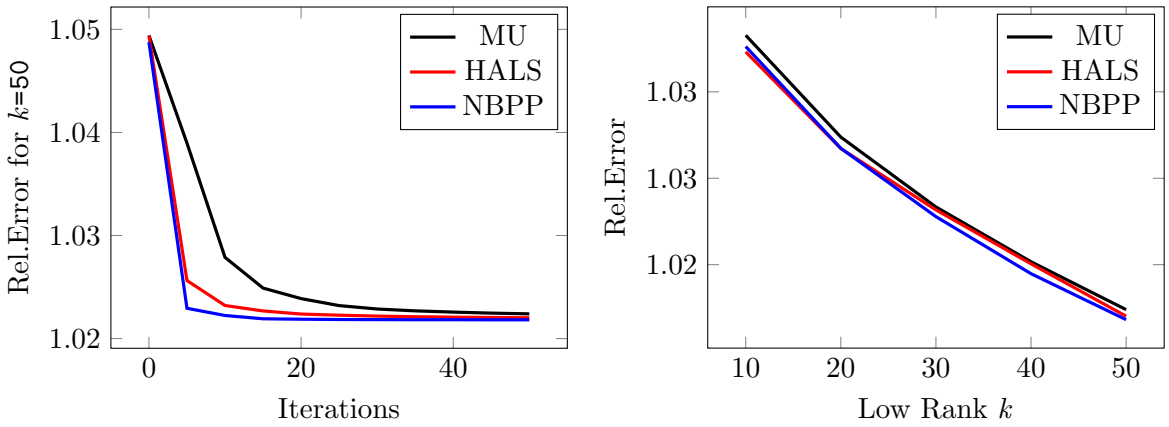


Figure 6: Comparison of NMF Algorithms – Sparse Wiki Data

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 $\mathbf{W}$:

$$\begin{aligned} \mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \tilde{\mathbf{W}}\mathbf{H} \right\|_F, \\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \mathbf{W}\tilde{\mathbf{H}} \right\|_F. \end{aligned} \tag{18}$$

Since each subproblem involves nonnegative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [28]. Block Principal Pivoting (BPP), 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.

As explained above, The other popular NMF algorithms $HALS$ and MU that update the factor matrices alternatively but does not maximally reduce the objective function value each time, in the same sense as in ANLS. These updates do not necessarily solve each of the subproblems (40) to optimality but simply improve the overall objective function (29). To show how these methods can fit into the AU-NMF framework, we discuss them in more detail.

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) according to equations (42), the method falls into the AU-NMF framework. After computing the matrices $\mathbf{H}\mathbf{H}^T$, $\mathbf{A}\mathbf{H}^T$, $\mathbf{W}^T\mathbf{W}$, 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. Instead of performing these $(m+n)k$ in an arbitrary order, if all of $\mathbf{W}$ is updated before $\mathbf{H}$ (or vice-versa) according to equations (16), 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 $\mathbf{H}$.

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. This framework that helps to computationally position $ANLS-BPP, MU$ and $HALS$ through a single framework is useful in many conversations in this thesis.

With these relevant foundations, we would like to present the core technical contributions of this thesis – (a) Bounded Matrix Low Rank Approximation (b) Communication Avoiding NMF and (c) HPC-NMF.

CHAPTER III

BOUNDED MATRIX LOW RANK APPROXIMATION

In this chapter, we are considering low rank approximation for input matrices that is bounded. For eg., in the case of recommender systems rating matrix,the input matrix **A** is bounded in between $[r_{min}, r_{max}]$ such as $[1, 5]$. We also propose a new improved scalable low rank approximation algorithm for such bounded matrices called Bounded Matrix Low Rank Approximation(BMA) that bounds every element of the approximation **WH**. We also present an alternate formulation to bound existing recommender system algorithms called BALS and discuss its convergence. Our experiments on real world datasets illustrate that the proposed method BMA outperforms the state of the art algorithms for recommender system such as Stochastic Gradient Descent, Alternating Least Squares with regularization, SVD++ and Bias-SVD on real world data sets such as Jester, Movielens, Book crossing, Online dating and Netflix.

3.1 Motivation

Low rank approximations vary depending on the constraints imposed on the factors as well as the measure for the difference between **A** and **WH**. Low rank approximations have produced a huge amount of interest in the data mining and machine learning communities due to its effectiveness for addressing many foundational challenges in these application areas. A few prominent techniques of machine learning that use low rank approximation are principal component analysis, factor analysis, latent semantic analysis, and non-negative matrix factorization (NMF), to name a few.

Over the last decade, NMF has emerged as an important low rank approximation technique, where the low-rank factor matrices are constrained to have only non-negative elements. In this chapter, we propose a new type of low rank approximation

where the elements of the approximation are bounded – that is, its elements are within a given range. We call this new low rank approximation Bounded Matrix Low Rank Approximation (BMA). BMA is different from NMF in that it imposes both upper and lower bounds *on its product* **WH** rather than non-negativity in each of the low rank factors $\mathbf{W} \geqslant 0$ and $\mathbf{H} \geqslant 0$. Thus, the goal is to obtain a lower rank approximation **WH** of a given input matrix **A**, where the elements of **WH** and **A** are bounded.

Let us consider a numerical example to appreciate the difference between NMF and BMA. Consider the 4x6 matrix with all entries between 1 and 10. In the Figure 7, the output low rank approximation is shown between BMA and NMF by running only one iteration for low rank 3. It is important to observe the following.

- All the entries in the BMA are bounded between 1 and 10, whereas, approximation generated out of NMF is not bounded in the same range of input matrix. This difference can be hugely pronounced in the case of very large input matrix and the current practice is when an entry in the low rank approximation is beyond the bounds, it is artificially truncated.

- In the case of BMA, as opposed to NMF, every individual low rank factors are unconstrained and takes even negative values.

In order to address the problem of an input matrix with missing elements, we will formulate a BMA that imposes bounds on a low rank matrix that is the best approximate for such matrices. The algorithm design considerations are – (1) Simple implementation (2) Scalable to large data and (3) Easy parameter tuning with no hyper parameters.

Formally, the BMA problem for an input matrix **A** is defined as

$$\mathbf{A} = \begin{pmatrix} 4 & 2 & 6 & 6 & 1 & 6 \\ 9 & 5 & 9 & 8 & 2 & 9 \\ 2 & 9 & 1 & 6 & 4 & 1 \\ 10 & 8 & 10 & 2 & 9 & 1 \end{pmatrix}$$

Input Bounded Matrix $\mathbf{A} \in [1, 10]$

$$\mathbf{A}'_{BMA} = \begin{pmatrix} 4.832 & 2.233 & 5.118 & 5.824 & 1.000 & 5.878 \\ 8.719 & 4.839 & 9.092 & 8.104 & 2.490 & 9.012 \\ 1.983 & 9.017 & 1.661 & 5.957 & 3.861 & 1.000 \\ 10.000 & 7.896 & 10.000 & 2.106 & 4.871 & 5.599 \end{pmatrix}$$

BMA Output $\mathbf{A}_{BMA}$. The error $\|\mathbf{A} - \mathbf{A}'_{BMA}\|_F^2$ is 40.621.

$$\mathbf{A}'_{NMF} = \begin{pmatrix} 4.939 & 3.378 & 5.312 & 5.021 & 3.131 & 4.736 \\ 8.066 & 5.794 & 8.696 & 8.495 & 4.649 & 8.036 \\ 5.850 & 4.600 & 6.241 & 4.548 & 4.036 & 4.343 \\ 9.693 & 7.549 & \mathbf{10.225} & 5.165 & 8.303 & 4.958 \end{pmatrix}$$

NMF Output $\mathbf{A}_{NMF}$. The error $\|\mathbf{A} - \mathbf{A}'_{NMF}\|_F^2$ is 121.59.

$$\begin{pmatrix} 4.138 & -0.002 & 7.064 \\ 7.037 & -0.008 & 9.429 \\ 3.730 & 0.031 & -4.844 \\ 6.776 & -0.020 & 1.000 \end{pmatrix} \begin{pmatrix} 1.214 & 1.384 & 1.202 & 0.753 & 0.734 & 0.768 \\ -90.835 & 49.650 & -92.455 & 170.147 & -9.311 & -0.525 \\ -0.058 & -0.478 & -0.011 & 0.441 & -0.292 & 0.382 \end{pmatrix}$$

BMA's Left and Right Low Rank Factors $\mathbf{W}_{BMA}$ and $\mathbf{H}_{BMA}$

$$\begin{pmatrix} 4.021 & 2.806 & 0.153 \\ 8.349 & 4.179 & 0.000 \\ 7.249 & 1.216 & 0.000 \\ 10.015 & 0.725 & 0.467 \end{pmatrix} \qquad \begin{pmatrix} 0.727 & 0.605 & 0.770 & 0.431 & 0.557 & 0.416 \\ 0.478 & 0.178 & 0.543 & 1.172 & 0.000 & 1.092 \\ 4.421 & 2.920 & 4.538 & 0.000 & 5.835 & 0.000 \end{pmatrix}$$

NMF's Non-negative Left and Right Low Rank Factors $\mathbf{W}_{NMF}$ and $\mathbf{H}_{NMF}$

Figure 7: Numerical Motivation for BMA

$$\begin{aligned} \min_{\mathbf{W},\mathbf{H}} \quad & \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{WH})\|_F^2 \\ \text{subject to} \quad & \\ & r_{min} \leqslant \mathbf{WH} \leqslant r_{max}, \end{aligned} \tag{19}$$

where r_{min} and r_{max} are the bounds and $\|\cdot\|_F$ stands for the Frobenius norm. In the case of an input matrix with missing elements, the low rank matrix is approximated only against the known elements of the input matrix. Hence, during error computation the filter matrix $\mathbf{M}$ includes only the corresponding elements of the low rank $\mathbf{WH}$ for which the values are known. Thus, $\mathbf{M}$ has '1' everywhere for input matrix $\mathbf{A}$ with all known elements. However, in the case of a recommender system, the matrix $\mathbf{M}$ has zero for each of the missing elements of $\mathbf{A}$. In fact, for recommender systems, typically only 1 or 2% of all matrix elements are known.

It should be pointed out that an important application for the above formulation is recommender systems, where the community refers to it as a matrix factorization. The unconstrained version of the above formulation (19), was first solved using Stochastic Gradient Descent (SGD) [16] and Alternating Least Squares with Regularization (ALSWR) [59]. However, we have observed that previous research has not leveraged the fact that all the ratings $r_{ij} \in \mathbf{A}$ are bounded within $[r_{min}, r_{max}]$. All existing algorithms *artificially truncate* their final solution to fit within the bounds.

Recently, there has been many innovations introduced into the the naive low rank approximation technique such as considering only neighboring entries during the factorization process, time of the ratings, and implicit ratings such as "user watched but did not rate". Hence, it is important to design a bounding framework that seamlessly integrates into the existing sophisticated recommender systems algorithms.

Let $f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H})$ be an existing recommender system algorithm that can predict all the (u, i) ratings, where $\mathbf{\Theta} = \{\boldsymbol{\theta}_1, \cdots, \boldsymbol{\theta}_l\}$ is the set of parameters apart from the low rank factors $\mathbf{W}, \mathbf{H}$. For example, in the recommender system context, certain implicit

signals are combined with the explicit ratings such as user watched a movie till the end but didn't rate it. We have to learn weights for such implicit signals to predict a user's rating. Such weights are represented as parameter $\mathbf{\Theta}$. For simplicity, we are slightly abusing the notation here. The $f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H})$ either represents estimating a particular value of (u, i) pair or it represents the complete estimated low rank k matrix $\hat{\mathbf{A}} \in \mathbb{R}^{n \times m}$. The ratings from such recommender system algorithms can be scientifically bounded by the following optimization problem based on low rank approximation to determine the unknown ratings.

$$\begin{aligned} \min_{\mathbf{\Theta}, \mathbf{W}, \mathbf{H}} \quad & \|\mathbf{M} \cdot *(\mathbf{A} - f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H}))\|_F^2 \\ \text{subject to} \quad & \\ & r_{min} \leqslant f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H}) \leqslant r_{max}. \end{aligned} \tag{20}$$

Traditionally, regularization is used to control the low rank factors $\mathbf{W}$ and $\mathbf{H}$ from taking larger values. However, this does not guarantee that the value of the product $\mathbf{WH}$ is in the given range. We also experimentally show that introducing the bounds on the product of $\mathbf{WH}$ outperforms the low rank approximation algorithms with regularization.

In this chapter, we present a survey of current state-of-the-art and the solution the Problems (19) and (20) will be presented. WE also describe the implementable algorithms and scalable techniques for solving large scale problems in multi core system with low memory. Finally, we present substantial experimental results illustrating that the proposed methods outperform the state-of-the-art algorithms for recommender systems such as Stochastic Gradient Descent, Alternating Least Squares with regularization, SVD++, Bias-SVD on real world data sets such as Jester, Movielens, Book crossing, Online dating and Netflix. This chapter is based primarily on our earlier work [23, 24].

3.2 Related Work

This section introduces the BMA and its application to recommender systems and reviews some of the prior research in this area. Following this section is a brief overview of our contributions.

The important milestones in matrix factorization for recommender systems have been achieved due to the Netflix competition (http://www.netflixprize.com/) where the winners were awarded 1 million US Dollars as grand prize.

Funk [16] first proposed matrix factorization for recommender system based on SVD, commonly called the Stochastic Gradient Descent (SGD) algorithm. Paterek [47] improved SGD by combining matrix factorization with baseline estimates. Koren, a member of the winning team of the Netflix prize, improved the results with his remarkable contributions in this area. Koren [32] proposed a baseline estimate based on mean rating, user–movie bias, combined with matrix factorization and called it Bias-SVD. In SVD++ [32], he extended this Bias-SVD with implicit ratings and considered only the relevant neighborhood items during matrix factorization. The Netflix dataset also provided the time of rating. However most of the techniques did not include time in their model. Koren [33] proposed time-svd++, where he extended his previous SVD++ model to include the time information. So far, all matrix factorization techniques discussed here are based on SVD and used gradient descent to solve the problem. Alternatively, Zhou et al. [59] used alternating least squares with regularization (ALSWR). Apart from these directions, there had been other approaches such as Bayesian tensor factorization [55], Bayesian probabilistic modelling [48], graphical modelling of the recommender system problem [43] and weighted low-rank approximation with zero weights for the missing values [44]. One of the recent works by Yu et al. [57] also uses coordinate descent to matrix factorization for recommender system. However, they study the tuning of coordinate descent optimization techniques for a parallel scalable implementation of matrix factorization

for recommender system. A detailed survey and overview of matrix factorization for recommender systems is given in [34].

3.2.1 Our Contributions

Given the above background, we highlight our contributions. We propose a novel matrix factorization called Bounded Matrix Low Rank Approximation (BMA) which imposes a lower and an upper bound for the estimated values of the missing elements in the given matrix. We solve the BMA using block coordinate descent method. From this perspective, this is the first work that uses the block coordinate descent method and experiment BMA for recommender systems. We present the details of the algorithm with supporting technical details and a scalable version of the naive algorithm. It is also important to study imposing bounds for existing recommender systems algorithms. We also propose a novel framework for Bounding existing ALS algorithms (called BALS). Also, we test our BMA algorithm, BALS framework on real world datasets and compare against state of the art algorithms SGD, SVD++, ALSWR and Bias-SVD.

3.3 Foundations

In the case of low rank approximation using NMF, the low rank factor matrices are constrained to have only non-negative elements. However, in the case of BMA, we constrain the elements of their product with an upper and lower bound rather than each of the two low rank factor matrices. In the section 2.1, we explained our BCD framework for NMF. For the convenience of the readers, we begin explaining using 2k Block BCD for NMF and subsequently its extension to solve BMA.

Consider partitioning the unknowns $\mathbf{W}$ and $\mathbf{H}$ into 2k blocks in which each block is a column of $\mathbf{W}$ or a row of $\mathbf{H}$, as explained in Figure 1. In this case, it is easier to

consider the objective function in the following form:

$$f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal) = \|\mathbf{A} - \sum_{j=1}^{k} \mathbf{w}_j \mathbf{h}_j^T\|_F^2, \tag{21}$$

where $\mathbf{W} = [\mathbf{w}_1, \cdots \mathbf{w}_k] \in \mathbb{R}_+^{n \times k}$ and $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_k]^\intercal \in \mathbb{R}_+^{k \times m}$. The form in Eq. (21) represents that $\mathbf{A}$ is approximated by the sum of k rank-one matrices.

Following the BCD scheme, we can minimize f by iteratively solving

$$\mathbf{w}_i \leftarrow \text{argmin}_{\mathbf{w}_i \geq 0} f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal)$$

for $i = 1, \cdots, k$, and

$$\mathbf{h}_i^\intercal \leftarrow \text{argmin}_{\mathbf{h}_i^\intercal \geq 0} f(\mathbf{w}_1, \cdots, \mathbf{w}_k, \mathbf{h}_1^\intercal, \cdots, \mathbf{h}_k^\intercal)$$

for $i = 1, \cdots, k$.

The $2k$-block BCD algorithm has been studied as Hierarchical Alternating Least Squares (HALS) proposed by Cichocki et al. [9, 8] and independently by Ho et al. [22] as rank-one residue iteration (RRI).

3.3.1 Bounded Matrix Low Rank Approximation

The building blocks of BMA are column vectors $\mathbf{w}_x$ and row vectors $\mathbf{h}_x^\intercal$ of the matrix $\mathbf{W}$ and $\mathbf{H}$ respectively. In this section, we discuss the idea behind finding these vectors $\mathbf{w}_x$ and $\mathbf{h}_x^\intercal$ such that all the elements in $\mathbf{T} + \mathbf{w}_x \mathbf{h}_x^\intercal \in [r_{min}, r_{max}]$ and the error $\|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{WH})\|_F^2$ is reduced. Here, $\mathbf{T} = \sum_{j=1, j \neq x}^{k} \mathbf{w}_j \mathbf{h}_j^\intercal$.

Problem (19) can be equivalently represented with a set of rank-one matrices $\mathbf{w}_x \mathbf{h}_x^\intercal$

as

$$\begin{aligned} \min_{\mathbf{w}_x, \mathbf{h}_x} \quad & \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{T} - \mathbf{w}_x \mathbf{h}_x^\intercal)\|_F^2 \\ & \forall x = [1, k] \\ \text{subject to} \quad & \\ & \mathbf{T} + \mathbf{w}_x \mathbf{h}_x^\intercal \leqslant r_{max} \\ & \mathbf{T} + \mathbf{w}_x \mathbf{h}_x^\intercal \geqslant r_{min} \end{aligned} \tag{22}$$

Thus, we take turns solving for $\mathbf{w}_x$ and $\mathbf{h}_x^\intercal$. That is, assume we know $\mathbf{w}_x$ and find $\mathbf{h}_x^\intercal$ and vice versa. In the entire section we assume fixing column $\mathbf{w}_x$ and finding row $\mathbf{h}_x^\intercal$. Without loss of generality, all the discussion pertaining to finding $\mathbf{h}_x^\intercal$ with fixed $\mathbf{w}_x$ hold for the other scenario of finding $\mathbf{w}_x$ with fixed $\mathbf{h}_x^\intercal$.

There are different orders of updates of vector blocks when solving Problem (22). For example,

$$\mathbf{w}_1 \rightarrow \mathbf{h}_1^\intercal \rightarrow \cdots \rightarrow \mathbf{w}_k \rightarrow \mathbf{h}_k^\intercal \tag{23}$$

and

$$\mathbf{w}_1 \rightarrow \cdots \rightarrow \mathbf{w}_k \rightarrow \mathbf{h}_1^\intercal \rightarrow \cdots \rightarrow \mathbf{h}_k^\intercal. \tag{24}$$

Kim et al. [27] prove that Eq. (21) satisfies the formulation of BCD method. Eq. (21) when extended with the matrix $\mathbf{M}$ becomes Eq. (22). Here, the matrix $\mathbf{M}$ is like a filter matrix that defines the elements of $(\mathbf{A} - \mathbf{T} - \mathbf{w}_x \mathbf{h}_x^\intercal)$ to be included for the norm computation. Thus, Problem (22) is similar to Problem (21) and we can solve by applying $2k$ block BCD to update $\mathbf{w}_x$ and $\mathbf{h}_x^\intercal$ iteratively, although equation (22) appears not to satisfy the BCD requirements directly. We focus on the scalar block case, as it is convenient to explain regarding imposing bounds on the product of the low rank factors $\mathbf{WH}$.

Also, according to BCD, the independent elements in a block can be computed simultaneously. Here, the computations of the elements $h_{xi}, h_{xj} \in \mathbf{h}_x^\intercal, i \neq j$, are independent of each other. Hence, the problem of finding row $\mathbf{h}_x^\intercal$ fixing column $\mathbf{w}_x$ is

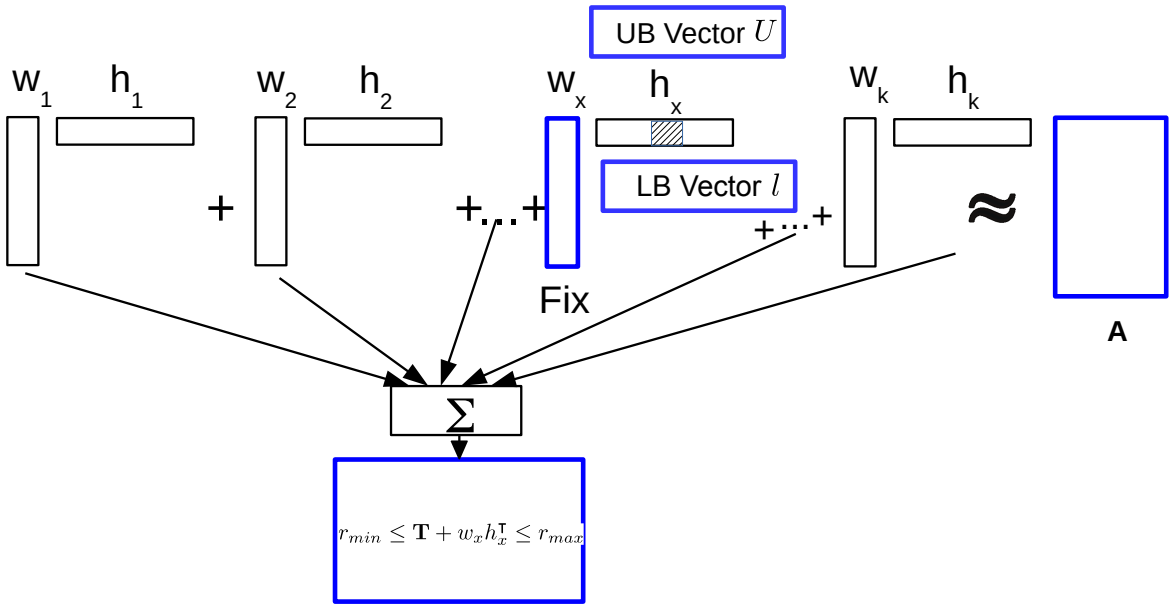


Figure 8: Bounded Matrix Low Rank Approximation Solution Overview

equivalent to solving the following problem

$$\begin{aligned} &\min_{h_{xi}} \|\mathbf{M}(:,i) \cdot *((\mathbf{A}-\mathbf{T})(:,i) - \mathbf{w}_x h_{xi})\|_F^2 \\ &\forall i = [1,m],\ \forall x = [1,k] \\ \text{subject to} \quad & \\ &\mathbf{T}(:,i) + \mathbf{w}_x h_{xi} \leqslant r_{max} \\ &\mathbf{T}(:,i) + \mathbf{w}_x h_{xi} \geqslant r_{min} \end{aligned} \tag{25}$$

To construct the row vector $\mathbf{h}_x^\intercal$, we use $k(n+m)$ scalar blocks based on problem formulation (25). Theorem 4 identifies these best elements that construct $\mathbf{h}_x^\intercal$. As shown in Figure 8, given the **bold** blocks, $\mathbf{T}$, $\mathbf{A}$ and $\mathbf{w}_x$, we find the row vector $\mathbf{h}_x^\intercal = [h_{x1}, h_{x2}, \cdots, h_{xm}]$ for Problem (25). For this, let us understand the boundary values of h_{xi} by defining two vectors, $\mathbf{l}$ bounding h_{xi} from below, and $\mathbf{u}$ bounding h_{xi} from above, i.e., $max(\mathbf{l}) \leqslant h_{xi} \leqslant min(\mathbf{u})$.

Definition The lower bound vector $\mathbf{l} = [l_1, \ldots, l_n] \in \mathbb{R}^n$ and the upper bound vector

$\mathbf{u} = [u_1, \ldots, u_n] \in \mathbb{R}^n$ for a given $\mathbf{w}_x$ and $\mathbf{T}$ that bound h_{xi} are defined $\forall j \in [1, n]$ as

$$l_j = \begin{cases} \dfrac{r_{min} - \mathbf{T}(j,i)}{w_{jx}}, & w_{jx} > 0 \\ \dfrac{r_{max} - \mathbf{T}(j,i)}{w_{jx}}, & w_{jx} < 0 \\ -\infty, & otherwise \end{cases}$$

and

$$u_j = \begin{cases} \dfrac{r_{max} - \mathbf{T}(j,i)}{w_{jx}}, & w_{jx} > 0 \\ \dfrac{r_{min} - \mathbf{T}(j,i)}{w_{jx}}, & w_{jx} < 0 \\ \infty, & otherwise. \end{cases}$$

It is important to observe that the defined $\mathbf{l}$ and $\mathbf{u}$ – referred as `LowerBounds` and `UpperBounds` in Algorithm 2, are for a given $\mathbf{w}_x$ and $\mathbf{T}$ to bound h_{xi}. Alternatively, if we are solving $\mathbf{w}_x$ for a given $\mathbf{T}$ and $\mathbf{h}_x$, the above function correspondingly represents the possible lower and upper bounds for w_{ix}, where $\mathbf{l}, \mathbf{u} \in \mathbb{R}^m$.

Theorem 3. *Given* $\mathbf{A}$, $\mathbf{T}$, $\mathbf{w}_x$, *the* h_{xi} *is always bounded as* $max(\mathbf{l}) \leqslant h_{xi} \leqslant min(\mathbf{u})$.

Proof. It is easy to see that if $h_{xi} < max(\mathbf{l})$ or $h_{xi} > min(\mathbf{u})$, then $\mathbf{T}(:,i) + \mathbf{w}_x h_{xi}^\intercal \notin [r_{min}, r_{max}]$. □

Here, it is imperative to note that if h_{xi}, results in $\mathbf{T}(:,i) + \mathbf{w}_x h_{xi}^\intercal \notin [r_{min}, r_{max}]$, this implies that h_{xi} is either less than the $max(\mathbf{l})$ or greater than the $min(\mathbf{u})$. It cannot be any other inequality.

Given the boundary values of h_{xi}, Theorem 4 defines the solution to Problem (25).

Theorem 4. *Given* $\mathbf{T}$, $\mathbf{A}, \mathbf{w}_x, \mathbf{l}$ *and* $\mathbf{u}$, *let*

$$\hat{h_{xi}} = ([\mathbf{M}(:,i) \cdot *(\mathbf{A} - \mathbf{T})(:,i)]^\intercal \mathbf{w}_x)/(\|\mathbf{M}(:,i) \cdot * \mathbf{w}_x\|_2^2).$$

The unique solution h_{xi} – referred as **`FindElement`** *in Algorithm 2 to least squares problem* (25) *is given as*

$$h_{xi} = \begin{cases} max(\mathbf{l}), & if\, \hat{h_{xi}} < max(\mathbf{l}) \\ min(\mathbf{u}), & if\, \hat{h_{xi}} > min(\mathbf{u}) \\ \hat{h_{xi}}, & otherwise. \end{cases}$$

Proof. Out of Boundary: $h_{xi} < max(\mathbf{l})$ or $h_{xi} > min(\mathbf{u})$. Under this circumstance, the best value for h_{xi} is either $max(\mathbf{l})$ or $min(\mathbf{u})$. We can prove this by contradiction. Let us assume there exists a $\tilde{h_{xi}} = max(\mathbf{l}) + \delta; \delta > 0$ that is optimal to the Problem (25) for $h_{xi} < max(\mathbf{l})$. However, for $h_{xi} = max(\mathbf{l}) < \tilde{h_{xi}}$ is still a feasible solution for the Problem (25). Also, there does not exist a feasible solution that is less than $max(\mathbf{l})$, because the Problem (25) is quadratic in h_{xi}. Hence for $h_{xi} < max(\mathbf{l})$, the optimal value for the Problem (25) is $max(\mathbf{l})$. In similar direction we can show that the optimal value of h_{xi} is $min(\mathbf{u})$ for $h_{xi} > min(\mathbf{u})$.

Within Boundary: $max(\mathbf{l}) \leqslant h_{xi} \leqslant min(\mathbf{u})$.

Let us consider the objective function of unconstrained optimization problem (25). That is., $f = \min_{h_{xi}} \|\mathbf{M}(:,i) \cdot *((\mathbf{A} - \mathbf{T})(:,i) - \mathbf{w}_x h_{xi})\|_2^2$. The minimum value is determined by taking the derivative of f with respect to h_{xi} and equating it to zero.

$$
\begin{aligned}
\frac{\partial f}{\partial h_{xi}} &= \frac{\partial}{\partial h_{xi}} \left(\sum_{\substack{\texttt{all known ratings} \\ \texttt{in column } i}} (\mathbf{E}_i - \mathbf{w}_x h_{xi})^2 \right) \quad (\texttt{where } \mathbf{E} = \mathbf{R} - \mathbf{T}) \\
&= \frac{\partial}{\partial h_{xi}} \left(\sum_{\substack{\texttt{all known ratings} \\ \texttt{in column } i}} (\mathbf{E}_i - \mathbf{w}_x h_{xi})^\intercal (\mathbf{E}_i - \mathbf{w}_x h_{xi}) \right) \\
&= \frac{\partial}{\partial h_{xi}} \left(\sum_{\substack{\texttt{all known ratings} \\ \texttt{in column } i}} (\mathbf{E}_i^\intercal - h_{xi}\mathbf{w}_x^\intercal)(\mathbf{E}_i - \mathbf{w}_x h_{xi}) \right) \\
&= \frac{\partial}{\partial h_{xi}} \left(\sum_{\substack{\texttt{all known ratings} \\ \texttt{in column } i}} h_{xi}^2 \mathbf{w}_x^\intercal \mathbf{w}_x - h_{xi}\mathbf{E}_i^\intercal \mathbf{w}_x - h_{xi}\mathbf{w}_x^\intercal \mathbf{E}_i + \mathbf{E}_i^\intercal \mathbf{E}_i \right) \\
&= 2\|\mathbf{M}(:,i). * \mathbf{w}_x\|_2^2 h_{xi} - 2[\mathbf{M}(:,i). * (\mathbf{A} - \mathbf{T})(:,i)]^\intercal \mathbf{w}_x
\end{aligned}
\tag{26}
$$

Now, equating $\frac{\partial f}{\partial h_{xi}}$ to zero will yield the optimum solution for the unconstrained optimization problem (25) as $h_{xi} = ([\mathbf{M}(:,i) \cdot *(\mathbf{A} - \mathbf{T})(:,i)]^\intercal \mathbf{w}_x)/(\|\mathbf{M}(:,i) \cdot * \mathbf{w}_x\|_2^2)$

□

In the similar direction the proof for Theorem 5 can also be established.

3.3.2 Bounding Existing ALS Algorithms (BALS)

Over the last few years, the recommender system algorithms have improved by leaps and bounds. The additional sophistication such as using only nearest neighbors during factorization, implicit ratings, time, etc., gave only a diminishing advantage for the Root Mean Square Error (RMSE) scores. That is, the improvement in RMSE score over the naive low rank approximation with implicit ratings is more than the improvement attained by utilizing both implicit ratings and time. Today, these algorithms are artificially truncating the estimated unknown ratings. However, it is important to investigate establishing bounds scientifically on these existing Alternating Least Squares (ALS) type algorithms.

Using matrix block BCD, we introduce a temporary variable $\mathbf{Z} \in \mathbb{R}^{n \times m}$ with box constraints to solve Problem (19),

$$\begin{aligned} &\min_{\mathbf{\Theta},\mathbf{Z},\mathbf{P},\mathbf{Q}} \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{Z})\|_F^2 + \alpha \|\mathbf{Z} - f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H})\|_F^2 \\ &\text{subject to} \\ &\qquad r_{min} \leqslant \mathbf{Z} \leqslant r_{max}. \end{aligned} \tag{27}$$

The key question is identifying optimal $\mathbf{Z}$. We assume the iterative algorithm has a specific update order, for example, $\boldsymbol{\theta}_1 \rightarrow \cdots \rightarrow \boldsymbol{\theta}_l \rightarrow \mathbf{W} \rightarrow \mathbf{H}$. Before updating these parameters, we should have an optimal $\mathbf{Z}$, with the most recent values of $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$.

Theorem 5. *The optimal* $\mathbf{Z}$*, given* $\mathbf{A}, \mathbf{W}, \mathbf{H}, \mathbf{M}, \mathbf{\Theta}$*, is* $\frac{\mathbf{M}.*(\mathbf{A}+\alpha f(\mathbf{\Theta},\mathbf{W},\mathbf{H}))}{1+\alpha} + \mathbf{M}'. * f(\mathbf{\Theta}), \mathbf{W}, \mathbf{H})$*, where* $\mathbf{M}'$ *is the complement of the indicator boolean matrix* $\mathbf{M}$*.*

Proof. Given $\mathbf{A}, \mathbf{W}, \mathbf{H}, \mathbf{M}, \mathbf{\Theta}$, the optimal $\mathbf{Z}$, is obtained by solving the following optimization problem.

$$\begin{aligned} &G = \min_{\mathbf{Z}} \|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{Z})\|_F^2 + \alpha \|\mathbf{Z} - f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H})\|_F^2 \\ &\text{subject to} \\ &\qquad r_{min} \leqslant \mathbf{Z} \leqslant r_{max}. \end{aligned} \tag{28}$$

Taking the gradient $\frac{\partial G}{\partial Z}$ of the above equation to find the optimal solution yields, $\frac{\mathbf{M}.*(\mathbf{A}+\alpha f(\mathbf{\Theta},\mathbf{W},\mathbf{H}))}{1+\alpha} + \mathbf{M}'. * f(\mathbf{\Theta}, \mathbf{W}, \mathbf{H})$. The derivation is in the same lines as explained in equations (26) □

In the same direction of Theorem 4, we can show that if the values of $\mathbf{Z}$ are outside $[r_{min}, r_{max}]$, that is., $\mathbf{Z} > r_{max}$ and $\mathbf{Z} < r_{min}$, the optimal value is $\mathbf{Z} = r_{max}$ and $\mathbf{Z} = r_{min}$ respectively.

In the next Section, the implementation of the algorithm for BMA and its variants

such as scalable and block implementations will be studied. Also, imposing bounds on existing ALS algorithms using $BALS$ will be investigated. As an example we will take existing algorithms from Graphchi [35] implementations and study imposing bounds using the $BALS$ framework.

3.4 Implementations

3.4.1 Bounded Matrix Low Rank Approximation

Given the discussion in the previous sections, we now have the necessary tools to construct the algorithm. In Algorithm 2, the $\mathbf{l}$ and $\mathbf{u}$ from Theorem 3 are referred to as `LowerBounds` and `UpperBounds`, respectively. Also, h_{xi} from Theorem 4 is referred to as `FindElement`. The BMA algorithm has three major functions: (1) Initialization, (2) Stopping Criteria and (3) Find the low rank factors $\mathbf{W}$ and $\mathbf{H}$. In later sections, the initialization and stopping criteria are explained in detail. For now, we assume that two initial matrices $\mathbf{W}$ and $\mathbf{H}$ are required, such that $\mathbf{WH} \in [r_{min}, r_{max}]$, and that a stopping criterion will be used for terminating the algorithm, when the constructed matrices $\mathbf{W}$ and $\mathbf{H}$ provide a good representation of the given matrix $\mathbf{A}$.

In the case of BMA algorithm, since multiple elements can be updated independently, we reorganize the scalar block BCD into $2k$ vector blocks. The BMA algorithm is presented as Algorithm 2.

Algorithm 2 works very well and yields low rank factors $\mathbf{W}$ and $\mathbf{H}$ for a given matrix $\mathbf{A}$ such that $\mathbf{WH} \in [r_{min}, r_{max}]$. However, when applied for very large scale matrices, such as recommender systems, it can only be run on machines with a large amount of memory. We address scaling the algorithm on multi core systems and machines with low memory in the next section.

input : Matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, $r_{min}, r_{max} > 1$, reduced rank k
output: Matrix $\mathbf{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{H} \in \mathbb{R}^{k \times m}$

// Rand initialization of W, H.
Initialize $\mathbf{W}$*,* $\mathbf{H}$ *as non-negative random matrices* ;
// modify random $\mathbf{WH}$ such that $\mathbf{WH} \in [r_{min}, r_{max}]$
// maxelement of $\mathbf{WH}$ without first column of $\mathbf{W}$ and first row of $\mathbf{H}$
$maxElement = max(\mathbf{W}(:, 2:end) * \mathbf{H}(2:end, :))$;
$\alpha \leftarrow \sqrt{\frac{r_{max} - 1}{maxElement}}$;
$\mathbf{W} \leftarrow \alpha \cdot \mathbf{W}$;
$\mathbf{H} \leftarrow \alpha \cdot \mathbf{H}$;
$\mathbf{W}(:, 1) \leftarrow 1$;
$\mathbf{H}(1, :) \leftarrow 1$;
$\mathbf{M} \leftarrow$ `ComputeRatedBinaryMatrix(`$\mathbf{A}$`)`;
while *stopping criteria not met* **do**
  for $x \leftarrow 1$ **to** k **do**
    $\mathbf{T} \leftarrow \sum_{j=1, j \neq x}^{k} \mathbf{w}_j \mathbf{h}_j^\intercal$;
    for $i \leftarrow 1$ **to** m **do**
      // Find vectors $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ as in Definition 3.3.1
      $\mathbf{l} \leftarrow$ `LowerBounds(`$r_{min}, r_{max}, \mathbf{T}, i, \mathbf{w}_x$`)`;
      $\mathbf{u} \leftarrow$ `UpperBounds(`$r_{min}, r_{max}, \mathbf{T}, i, \mathbf{w}_x$`)`;
      // Find vector $\mathbf{h}_x^\intercal$ fixing $\mathbf{w}_x$ as in Theorem 4
      $h_{xi} \leftarrow$ `FindElement(`$\mathbf{p_x}, \mathbf{M}, \mathbf{A}, \mathbf{T}, i, x$`)`;
      if $h_{xi} < max(\mathbf{l})$ **then**
        $h_{xi} \leftarrow max(\mathbf{l})$;
      else if $h_{xi} > min(\mathbf{u})$ **then**
        $h_{xi} \leftarrow min(\mathbf{u})$;
    for $i \leftarrow 1$ **to** n **do**
      // Find vectors $\mathbf{l}, \mathbf{u} \in \mathbb{R}^m$ as in Definition 3.3.1
      $\mathbf{l} \leftarrow$ `LowerBounds(`$r_{min}, r_{max}, \mathbf{T}, i, \mathbf{h}_x^\intercal$`)`;
      $\mathbf{u} \leftarrow$ `UpperBounds(`$r_{min}, r_{max}, \mathbf{T}, i, \mathbf{h}_x^\intercal$`)`;
      // Find vector $\mathbf{w}_x$ fixing $\mathbf{h}_x^\intercal$ as in Theorem 4
      $w_{ix} \leftarrow$ `FindElement(`$\mathbf{h}_x^\intercal, \mathbf{M}^\intercal, \mathbf{R}^\intercal, \mathbf{T}^\intercal, i, x$`)`;
      if $w_{ix} < max(\mathbf{l})$ **then**
        $w_{ix} \leftarrow max(\mathbf{l})$;
      else if $w_{ix} > min(\mathbf{u})$ **then**
        $w_{ix} \leftarrow min(\mathbf{u})$;

Algorithm 2: Bounded Matrix Low Rank Approximation (BMA)

3.4.2 Scaling up Bounded Matrix Low Rank Approximation

In this section, we address the issue of scaling the algorithm for large matrices with missing elements. Two important aspects of making the algorithm run for large matrices are running time and memory. We discuss the parallel implementation of the algorithm, which we refer to as *Parallel Bounded Matrix Low Rank Approximation*. Subsequently, we also discuss a method called *Block Bounded Matrix Low Rank Approximation*, which will outline the details of executing the algorithm for large matrices in low memory systems. Let us start this section by discussing *Parallel Bounded Matrix Low Rank Approximation*.

3.4.2.1 Parallel Bounded Matrix Low Rank Approximation

In the case of the BCD method, the solutions of the sub-problems that depend on each other have to be computed sequentially to make use of the most recent values. However, if solutions for some blocks are independent of each other, it is possible to compute them simultaneously. We can observe that, according to Theorem 4, all elements $h_{xi}, h_{xj} \in \mathbf{h}_x^\intercal, i \neq j$ are independent of each other. We are leveraging this characteristic to parallelize the **for** loops in Algorithm 2. Nowadays, virtually all commercial processors have multiple cores. Hence, we can parallelize finding the h_{xi}'s across multiple cores. Since it is trivial to change the **for** in step 12 and step 20 of Algorithm 2 to **parallel for** the details will be omitted.

It is obvious to see that the $\mathbf{T}$ at step 11 in Algorithm 2 requires the largest amount of memory. Also, the function *FindElement* in step 15 takes a sizable amount of memory. Hence, it is not possible to run the algorithm for large matrices on machines with low memory, e.g., with rows and columns on the scale of 100,000's. Thus, we propose the following algorithm to mitigate this limitation: Block BMA.

3.4.2.2 Block Bounded Matrix Low Rank Approximation

To facilitate understanding of this section, let us define β – a data structure in memory factor. That is, maintaining a floating scalar as a sparse matrix with one element or full matrix with one element takes different amounts of memory. This is because of the data structure that is used to represent the numbers in the memory. The amount of memory is also dependent on using single or double precision floating point precision. Typically, in Matlab, the data structure in memory factor β for full matrix is around 10. Similarly, in Java, the β factor for maintaining a number in an ArrayList is around 8. Let, $memsize(v)$ be the function that returns the approximate memory size of a variable v. Generally, $memsize(v)$ = number of elements in v * size of each element * β. Consider an example of maintaining 1000 floating point numbers on an ArrayList of a Java program. The approximate memory would be 1000*4*8 = 32000 bytes $\approx$ 32KB in contrast to the actual 4KB due to the factor β=8.

As discussed earlier, for most of the real world large datasets such as Netflix, Yahoo music, online dating, book crossing, etc., it is impossible to keep the entire matrix $\mathbf{T}$ in memory. Also, notice that, according to Theorem 4 and Definition 3.3.1, we need only the i-th column of $\mathbf{T}$ to compute h_{xi}. The block size of $h_{x\mathbf{i}}$ to compute in one core of the machine is dependent on the size of $\mathbf{T}$ and $FindElements$ that fits in memory.

On the one hand, partition $\mathbf{h}_x$ to fit the maximum possible $\mathbf{T}$ and $FindElements$ in the entire memory of the system. If very small partitions are created such that, we can give every core some amount of work so that the processing capacity of the system is not underutilized. The disadvantage of the former, is that only one core is used. However, in the latter case, there is a significant communication overhead. Figure 9 gives the pictorial view of the Block Bounded Matrix Low Rank Approximation.

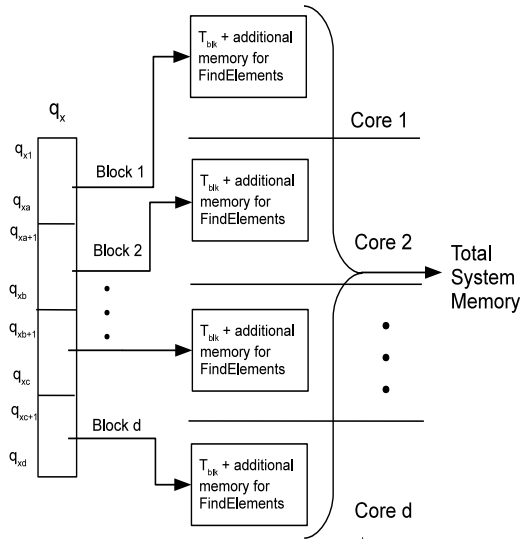


Figure 9: Block Bounded Matrix Low Rank Approximation

We determined the number of blocks = memsize(full($\mathbf{A}$)+other variables of FindElement)/(system memory * number of d cores). The full($\mathbf{A}$) is a non-sparse representation and $d \leqslant$ number of cores available in the system. Typically, for most of the datasets, we achieved minimum running time when we used 4 cores and 16 blocks. That is, we find 1/16-th of $\mathbf{h}_x^\intercal$ concurrently on 4 cores.

For convenience, we have presented the Block BMA as Algorithm 3. We describe only the algorithm to find the partial vector of $\mathbf{h}_x^\intercal$ given $\mathbf{w}_x$. To find more than one element, Algorithm 2 is modified such that the vectors $\mathbf{l}, \mathbf{u}, \mathbf{w}_x$ are matrices $\mathbf{L}, \mathbf{U}, \mathbf{W}_{blk}$, respectively, in Algorithm 3. Algorithm 3 replaces the steps 12 – 19 in Algorithm 2 for finding $\mathbf{h}$ and similarly for finding $\mathbf{w}$ from step 20 – 27. The initialization and the stopping criteria for Algorithm 3 are similar to those of Algorithm 2. Also included are the necessary steps to handle numerical errors as part of Algorithm 3 explained in Section 3.5. Figure 10 in Section 3.5, presents the speed up of the algorithm.

input : Matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, set of indices $\mathbf{i}$, current $\mathbf{w}_x$, x, current $\mathbf{h}'_x$, r_{min}, r_{max}

output: Partial vector $\mathbf{h}_x$ of requested indices $\mathbf{i}$

`// ratings of input indices` $\mathbf{i}$

$\mathbf{A}_{blk} \leftarrow \mathbf{A}(:, \mathbf{i})$;

$\mathbf{M}_{blk} \leftarrow$ `ComputeRatedBinaryMatrix(`$\mathbf{A}_{blk}$`)`;

$\mathbf{W}_{blk} \leftarrow$ `Replicate(`$\mathbf{w}_x, size(\mathbf{i})$`)`;

`// save` $\mathbf{h}_x(\mathbf{i})$

$\mathbf{h}'_{blk} \leftarrow \mathbf{h}_x(\mathbf{i})$;

`//` $\mathbf{T}_{blk} \in n \times size(\mathbf{i})$ `of input indices` $\mathbf{i}$

$$\mathbf{T}_{blk} \leftarrow \sum_{j=1, j \neq x}^{k} \mathbf{w}_j \mathbf{h}_{blk}^{\intercal};$$

`// Find matrix` $\mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times size(\mathbf{i})}$ `as in Definition 3.3.1`

$\mathbf{L} \leftarrow$ `LowerBounds(`$r_{min}, r_{max}, \mathbf{T}, \mathbf{i}, \mathbf{w}_x$`)`;

$\mathbf{U} \leftarrow$ `UpperBounds(`$r_{min}, r_{max}, \mathbf{T}, \mathbf{i}, \mathbf{w}_x$`)`;

`// Find vector` $\mathbf{h}_{blk}$ `fixing` $\mathbf{w}_x$ `as in Theorem 4`

$\mathbf{h}_{blk} = ([\mathbf{M}_{blk} \cdot *(\mathbf{A}_{blk} - \mathbf{T}_{blk})]^{\intercal} \mathbf{W}_{blk}) / (\|\mathbf{M}_{blk} \cdot * \mathbf{W}_{blk}\|_F^2)$;

`// Find indices of` $\mathbf{h}_{blk}$ `that are not within bounds`

$\mathbf{idxlb} \leftarrow find(\mathbf{h}_{blk} < max(\mathbf{L}))$;

$\mathbf{idxub} \leftarrow find(\mathbf{h}_{blk} > min(\mathbf{U}))$;

`// case A & B numerical errors in Section 3.5`

$idxcase1 \leftarrow find([\mathbf{h}'_{blk} \approx max(\mathbf{L})] or [\mathbf{h}'_{blk} \approx min(\mathbf{U}])$;

$idxcase2 \leftarrow find([max(\mathbf{L}) \approx min(\mathbf{U})] or [max(\mathbf{L}) > min(\mathbf{U})])$;

$idxdontchange \leftarrow idxcase1 \cup idxcase2$;

`// set appropriate values of` $\mathbf{h}_{blk} \notin [max(\mathbf{L}), min(\mathbf{U})]$

$\mathbf{h}_{blk}(\mathbf{idxlb} \setminus idxdontchange) \leftarrow max(\mathbf{L})(idxlb \setminus idxdontchange)$;

$\mathbf{h}_{blk}(\mathbf{idxub} \setminus idxdontchange) \leftarrow min(\mathbf{U})(idxub \setminus idxdontchange)$;

$\mathbf{h}_{blk}(idxdontchange) \leftarrow \mathbf{h}'_{blk}(idxdontchange)$;

Algorithm 3: Block BMA

3.4.3 Bounding Existing ALS Algorithms(BALS)

In this section we examine the algorithm for solving the Equation (27) based on Theorem 5 to find the low rank factors $\mathbf{W}$, $\mathbf{H}$ and $\mathbf{\Theta}$. For the time being, assume that we need an initial, $\mathbf{\Theta}, \mathbf{W}$ and $\mathbf{H}$ to start the algorithm. Also, we need update functions for $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$ and a stopping criteria for terminating the algorithm. The stopping criteria determines whether the constructed matrices $\mathbf{W}$ and $\mathbf{H}$ and $\mathbf{\Theta}$ provide a good representation of the given matrix $\mathbf{A}$.

input : Matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, $r_{min}, r_{max} > 1$, reduced rank k
output: Parameters $\mathbf{\Theta}$, Matrix $\mathbf{W} \in \mathbb{R}^{n \times k}$ and $\mathbf{H} \in \mathbb{R}^{k \times m}$

`// Rand initialization of Θ, W, H.`
Initialize $\mathbf{\Theta}, \mathbf{W}$, $\mathbf{H}$ *as a random matrix* ;
$\mathbf{M} \leftarrow$ `ComputeRatedBinaryMatrix(A)`;
`// Compute Z as in Theorem 5`
$\mathbf{Z} \leftarrow$ `ComputeZ(`$\mathbf{A}, \mathbf{W}, \mathbf{H}, \mathbf{\Theta}$`)`;
while *stopping criteria not met* **do**
  $\boldsymbol{\theta_i} \leftarrow \underset{\boldsymbol{\theta}_i}{\operatorname{argmin}} \|\mathbf{M} \cdot *(\mathbf{A} - f(\boldsymbol{\theta}_i, \mathbf{W}, \mathbf{H}))\|_F^2 \ \ \forall 1 \leqslant i \leqslant l$;
  $\mathbf{W} \leftarrow \underset{\mathbf{W}}{\operatorname{argmin}} \|\mathbf{M} \cdot *(\mathbf{A} - f(\boldsymbol{\theta}_i, \mathbf{W}, \mathbf{H}))\|_F^2$;
  $\mathbf{H} \leftarrow \underset{\mathbf{H}}{\operatorname{argmin}} \|\mathbf{M} \cdot *(\mathbf{A} - f(\boldsymbol{\theta}_i, \mathbf{W}, \mathbf{H}))\|_F^2$;
  $\mathbf{Z} \leftarrow$ `ComputeZ(`$\mathbf{A}, \mathbf{W}, \mathbf{H}, \mathbf{\Theta}$`)`;
  if $z_{ij} > r_{max}$ **then**
    $z_{ij} = r_{max}$
  if $z_{ij} < r_{min}$ **then**
    $z_{ij} = r_{min}$

Algorithm 4: Bounding Existing ALS Algorithm (BALS)

In the *ComputeZ* function, if the values of $\mathbf{Z}$ are outside $[r_{min}, r_{max}]$, that is., $\mathbf{Z} > r_{max}$ and $\mathbf{Z} < r_{min}$, set the corresponding values of $\mathbf{Z} = r_{max}$ and $\mathbf{Z} = r_{min}$ respectively.

Most of the ALS based recommender system algorithms have clear defined blocks on $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$ as discussed in Section 2.1. That is, either they are partitioned as

a matrix, vector or element blocks. Also, there is always an update order that is adhered to. For example, $\boldsymbol{\theta}_1 \rightarrow \boldsymbol{\theta}_2 \rightarrow \cdots \boldsymbol{\theta}_l \rightarrow \mathbf{w}_1 \rightarrow \mathbf{w}_2 \rightarrow \cdots \rightarrow \mathbf{w}_k \rightarrow \mathbf{h}_1 \rightarrow \mathbf{h}_2 \rightarrow \cdots \rightarrow \mathbf{h}_k$. If the algorithm meets these characteristics, we can prove that the algorithm converges to a stationary point.

Corollary 6. *If the recommender system algorithm f based on alternating least squares, satisfies the following characteristics: (1) is an iterative coordinate descent algorithm, (2) defines blocks over the optimization variables $\boldsymbol{\Theta}, \mathbf{W}, \mathbf{H}$, and (3) has orderly optimal block updates of one block at a time and always uses the latest blocks, f converges to a stationary point of (27).*

Proof. This is based on Theorem 1. The BALS Algorithm 4 for the formulation (27) satisfies the above characteristics and hence the algorithm converges to a stationary point. □

At this juncture, it is important to discuss the applicability of BMA for gradient descent type of algorithms such as SVD++, Bias-SVD, time-SVD++, and others. For brevity, we consider the SGD for the simple matrix factorization problem explained in equation (19). For a gradient descent type of algorithm, we update the current value of $w_{uk} \in \mathbf{W}$ and $h_{ki} \in \mathbf{H}$ based on the gradient of the error $e_{ui} = a_{ui} - f(\mathbf{W}', \mathbf{H}')$. Assuming $\lambda_w = \lambda_h = \lambda$, the update equations are $w_{uk} = w'_{uk} + (e_{ui}h'_{ki} - \lambda w'_{uk})$ and $h_{ki} = h'_{ki} + (e_{ui} * w'_{uk} - \lambda h'_{ki})$. In the case of BALS for unobserved pairs (u, i), we use $z_{ui} = f(\mathbf{W}', \mathbf{H}')$ instead of $a_{ui} = 0$. Thus, in the case of the BALS extended gradient descent algorithms, the error $e_{ui} = 0$, for unobserved pairs (u, i). That is, the updates of w_{uk} and h_{ki} are dependent only on the observed entries and our estimations for unrated pairs (u, i) do not have any impact.

It is also imperative to understand that we are considering only the existing recommender system algorithms that minimizes the RMSE error of the observed entries against its estimation as specified in (20). We have not analyzed the problems that

utilize different loss functions such as KL-Divergence.

In this section, we also present an example for bounding an existing algorithm, Alternating Least Squares, with Regularization in Graphchi. GraphChi [35] is an open source library that allows distributed processing of very large graph computations on commodity hardware. It breaks large graphs into smaller chunks and uses a parallel sliding windows method to execute large data mining and machine learning programs in small computers. As part of the library samples, it provides implementations of many recommender system algorithms. In this section we discuss extending the existing ALSWR implementation in Graphchi to impose box constraints using our framework.

Graphchi programs are written in the vertex-centric model and runs vertex-centric programs asynchronously (i.e., changes written to edges are immediately visible to subsequent computation), and in parallel. Any Graphchi program has three important functions: (1) $beforeIteration$, (2) $afterIteration$, and (3) *update* function. The function $beforeIteration$ and $afterIteration$ are executed sequentially in a single core, whereas the *update* function for all the vertices is executed in parallel across multiple cores of the machine. Such parallel updates are useful for updating independent blocks. For example, in our case, every vector $\mathbf{w}_i^\intercal$ is independent of $\mathbf{w}_j^\intercal$, for $i \neq j$.

Graphchi models the collaborative filtering problem as a bipartite graph between a user vertex and item vertex. The edges that flow between the user-item partition are ratings. That is, a weighted edge, between a user u and an item i vertex represent the user u's rating for the item i. The user vertex has only outbound edges and an item vertex has only inbound edges. The *update* function is called for all the user and item vertices. The vertex *update* solves a regularized least-squares system, with neighbors' latent factors as input.

One of the major challenges for existing algorithms to enforce bounds using the

BALS framework is memory. The matrix $\mathbf{Z}$ is dense and may not be accommodative in memory. That is, consider the $\mathbf{Z}$ for the book crossing dataset[1]. The dataset provides ratings of 278,858 users against 271,379 books. The size of $\mathbf{Z}$ for such a dataset would be $number of users * number of items * size(double)$ = 278858*271379*8 bytes $\approx$ 563GB. This data size is too large even for a server system. To overcome this problem, we save the $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$ of previous iterations as $\mathbf{\Theta'}, \mathbf{W'}, \mathbf{H'}$. Instead of having the the entire $\mathbf{Z}$ matrix in memory, we compute the z_{ui} during the update function.

In the interest of space, we present the pseudo code for the *update* function alone. In the $beforeIteration$, function, we backup the existing variables $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$ as $\mathbf{\Theta'}, \mathbf{W'}, \mathbf{H'}$. The $afterIteration$ function computes the RMSE of the validation/training set and determines the stopping criteria.

input : Vertex v user/item, GraphContext ctx, $\mathbf{H}, \mathbf{W'}, \mathbf{H'}, \mathbf{\Theta'}$
output: The u^{th} row of $\mathbf{W}$ matrix $\mathbf{w}_u^\intercal \in \mathbb{R}^k$ or the i^{th} column $\mathbf{h}_i \in \mathbb{R}^k$
// u^{th} row of matrix Z based on Theorem 5. $\mathbf{\Theta'}, \mathbf{W'}, \mathbf{H'}$ are the $\mathbf{\Theta}, \mathbf{W}, \mathbf{H}$ from previous iteration.
// Whether the vertex is a user/item vertex is determined by the number of incoming/outgoing edges. For user vertex the number of incoming edges = 0 and for item vertex the number of outgoing edges = 0
1 **if** *vertex v is user u vertex* **then**
    // update $\mathbf{w}_u^\intercal$
2   $\mathbf{z}_u^\intercal \in \mathbb{R}^m \leftarrow f(\mathbf{W'}, \mathbf{H'})$;
    // We are replacing the $\mathbf{a}_u$ in the original algorithm with $\mathbf{z}_u$
3   $\mathbf{w}_u^\intercal \leftarrow (\mathbf{H}\mathbf{H}^\intercal) \setminus (\mathbf{z}_u^\intercal * \mathbf{H}^\intercal)^\intercal$;
4 **else**
    // update $\mathbf{h}_i$
5   $\mathbf{z}_i \in \mathbb{R}^n \leftarrow f(\mathbf{W'}, \mathbf{H'})$;
    // We are replacing the $\mathbf{a}_i$ in the original algorithm with $\mathbf{z}_i$
6   $\mathbf{h}_i \leftarrow (\mathbf{W}^\intercal\mathbf{W}) \setminus (\mathbf{W}^\intercal * \mathbf{z}_i)$;

Algorithm 5: update function

[1]The details about this dataset can be found in Table 3

Considering Algorithm 5, it is important to observe that we use the previous iteration's $\mathbf{W}', \mathbf{H}', \mathbf{\Theta}'$ only for the computation of $\mathbf{Z}$. However, for $\mathbf{W}, \mathbf{H}$ updates, the current latest blocks are used. Also, we cannot store the matrix $\mathbf{M}$ in memory. We know that Graphchi, as part of the $Vertex$ information, passes the set of incoming and outgoing edges to and from the vertex. The set of outgoing edges from the user vertex u to the item vertex v, provides information regarding the items rated by the user u. Thus, we use this information rather than maintaining $\mathbf{M}$ in memory. The performance comparison between ALSWR and ALSWR-Bounded on the Netflix dataset [1] is presented in Table 6. Similarly, we also bounded Probabilistic Matrix Factorization (PMF) in the Graphchi library and compared the performances of bounded ALSWR and bounded PMF algorithms using the BALS framework with its artificially truncated version on various real world datasets (see Table 7).

3.4.4 Parameter Tuning

In the case of recommender systems the missing ratings are provided as ground truth in the form of test data. The dot product of $\mathbf{W}(u,:)$ and $\mathbf{H}(:,i)$ gives the missing rating of a (u,i) pair. In such cases, the accuracy of the algorithm is determined by the Root Mean Square Error (RMSE) of the predicted ratings against the ground truth. It is unimportant how good the algorithm converges for a given rank k.

This section discusses ways to improve the RMSE of the predictions against the missing ratings by tuning the parameters of the BMA algorithm and BALS framework.

3.4.4.1 Initialization

The BMA algorithm can converge to different points depending on the initialization. In Algorithm 2, it was shown how to use random initialization so that $\mathbf{WH} \in [r_{min}, r_{max}]$. In general, this method should provide good results.

However, in the case of recommender systems, this initialization can be tuned,

which can give even better results. According to Koren [32], one good baseline estimate for a missing rating (u, i) is $\mu + p_u + q_i$, where μ is the average of the known ratings, and p_u and q_i are the bias of user u and item i, respectively. We initialized $\mathbf{W}$ and $\mathbf{H}$ in the following way

$$\mathbf{W} = \begin{pmatrix} \frac{\mu}{k-2} & \cdots & \frac{\mu}{k-2} & p_1 & 1 \\ \vdots & & \vdots & \vdots & \vdots \\ \frac{\mu}{k-2} & \cdots & \frac{\mu}{k-2} & p_n & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{H} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 1 \\ q_1 & q_2 & \cdots & q_m \end{pmatrix},$$

such that $\mathbf{WH}(u, i) = \mu + p_u + q_i$. That is, let the first $k-2$ columns of $\mathbf{W}$ be $\frac{\mu}{k-2}$, $\mathbf{W}(:, k-1) = \mathbf{p}$ and $\mathbf{W}(:, k) = 1$. Let all the $k-1$ rows of $\mathbf{H}$ be 1's and $\mathbf{H}(k, :) = \mathbf{q}^\intercal$. We call this a baseline initialization.

3.4.4.2 Reduced Rank k

In the case of regular low rank approximation with all known elements, the higher the k, the closer the low rank approximation is to the input matrix [27]. However, in the case of predicting with the low rank factors, a good k depends on the nature of the dataset. Even though, for a higher k, the low rank approximation is closer to the known rating of the input $\mathbf{A}$, the RMSE on the test data may be poor. In Table 4, we can observe the behavior of the RMSE on the test data against k. In most cases, a good k is determined by trial and error for the prediction problem.

3.4.4.3 Stopping Criterion $\mathfrak{C}$

The stopping criterion defines the goodness of the low rank approximation for the given matrix and the task for which the low rank factors are used. The two common stopping criteria are – (1) For a given rank k, the product of the low rank factors $\mathbf{WH}$ should be close to the known ratings of the matrix and (2) The low rank factors

$\mathbf{W}, \mathbf{H}$ should perform the prediction task on a smaller validation set which has the same distribution as the test set. The former is common when all the elements of $\mathbf{A}$ are known. We discuss only the latter, which is important for recommender systems.

The stopping criterion $\mathfrak{C}$ for the recommender system is the increase of $\sqrt{\frac{\|\mathbf{M} \cdot *(\mathbf{V} - \mathbf{WH})\|_F^2}{numRatings\ in\ \mathbf{V}}}$, for some validation matrix $\mathbf{V}$, which has the same distribution as the test matrix between successive iterations. Here, $\mathbf{M}$ is for the validation matrix $\mathbf{V}$. This stopping criterion has diminishing effect as the number of iterations increases. Hence, we also check whether $\sqrt{\frac{\|\mathbf{M} \cdot *(\mathbf{V} - \mathbf{WH})\|_F^2}{numRatings\ in\ \mathbf{V}}}$ did not change in successive iterations at a given floating point precision, e.g., 1e-5.

It is trivial to show that, for the above stopping criterion $\mathfrak{C}$, Algorithm 2 terminates for any input matrix $\mathbf{A}$. At the end of an iteration, we terminate if the RMSE on the validation set has either increased or marginally decreased.

3.5 Experimentation

Experimentation was conducted in various systems with memory as lows as 16GB. One of the major challenges during experimentation is numerical errors. The numerical errors could result in $\mathbf{T} + \mathbf{w}_x \mathbf{h}_x^\intercal \notin [r_{min}, r_{max}]$. The two fundamental questions to solve the numerical errors are: (1) How to identify the occurrence of a numerical error? and (2) What is the best possible value to choose in the case of a numerical error?

We shall start by addressing the former question of potential numerical errors that arise in the BMA Algorithm 2. It is important to understand that if we are well within bounds, i.e., if $max(\mathbf{l}) < q_{xi} < min(\mathbf{u})$, we are not essentially impacted by the numerical errors. It is critical only when q_{xi} is out of the bounds, that is, $q_{xi} < max(\mathbf{l})$ or $q_{xi} > min(\mathbf{u})$ and approximately closer to the boundary discussed as in (*Case A* and *Case B*). For discussion let us assume we are improving the old value of h'_{xi} to h_{xi} such that we minimize the error $\|\mathbf{M} \cdot *(\mathbf{A} - \mathbf{T} - \mathbf{w}_x \mathbf{h}_x^\intercal)\|_F^2$.

Table 3: Datasets for experimentation

Dataset	**Rows**	**Columns**	**Ratings** (millions)	**Density**	**Ratings Range**
Jester	73421	100	4.1	0.5584	[-10,10]
Movielens	71567	10681	10	0.0131	[1,5]
Dating	135359	168791	17.3	0.0007	[1,10]
Book crossing	278858	271379	1.1	0.00001	[1,10]
Netflix	17770	480189	100.4	0.01	[1,5]

Case A: $h'_{xi} \approx max(\mathbf{l})$ or $h'_{xi} \approx min(\mathbf{u})$:

This is equivalent to saying h'_{xi} is already optimal for the given $\mathbf{w}_x$ and $\mathbf{T}$ and there is no further improvement possible. Under this scenario, if $h'_{xi} \approx h_{xi}$ it is better to retain h'_{xi} irrespective of the new h_{xi} found.

Case B: $max(\mathbf{l}) \approx min(\mathbf{u})$ or $max(\mathbf{l}) > min(\mathbf{u})$:

According to Theorem 3, we know that $max(\mathbf{l}) < min(\mathbf{u})$. Hence, if $max(\mathbf{l}) > min(\mathbf{u})$, it is only the result of numerical errors.

In all the above cases during numerical errors, we are better off retaining the old value h'_{xi} against the new value h_{xi}. This covers Algorithm 3 – Block BMA for consideration of numerical errors.

We experimented with this Algorithm 3 among varied bounds using very large matrix sizes taken from the real world datasets. The datasets used for our experiments included the Movielens 10 million [1], Jester [18], Book crossing [60] and Online dating dataset [5]. The characteristics of the datasets are presented in Table 3.

We have chosen Root Mean Square Error (RMSE) – a defacto metric for recommender systems. The RMSE is compared for BMA with baseline initialization (BMA –Baseline) and BMA with random initialization (BMA –Random) against the other algorithms on all the datasets. The algorithms used for comparison are ALSWR (alternating least squares with regularization) [59], SGD [16], SVD++ [32] and Bias-SVD [32] and its implementation in Graphlab (http://graphlab.org/) [40] software

Table 4: RMSE Comparison of Algorithms on Real World Datasets

Dataset	k	**BMA Baseline**	**BMA Random**	**ALSWR**	**SVD++**	**SGD**	**Bias-SVD**
Jester	10	4.3320	4.6289	5.6423	5.5371	5.7170	5.8261
Jester	20	4.3664	4.7339	5.6579	5.5466	5.6752	5.7862
Jester	50	4.5046	4.7180	5.6713	5.5437	5.6689	5.7956
Movielens10M	10	0.8531	0.8974	1.5166	1.4248	1.2386	1.2329
Movielens10M	20	0.8526	0.8931	1.5158	1.4196	1.2371	1.2317
Movielens10M	50	0.8553	0.8932	1.5162	1.4204	1.2381	1.2324
Dating	10	1.9309	2.1625	3.8581	4.1902	3.9082	3.9052
Dating	20	1.9337	2.1617	3.8643	4.1868	3.9144	3.9115
Dating	50	1.9434	2.1642	3.8606	4.1764	3.9123	3.9096
Book Crossing	10	1.9355	2.8137	4.7131	4.7315	5.1772	3.9466
Book Crossing	20	1.9315	2.4652	4.7212	4.6762	5.1719	3.9645
Book Crossing	50	1.9405	2.1269	4.7168	4.6918	5.1785	3.9492

package. We implemented our algorithm in Matlab and used the parallel computing toolbox for parallelizing across multiple cores.

For parameter tuning, we varied the number of reduced rank k and tried different initial matrices for our algorithm to compare against all other algorithms mentioned above. For every k, every dataset was randomly partitioned into 85% training, 5% validation and 10% test data. We ran all algorithms on these partitions and computed their RMSE scores. We repeated each experiment 5 times and reported their RMSE scores in Table 4, where each resulting value is the average of the RMSE scores on a randomly chosen test set for 5 runs. Table 4 summarizes the RMSE comparison of all the algorithms.

The Algorithm 2 consistently outperformed existing state-of-the-art algorithms. One of the main reason for the consistent performance is the absence of hyper parameters. In the case of machine learning algorithms, there are many parameters that need to be tuned for performance. Even though the algorithms perform the best when provided with the right parameters, identifying these parameters is a formidable challenge, usually by trial and error methods. For example, in Table 4, we can observe

that the Bias-SVD, an algorithm without hyper parameters, performed better than its extension SVD++ with default parameters in many cases. The BMA algorithm without hyper parameters performed well on real world datasets, albeit a BMA with hyper parameters and the right parametric values would have performed even better.

Recently, there has been a surge in interest to understand the temporal impact on the ratings. Time-svd++ [33] is one such algorithm that leverages the time of rating to improve prediction accuracy. Also, the most celebrated dataset in the recommender system community is the Netflix dataset, since the prize money is attractive and it represents the first massive dataset for recommender systems that was publicly made available. The Netflix dataset consists of 17,770 users who rated 480,189 movies in a scale of [1 ... 5]. There was a total of 100,480,507 ratings in the training set and 1,408,342 ratings in the validation set. All the algorithms listed above were invented to address the Netflix challenge. Even though the book crossing dataset [60] is bigger than the Netflix, we felt our study is not complete without experimenting on Netflix and comparing against time-SVD++. However, the major challenge is that the Netflix dataset has been withdrawn from the internet and its test data is no longer available. Hence, we extracted a small sample of 5% from the training data as a validation set and tested the algorithm against the validation set that was supplied as part of the training package. We performed this experiment and the results are presented in Table 5. For a better comparison, we also present the original Netflix test scores for SVD++ and time-SVD++ algorithms from [33]. These are labeled *SVD++-Test* and *time-SVD++-Test*, respectively. Our BMA algorithm outperformed all the algorithms on the Netflix dataset when tested on the validation set supplied as part of the Netflix training package.

Additionally, we conducted an experiment to study the speed-up of the algorithm on the Netflix dataset. This is a simple speed-up experiment conducted with Matlab's Parallel Computing Toolbox on a dual socket Intel E7 system with 6 cores on each

Table 5: RMSE Comparison of BMA with other algorithms on Netflix

Algorithm	$k = 10$	$k = 20$	$k = 50$	$k = 100$
BMA Baseline	0.9521	0.9533	0.9405	0.9287
BMA Random	0.9883	0.9569	0.9405	0.8777
ALSWR	1.5663	1.5663	1.5664	1.5663
SVD++	1.6319	1.5453	1.5235	1.5135
SGD	1.2997	1.2997	1.2997	1.2997
Bias-SVD	1.3920	1.3882	1.3662	1.3354
time-svd++	1.1800	1.1829	1.1884	1.1868
SVD++-Test	0.9131	0.9032	0.8952	0.8924
time-SVD++-Test	0.8971	0.8891	0.8824	0.8805

socket. We collected the running time of the Algorithm 3 to compute the low rank factors $\mathbf{W}$ and $\mathbf{H}$ with $k = 50$, using 1, 2, 4, 8, and 12 parallel processes. Matlab's Parallel Computing Toolbox permits starting at most 12 Matlab workers for a local cluster. Hence, we conducted the experiment up to a pool size of 12. Figure 10 shows the speed-up of Algorithm 3. We observe from the graph that, up to pool size 8, the running time decreases with increasing pool size. However, the overhead costs such as communication and startup costs for running 12 parallel tasks surpasses the advantages of parallel execution. This simple speed-up experiment shows promising reductions in running time of the algorithm. A sophisticated implementation of the algorithm with low level parallel programming interfaces such as MPI, will result in better speed-ups.

In this section, we also present the results of bounding existing ALS type algorithms as explained in Section 3.3.2 and 3.4.3. The performance comparison between ALSWR and ALSWR-Bounded on the Netflix dataset is presented in Table 6. Similarly, we also bounded Probabilistic Matrix Factorization (PMF) in Graphchi library. We then compared the performances of both ALSWR and PMF algorithms on various real world datasets, which are presented in Table 7.

In this chapter, we presented a new matrix factorization for recommender systems

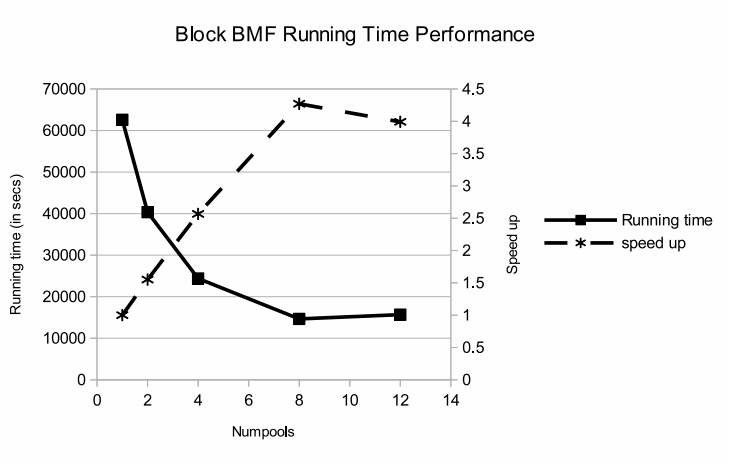


Figure 10: Speed up experimentation for Block BMA Algorithm 3

Table 6: RMSE Comparison of ALSWR on Netflix

Algorithm	$k = 10$	$k = 20$	$k = 50$
ALSWR	0.8078	0.755	0.6322
ALSWR-Bounded	0.8035	0.7369	0.6156

Table 7: RMSE Comparison using BALS framework on Real World Datasets

Dataset	k	**ALSWR Bounded**	**PMF Bounded**	**ALSWR**	**PMF**
Jester	10	4.4406	4.2011	4.4875	4.2949
Jester	20	4.8856	4.3018	5.0288	4.4608
Jester	50	5.6177	4.6893	6.1906	4.7383
ML-10M	10	0.8869	0.8611	0.9048	0.8632
ML-10M	20	0.9324	0.8752	0.9759	0.8891
ML-10M	50	1.0049	0.8856	1.1216	0.9052
Dating	10	2.321	1.9503	2.3206	1.9556
Dating	20	2.3493	1.9652	2.4458	1.9788
Dating	50	2.7396	2.0647	2.7406	2.0752
Book Crossing	10	4.6937	5.4676	4.7805	5.4901
Book Crossing	20	4.7977	5.3977	4.8889	5.4862
Book Crossing	50	5.0102	5.2281	5.0018	5.4707

called Bounded Matrix Low Rank Approximation (BMA), which imposes a lower and an upper bound on every estimated missing element of the input matrix. Also, we presented substantial experimental results on real world datasets illustrating that our proposed method outperformed the state-of-the-art algorithms for recommender system.

In future work we plan to extend BMA to tensors, i.e., multi-way arrays. Also, similar to time-SVD++, we will use time, neighborhood information, and implicit ratings during the factorization. A major challenge of BMA algorithm is that it loses sparsity during the product of low rank factors **WH**. This limits the applicability of BMA to other datasets such as text corpora and graphs where sparsity is important. Thus, we plan to extend BMA for sparse bounded input matrices as well. During our experimentation, we observed linear scale-up for Algorithm 3 in Matlab. However, the other algorithms from Graphlab are implemented in C/C++ and take less clock time. A C/C++ implementation of Algorithm 3 would an important step in order to compare the running time performance against the other state-of-the-art algorithms. Also, we will experiment with BMA on other types of datasets that go beyond those designed for recommender systems.

CHAPTER IV

COMMUNICATION AVOIDING NMF

Distributed NMF for very large matrices is an important problem in the community. In this chapter, we propose a distributed MPI-based Communication Avoiding NMF(CANMF) algorithm inspired by the parallel Jacobi method and Block Principal Pivoting (BPP). We are the first to propose Communication Avoiding NMF that can handle both large sparse and dense matrices by avoiding the communication between machines. Considering the sensitivity of the data in distributed environment, CANMF also ensures the input data from one machine will not be communicated to other machines in the cluster. In CANMF, we carefully chose to communicate only a portion of one of the low rank factors to the adjacent machine for every iteration. This reduces the communication cost from $O((m+n)klogP)$ to $O(\frac{nk}{P})$ for P parallel processes. We compare the performance of our algorithm with commonly used NMF algorithms such as multiplicative update, HALS on sparse-dense synthetic and real world datasets. We also present the scalability results of our CANMF algorithm.

4.1 Distributed Non-negative Matrix Factorization

Non-negative Matrix Factorization (NMF) is the problem of finding two low rank factors $\mathbf{W} \in \mathbb{R}_+^{m\times k}$ and $\mathbf{H} \in \mathbb{R}_+^{n\times k}$ for a given input matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, such that $\mathbf{A} \approx \mathbf{W}\mathbf{H}^T$, where $k << min(m,n)$–typically in the order of 50's. Formally, NMF problem [50] can be defined as

$$\min_{\mathbf{W}\geqslant 0,\mathbf{H}\geqslant 0} \quad f(\mathbf{W},\mathbf{H}) \equiv \|\mathbf{A}-\mathbf{W}\mathbf{H}^T\|_F^2 \tag{29}$$

There is a vast literature on algorithms for NMF and their convergence properties

[28]. The commonly adopted NMF algorithms are – (i) the easy to implement Multiplicative Update (MU) [50] (ii) Hierarchical Alternative Least Squares algorithm (HALS)[10] (iii) Sophisticated fast converging block principal pivot (BPP) [30] (iv) Stochastic Gradient Descent (SGD) Updates [17]. As described in Equation 40, most of the algorithms in NMF literature are based on Alternating Non-negative Least Squares (ANLS) scheme that iteratively optimizes each of the low rank factors $\mathbf{W}$ and $\mathbf{H}$ while keeping the other fixed. It is important to note that in such iterative alternating minimization technique, each subproblem is a constrained convex optimization problem. Each of this sub problems is then solved using standard optimization techniques such as projected gradient, interior point etc., and detailed survey for solving this constrained convex optimization problem can be found in [52][28]. In this chapter, for solving the sub problem, we use a fast active-set based method called Block Principal Pivoting(BPP) [30].

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. In this direction, it is important to study low rank approximation methods in a data-distributed environment and specifically in the current state of *high performance computing* (HPC). For example, in many large scale scenarios, data samples are collected and stored over many general purpose computers. Local computation is preferred as local access of data is generally faster than remote access due to network latency and network bandwidth. It may appear that it is being achieved using Hadoop, where computations happen with the locally available data. However, the NMF algorithm on Hadoop have the tendency to perform a very costly input data shuffle between the machines by assigning the keys to the dataset. For example., the key could be row number for each row of the input matrix. Sometimes, this is very expensive for dense input matrices.

In another situation where privacy is an issue, organizations may be reluctant to

share their own data while they still want to build good models using other organizations' data. For example, blood samples, images, finger prints, health diagnosis, and DNA samples are considered sensitive and private. In such realistic scenarios, communicating the original data with a central resource or between resources is discouraged. This work aims to propose an NMF algorithm that could work with data residing distributively over a connected network (e.g. a cluster). In the current state of the art Hadoop based distributed NMF algorithm, the algorithm is insensitive to the distribution of input matrix based on keys and there is a high chance that data may be placed in an unintended machine. As the name suggests, the proposed Communication Avoiding NMF(CANMF) algorithm does not communicate the original data $\mathbf{A}$, the left low rank factor $\mathbf{W}$ among machines in the network and communicates only a portion of one of the right low rank factor $\mathbf{H}$ to the adjacent machine.

As datasets are getting bigger and bigger, data mining algorithms also need to take into account the ability to handle large scale datasets, and the computations are often carried out in a distributed manner. Most of the NMF algorithms involve multiplication or decomposition of matrices which can be very expensive for large and distributed matrices. This is especially true in HPC settings where one has to not only compute but also communicate these matrices over the computing nodes. The idea proposed in this work is that one could use independent blocks of the data matrix $\mathbf{A}$ to update separated blocks of $\mathbf{W}$ and $\mathbf{H}$ in parallel using Block Principal Pivot (BPP) method.

According to the ANLS Framework, first partition the variables of the NMF problem into two blocks $\mathbf{W}$ and $\mathbf{H}$. Then solve the following equations iteratively until a stopping criteria is satisfied.

$$\begin{aligned} \mathbf{W} &\leftarrow \underset{\mathbf{W}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \mathbf{W}\mathbf{H}^T\right\|_F^2, \\ \mathbf{H} &\leftarrow \underset{\mathbf{H}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \mathbf{W}\mathbf{H}^T\right\|_F^2. \end{aligned} \tag{30}$$

The optimization sub-problem for $\mathbf{W}$ and $\mathbf{H}$ are essentially non-negativity constrained least squares (NLS) which could be solved by a number of methods from generic constrained convex optimization to active-set methods. For the proposed CANMF algorithm in this chapter, we use the sophisticated block principal pivoting [30] method to solve the non-negative least squares problem.

4.1.1 Naively Parallel BPP (NBPP):

The NLS problem with multiple right-hand sides could be parallelized on the observation that the problems for multiple right-hand sides are independent from each other. That is, one could solve several instances of Eq. (10) independently for different $\mathbf{b}$ where $\mathbf{C}$ is fixed. This observation suggests that one could optimize row blocks of $\mathbf{W}$ and $\mathbf{H}$ in parallel. Let us divide $\mathbf{W}$ and $\mathbf{H}$ into row blocks $\mathbf{W}_1, \ldots, \mathbf{W}_P$ and $\mathbf{H}_1, \ldots, \mathbf{H}_P$, respectively. The data matrix $\mathbf{A}$ is then double-partitioned accordingly into row blocks $\mathbf{A}_1, \ldots, \mathbf{A}_P$ and column blocks $\mathbf{A}^1, \ldots, \mathbf{A}^P$ (see Figure 11). With these partitions of the data and the variables, one could implement the BPP algorithm in parallel (see Figure 12). However, one clear disadvantage of the naively parallel BPP algorithm is that the factor matrices need to be broadcasted to all machines to update the next alternating factor (like $MPI\_ALLGATHER$[1]). This increases the communication cost to $O((m+n)klogP)$ since all machines need the most up-to-date value of the factor matrices. This communication cost with out loss of generality is applicable to ANLS based iterative NMF algorithms such as multiplicative update, hierarchical alternating least squares etc. Typically to avoid network congestion, we assume, every process takes its turn to broadcast. In the next subsection, we will discuss an idea that allows one to avoid this communication bottleneck.

[1]On completion of this MPI routine, all the nodes will have everyone's data.

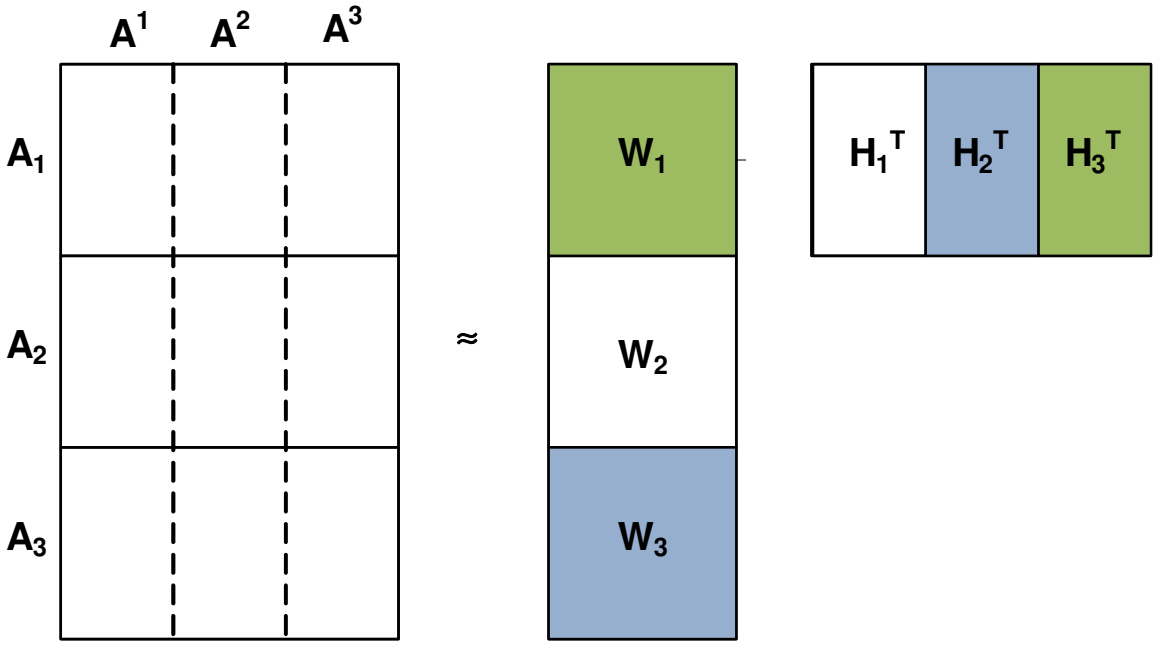


Figure 11: Data & variables partition for parallel BPP.

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}, k \ll \min(m, n)$. $\mathbf{A}$ is double-partitioned into row blocks $\mathbf{A}^1, \ldots, \mathbf{A}^P$ and column blocks $\mathbf{A}^1, \ldots, \mathbf{A}^P$ over P processors.
1. (parallel) Initialize $\mathbf{H}_i$'s.
2. Loop $t = 1, 2, \ldots$
2.1. (parallel) Update $\mathbf{W}_i \leftarrow \underset{\mathbf{W}_i \geqslant 0}{\operatorname{argmin}} f_{A_i}(\mathbf{W}_i, \mathbf{H})$.
2.2. Concatenate $\mathbf{W}_i$'s into $\mathbf{W}$ and broadcast.
2.3. (parallel) Update $\mathbf{H}_i \leftarrow \underset{\mathbf{H} \geqslant 0}{\operatorname{argmin}} f_{A^i}(\mathbf{W}, \mathbf{H}_i)$.
2.4. Concatenate $\mathbf{H}_i$'s into $\mathbf{H}$ and broadcast.

Figure 12: Naively parallel BPP.

4.1.2 Communication Avoiding NMF (CANMF)

In high performance computing settings where the data are often split and reside in different machines, only a block of $\mathbf{A}$ is available for each processor. Let us look at the update of the factor matrices in each machine. Figure 13 shows that for a given data block $\mathbf{A}_{*J}$ (assume column partition), the corresponding block $\mathbf{H}_{J*}$ can be updated

via the optimization problem

$$\mathbf{H}_{J*} \leftarrow \underset{\mathbf{H}_{J*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{*J} - \mathbf{W}\mathbf{H}_{J*}^T \right\|_F^2 . \tag{31a}$$

Similarly, the block $\mathbf{W}_{I*}$ could be updated if the corresponding row block $\mathbf{A}_{I*}$ is available in the machine as

$$\mathbf{W}_{I*} \leftarrow \underset{\mathbf{W}_{I*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{I*} - \mathbf{W}_{I*}\mathbf{H}^T \right\|_F^2 . \tag{31b}$$

Both updates use the full data blocks $\mathbf{A}_{*J}$ and $\mathbf{A}_{I*}$ available in the machine. As a result, these updates require the full factor matrix $\mathbf{W}$ to be sent to all machines that are going to update its $\mathbf{H}$'s blocks. Similarly, in case the machines are going to update $\mathbf{W}$'s blocks, they will need the full matrix $\mathbf{H}$.

The main bottle neck in implementing parallel alternating NMF algorithms is thus the communication of the factor matrices. In [31], the authors proposed using a subset of rows or columns of the original matrix to improve the performance of BPP. In order to proceed, we also propose to use only a sub-block $\mathbf{A}_{I,J}$ of $\mathbf{A}_{*J}$ or $\mathbf{A}_{I*}$ to optimize the corresponding sub-blocks $\mathbf{W}_{I*}$ and $\mathbf{H}_{J*}$ (see Figure 13). The optimization problems become

$$\mathbf{H}_{J*} \leftarrow \underset{\mathbf{H}_{J*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{I,J} - \mathbf{W}_{I*}\mathbf{H}_{J*}^T \right\|_F^2 , \tag{32a}$$

$$\mathbf{W}_{I*} \leftarrow \underset{\mathbf{W}_{I*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{I,J} - \mathbf{W}_{I*}\mathbf{H}_{J*}^T \right\|_F^2 . \tag{32b}$$

The advantages of this update scheme are two fold

- The sub-problems are smaller. Each machine uses a sub-block $\mathbf{A}_{I,J}$ instead of full column or full row blocks $\mathbf{A}_{*J}, \mathbf{A}_{I*}$.

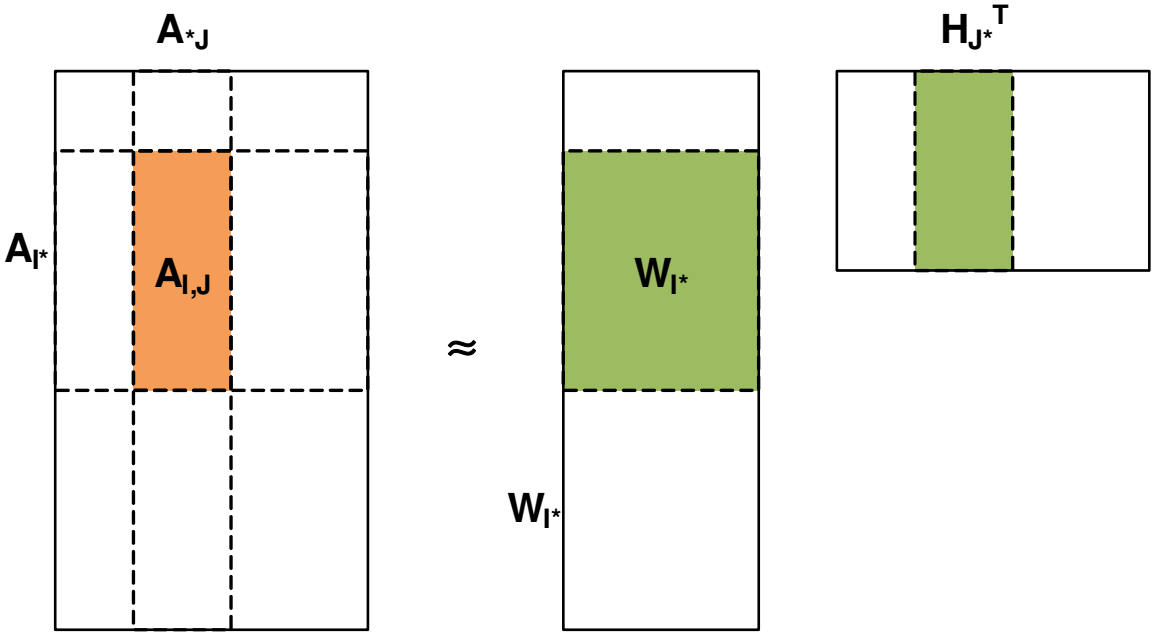


Figure 13: Communication Avoiding NMF

- Each machine only needs a block of the factor matrices to optimize the other factor matrix's block. The communication cost among processors is expected to be reduced significantly especially when the data is large.

4.1.2.1 Large sample justification for CANMF

Let us consider the update of a column $\mathbf{h}$ of $\mathbf{H}_{J*}^T$ in Eq. (32a). We would like that the solution from this update is close to the solution that uses the whole matrix $\mathbf{W}$ and the corresponding column $\mathbf{a}$ of $\mathbf{A}_{*J}$ in Eq (31a).

Denote $f_{\mathbf{W},\mathbf{a}}(\mathbf{h}) = \frac{1}{2\text{len}(\mathbf{a})}\|\mathbf{W}\mathbf{h} - \mathbf{a}\|_2^2$, where $\text{len}(\mathbf{a})$ is the length of $\mathbf{a}$ and denote $\mathbf{h}^I = \arg\min_{\mathbf{h}\geqslant 0} f_{\mathbf{W}_{I*},\mathbf{a}_I}(\mathbf{h})$ the minimum for Eq. (32a) and $\mathbf{h}^* = \arg\min_{\mathbf{h}\geqslant 0} f_{\mathbf{W},\mathbf{a}}(\mathbf{h})$ the minimum for Eq. (31a).

Let $\mathcal{P}$ be the uniform distribution on the row indices $\{1, 2, \ldots, m\}$. We have that

the set of indices I is drawn identically and independently from $\mathcal{P}$ and

$$f_{\mathbf{W},\mathbf{a}}(\mathbf{h}) = E_{i\sim\mathcal{P}}[f_{\mathbf{W}_{i*},a_i}(\mathbf{h})]$$

If $f_{\mathbf{W}_{i*},a_i}(\mathbf{h})$ is bounded (which is the case in the experiments performed in Section 5.5), for a given $\epsilon > 0$, by Hoeffding inequality, with high probability $1 - \delta(\epsilon) = 1 - 2\exp\{-O(\text{cardinality}(I)^2\epsilon^2)\}$, we have

$$|f_{\mathbf{W}_{I*},\mathbf{a}_I}(\mathbf{h}) - E_{i\sim\mathcal{P}}[f_{\mathbf{W}_{i*},a_i}(\mathbf{h})]| \leqslant \epsilon, \forall \mathbf{h}.$$

From the optimality of $\mathbf{h}^I$ and $\mathbf{h}^*$, we have

$$\begin{aligned} f_{\mathbf{W}_{I*},\mathbf{a}_I}(\mathbf{h}^I) &\leqslant f_{\mathbf{W}_{I*},\mathbf{a}_I}(\mathbf{h}^*) \\ \Rightarrow f_{\mathbf{W},\mathbf{a}}(\mathbf{h}^*) \leqslant f_{\mathbf{W},\mathbf{a}}(\mathbf{h}^I) &\leqslant f_{\mathbf{W},\mathbf{a}}(\mathbf{h}^*) + 2\epsilon \end{aligned} \tag{33}$$

If W has full column rank then by strong convexity and optimality of $\mathbf{h}^*$,

$$\begin{aligned} f_{\mathbf{W},\mathbf{a}}(\mathbf{h}^I) &\geqslant f_{\mathbf{W},\mathbf{a}}(\mathbf{h}^*) + \frac{\sigma_{\min}(\mathbf{W})}{2}\|\mathbf{h}^I - \mathbf{h}^*\|_2^2 \\ \Rightarrow \|\mathbf{h}^I - \mathbf{h}^*\|_2^2 &\leqslant \frac{4\epsilon}{\sigma_{\min}(\mathbf{W})}, \end{aligned} \tag{34}$$

where $\sigma_{\min}(\mathbf{W})$ is the smallest positive singular value of $\mathbf{W}$. Therefore, the sub-block based update $\mathbf{h}^I$ is close to the original update $\mathbf{h}^*$ with high probability when the smallest singular value of $\mathbf{W}$ is relatively large, i.e., when $\mathbf{W}$ is well conditioned.

4.2 Distributed Implementation

We now have the necessary tools to construct the new NMF algorithm, Communication Avoiding NMF. The sub-block update discussed in the previous section allows us to exploit the structure of the matrix factorization problem to derive a Communication Avoiding NMF algorithm. The idea is to choose a set of *independent blocks*

of the data matrix $\mathbf{A}$ so that each machine could use a block of data and update the corresponding blocks of the factor matrices $\mathbf{W}, \mathbf{H}$ independently in a distributed manner. After all the machines have updated its blocks variables, they could communicate theirs blocks of variables to other machines. This effectively forces a different set of independent blocks of the data matrix $\mathbf{A}$ to be used in the next iteration. In this section, we will discuss the distributed implementation of CANMF algorithm components: *initialization*, *computation* of $\mathbf{W}, \mathbf{H}$'s blocks, and *communication*.

Let us assume that each machine only keeps a row block of the data $\mathbf{A}$, denote $\mathbf{A}_i$ the block of the i-th machine, $i = 1, \ldots, P$. We also split the variables in $\mathbf{W}$ and $\mathbf{H}$ each into P blocks $\mathbf{W}_1, \ldots, \mathbf{W}_P$ and $\mathbf{H}_1, \ldots, \mathbf{H}_P$ (See Figure 14 for initial configuration). The blocks $\mathbf{A}_i$ are also split accordingly into $\mathbf{A}_{i1}, \ldots, \mathbf{A}_{iP}$. In the first iteration, the i-th machine will manage $\mathbf{W}_i$ and $\mathbf{H}_i$.

4.2.1 Initialization

One could initialize $\mathbf{W}$ and $\mathbf{H}$ distributively across all machines. In our implementation, we initialize $\mathbf{W}$ and $\mathbf{H}$ elements uniformly random in $[0, 1]$. As the blocks are split among machines, we initialize the random seed taking into account the indices of the machines to avoid using the same seed value in all machines. In particular, we use the following C++ command

```
srand( time(NULL) ^ (rank+1} );
```

where `rank` is the 0-based index of the machine. It should be noted that we only need to initialize either $\mathbf{W}$ or $\mathbf{H}$ depending on the update order of the alternating algorithm. In particular, if $\mathbf{H}$ is updated before $\mathbf{W}$ then $\mathbf{W}$ should be initialized. Otherwise, $\mathbf{H}$ should be initialized.

4.2.2 Computation of W and H blocks

During the execution of the algorithm, each machine is responsible for a block in $\mathbf{W}$ and a block in $\mathbf{H}$. Denote r_i and c_i the indices of $\mathbf{W}$ and $\mathbf{H}$'s blocks that the i-th

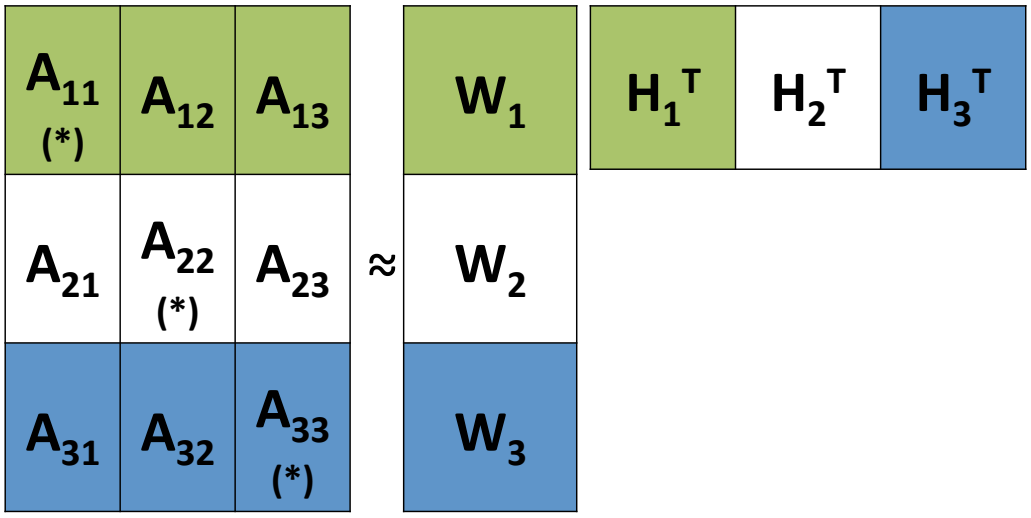


Figure 14: Initial configuration: (*) indicate the data blocks involved in the initial iteration, colours distinguish the different machines' data and variables.

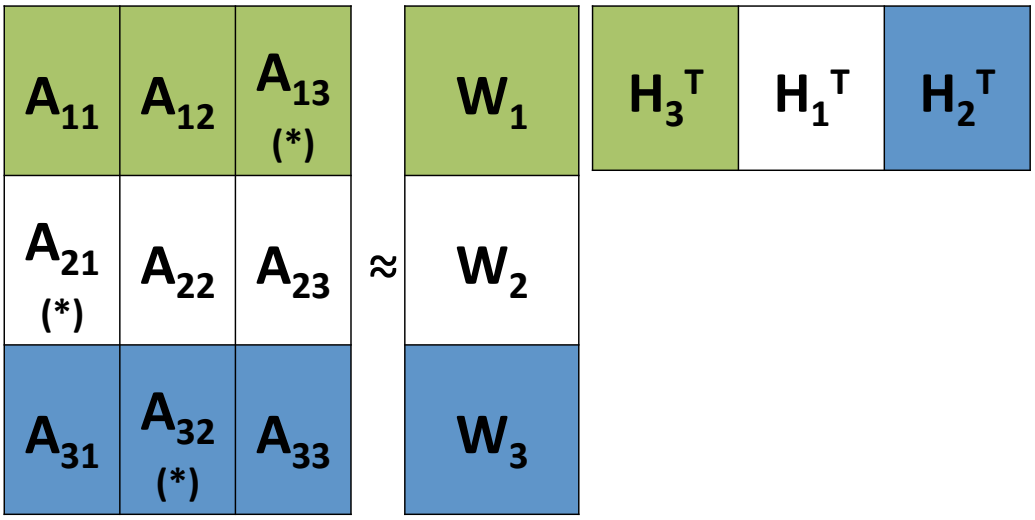


Figure 15: Next configuration: (*) indicate the used data blocks, colors distinguish the machines' data and variables. $\mathcal{M}_1$ manages $\mathbf{W}_1$ and $\mathbf{H}_3$, $\mathcal{M}_2$ manages $\mathbf{W}_2$ and $\mathbf{H}_1$, $\mathcal{M}_3$ manages $\mathbf{W}_3$ and $\mathbf{H}_2$.

machine is currently managing. The computation carried out in the i-th machine is the alternating update of $\mathbf{W}_{r_i}$ and $\mathbf{H}_{c_i}$ as follows

$$\mathbf{H}_{c_i} \leftarrow \underset{\mathbf{H}_{c_i} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{r_i,c_i} - \mathbf{W}_{r_i} \mathbf{H}_{c_i}^T \right\|_F^2, \tag{35a}$$

$$\mathbf{W}_{r_i} \leftarrow \underset{\mathbf{W}_{r_i} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}_{r_i,c_i} - \mathbf{W}_{r_i} \mathbf{H}_{c_i}^T \right\|_F^2. \tag{35b}$$

where $\mathbf{A}_{r_i,c_i}$ is the corresponding data block (see Figure 14). At this point, it is very tempting to apply BPP on the current data block $\mathbf{A}_{r_i,c_i}$ to find $\mathbf{W}_{r_i}$ and $\mathbf{H}_{c_i}$. However, the optimality of BPP means that all previous computation efforts using *other data blocks* are ignored. Therefore, we propose a dampening update as follows

$$\mathbf{H}_{c_i} \leftarrow \alpha \mathbf{H}'_{c_i} + (1-\alpha)\mathbf{H}_{c_i}, \tag{36a}$$

$$\mathbf{W}_{r_i} \leftarrow \alpha \mathbf{W}'_{r_i} + (1-\alpha)\mathbf{W}_{r_i}, \tag{36b}$$

where $\mathbf{W}'_{c_i}$ and $\mathbf{H}'_{c_i}$ are the solutions returned by BPP. We choose step size α in the form $t^{-1/\beta}$ with $\beta \geqslant 2$ and t is the current iteration number. This guarantees a diminishing step size while the sum of all step sizes approaches infinity when the iteration number is large.

In the first iteration, the algorithm would use only the diagonal blocks of the data matrix $\mathbf{A}$. As discussed below, in all iterations, although only P blocks of $\mathbf{A}$ are used, all blocks of $\mathbf{W}$ and $\mathbf{H}$ are updated in parallel.

4.2.3 Communication

4.2.3.1 Cyclic updates

After each machine has updated its blocks $\mathbf{W}_{r_i}$ and $\mathbf{H}_{c_i}$, there are two choices to proceed

- Keeping alternating between $\mathbf{W}_{r_i}$ and $\mathbf{H}_{c_i}$ to further optimize these blocks.

- Sending either $\mathbf{W}_{r_i}$ or $\mathbf{H}_{c_i}$ to other machines so that other machines could update their block of variable with new information.

Obviously, the first choice will only utilize the same blocks of data matrix $\mathbf{A}$ over and over again and no new block of data is introduced into the optimization process. At a certain point, a machine has to communicate its block of variables to other machines so that the data matrix $\mathbf{A}$ is gradually fully utilized. When every machine communicates its block to another machine and uses the received block to update its variables, one has used the *Jacobi* iterative method where each block is updated using the currently available variables not the most up-to-date ones as in the Gauss-Seidel method.

Inspired by the parallel Jacobi methods in [41, 42], we propose that each machine would send the $\mathbf{H}_{c_i}$ blocks to the next machine in a cyclic manner as follows

$$\mathcal{M}_i \overset{\mathbf{H}_{c_i}}{\rightarrow} \mathcal{M}_{i+1}, \mathcal{M}_P \overset{\mathbf{H}_{c_P}}{\rightarrow} \mathcal{M}_1$$

where $\mathcal{M}_i$ is the i-th machine, $i = 1, \ldots, P-1$. In other words, if we use superscript t to denote the iteration number then we have

$$\begin{aligned} c_{i+1}^{(t+1)} &= c_i^{(t)}, c_1^{(t+1)} = c_P^{(t)}, \\ r_i^{(t+1)} &= r_i^{(t)}, i = 1, \ldots, P. \end{aligned} \tag{37}$$

The $\mathbf{W}$'s blocks remain unmoved because each machine only has the corresponding row block of $\mathbf{A}$. In the next iteration, as each machine will manage its new blocks and would use a different data block to carry out its computation (see Figure 15). As seen in the figure, the next set of data blocks is a new set of independent blocks. This allows all machines to update its blocks of variables independently of other machines in a distributed manner.

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}, k \ll \min(m,n)$ and P machines $\mathcal{M}_i, i = 1, \ldots, P$. $\mathbf{A}$ is split into row blocks $\mathbf{A}_1, \ldots, \mathbf{A}_P$.
1. (parallel) Initialize $r_i = c_i = i, \forall i$.
2. (parallel) Initialize $\mathbf{W}_i, \mathbf{H}_i$ uniformly random from $[0,1]$ using different seed in each machine.
3. Loop $t = 1, 2, \ldots$
3.1. (parallel) Find $\mathbf{W}_{r_i}, \mathbf{H}_{c_i}$ with Eq. (35) as $\mathbf{W}'_{r_i}, \mathbf{H}'_{c_i}$.
3.2. (parallel) Update $\mathbf{W}_{r_i} \leftarrow \alpha \mathbf{W}'_{r_i} + (1-\alpha)\mathbf{W}_{r_i}$
3.3. (parallel) Update $\mathbf{H}_{c_i} \leftarrow \alpha \mathbf{H}'_{c_i} + (1-\alpha)\mathbf{H}_{c_i}$
3.4. (parallel) if $\mod(t, P) \neq 0$, send $\mathbf{H}_{c_i}$ from $\mathcal{M}_i$ to $\mathcal{M}_{i+1}, i = 1, \ldots, P-1$, send $\mathbf{H}_{c_P}$ from $\mathcal{M}_P$ to $\mathcal{M}_1$. Update c_i with Eq. (37).
3.5. (parallel) if $\mod(t, P) = 0$, generate a random permutation π and send $\mathbf{H}_{c_i}$ from $\mathcal{M}_i$ to $\mathcal{M}_{\pi_i}, i = 1, \ldots, P$. Update c_i accordingly.

Figure 16: Communication Avoiding NMF.

4.2.3.2 Communication overhead

The communication cost of each iteration of CANMF is only $O(nk/P)$ because each machine sends and receives 1 block from the adjacent machine, all in parallel. This is a clear improvement compared to the overhead of naive parallel implementation that required an $MPI\_ALLGATHER$ for the low rank factors. Furthermore, as one is always able to make $n < m$ (transpose $\mathbf{A}$ if necessary), we can make the communication cost even smaller. The CANMF algorithm is summarized in Figure 16. Note that, when we send $\mathbf{H}_{c_i}$ from a machine to another, we update c_i accordingly. In Section 5.5, as part of the scalability experiments, we show the advantage of CANMF's communication cost.

One of the important challenge in the CANMF algorithm is determining the parameter α. Since each sub problem in Eq. (35), is constrained convex problem determining α becomes simple. We define α as $\frac{1}{t^{\frac{1}{\alpha}}}$. We ran the algorithm with an initial α, say $\frac{1}{\sqrt{t}}$ and determined the α that gave maximum error reduction between two successive iterations.

4.3 Experiments

In the data mining and machine learning community, there had been a large interest in using Hadoop for large scale implementation. Hadoop does lots of disk I/O and was designed for processing gigantic text files. Many of the real world data sets that is available for research are large scale sparse internet text data such as bag of words, recommender systems, social networks etc. Towards this end, there had been interest towards Hadoop implementation of matrix factorization algorithm [17, 39, 37]. However, the use of NMF is beyond the sparse internet data and also applicable for dense real world data such as video, image etc. Hence in order to keep our implementation applicable to wider audience, we chose MPI for distributed implementation. Apart from the application point of view, we decided MPI C++ implementation for other technical advantages that is necessary for scientific application such as (1) it can leverage the recent hardware improvements (2) effective communication between nodes (3) availability of numerically stable BLAS and LAPACK routines etc. We identified a few synthetic and real world datasets to experiment with our MPI implementation and a few baselines to compare our performance.

4.3.1 Datasets

We used sparse and dense matrices that are synthetically generated and from real world. We will explain the datasets in this section.

- Dense Synthetic Matrix($DSYN$): We generated two uniform matrices of size 100000x100 and 100x50000. We multiplied these two matrices that yielded a dense matrix of size 100000x50000 and added standard gaussian noise.

- Sparse Synthetic Matrix ($SSYN$): We used Matlab's sprandn generator to generate a sparse matrix of size 200000x100000 with density of 0.1. We shifted the matrix to construct a non-negative matrix.

- Dense Real World data(*Video*): Generally, NMF is performed in the video data for back ground subtraction to detect the moving objects. The low rank matrix $\hat{\mathbf{A}} \approx \mathbf{W}\mathbf{H}^T$ represents background and the error matrix $\mathbf{A} - \hat{\mathbf{A}}$ has the moving objects. Detecting moving objects find many applications in real world such as traffic estimation, security monitoring etc. In the case of detecting moving objects, only the last minute or two video is taken from the live video camera. The algorithm to incrementally adjust the NMF based on the new streaming video is presented in [28]. To simulate this scenario, we collected a video in a busy intersection of our 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, such matrix was dense of size 1.3 million x 2400.

- Sparse Real World data (*Wiki*): We collected 810678 documents written in English from wikipedia excluding any meta pages like disambiguation pages. There were totally 1395551 distinct terms across all these documents. From this original unprocessed term-document matrix, we discarded words that appeared less than 3 times and documents containing less than five words. After this preprocessing, the matrix reduced to 387086x736048, where columns were documents and rows were vocabulary. We computed the tf-idf of this matrix and used it for experiments. In the entire experimentation section other than scalability results, sparse real world data means this specific data set.

- Sparse Real World data *Webbase* : To understand the scalability of the CANMF algorithm, we identified this dataset of a very large directed sparse graph with near 118 million nodes (118,142,155) with nearly 1 billion edges (1,019,903,190). The dataset was first reported by Boldi and Vigna [4]. We use the matrix market sparse representation for this graph only for conducting the scalability experiment on a cluster.

We compare our proposed algorithm CANMF on the above explained datasets using the following baselines.

4.3.2 Baselines

The objective of this experimentation is to empirically understand our Large Sample Justification explained in Section 4.1.2.1. That is., understand how good are the low rank factors $\mathbf{W}$ and $\mathbf{H}$ generated by our algorithm CANMF. Hence, we compare the relative residual error $\frac{\|\mathbf{A} - \mathbf{W}\mathbf{H}^T\|_F}{\|\mathbf{A}\|_F}$ against the carefully chosen baselines.

For baselines we wanted to use the state of the art distributed NMF algorithms. However all the distributed NMF algorithms were designed for sparse matrices. When we converted our dense matrix of size 35GB to sparse matrix coordinate format, the size of the matrix file was blown up to terabytes. We tested the Hadoop based MU implementation on a small dense matrix of size 2000x2000. Every iteration of MU on Hadoop involves 6 steps and each of these steps was writing almost the size of matrix into the HDFS. Because of these huge I/O operation even a small matrix was taking like 9 minutes to complete one iteration. This made running these experiments almost impossible for large dense matrices. Hadoop implementations are suitable for very large sparse matrices to run on sophisticated massive Hadoop cluster. Also, when we carefully analyzed the algorithm, we could understand that the state of the art distributed scalable NMF algorithms were classical algorithm with each individual matrix operations tuned to run distributively in Hadoop. That is., we could very well compare the quality of our algorithm against classical NMF algorithms instead of using the Hadoop implementation. Also, using these implementation for sparse matrices had different challenges. Some of these algorithms considered the zero values as unobserved for supporting recommender systems rating matrix. Considering these reasons, we choose the following baselines implemented in C++.

4.3.2.1 Multiplicate Update(MU)

Until the stopping criteria is satisfied, update $\mathbf{H}$ and $\mathbf{W}$ using the most recent $\mathbf{H}$ and $\mathbf{W}$ as follows:

$$w_{ij} \leftarrow w_{ij} \frac{(\mathbf{AH})_{ij}}{(\mathbf{WH}^T\mathbf{H})_{ij}}$$
$$h_{ij} \leftarrow h_{ij} \frac{(\mathbf{A}^T\mathbf{W})_{ij}}{(\mathbf{HW}^T\mathbf{W})_{ij}}$$

4.3.2.2 Hierarchical Alternating Least Squares(HALS)

The objective function for HALS can be explained as follows

$$f(\mathbf{w}_1, \ldots, \mathbf{w}_k, \mathbf{h}_1, \ldots, \mathbf{h}_k) = \left\| \mathbf{A} - \sum_{j=1}^{k} \mathbf{w}_j \mathbf{h}_j^T \right\|_F^2, \tag{38}$$

where $\mathbf{W} = [\mathbf{w}_1, \ldots, \mathbf{w}_k] \in \mathbb{R}^{m \times k}$ and $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_k] \in \mathbb{R}^{n \times k}$. Eq. (38) shows $\mathbf{A}$ is approximated by sum of k rank-one matrices. Following the BCD framework, we can minimize f by iteratively solving until stopping criteria

$$\mathbf{w}_i = \underset{\mathbf{w}_i \geqslant 0}{\operatorname{argmin}} f(\mathbf{w}_1, \ldots, \mathbf{w}_k, \mathbf{h}_1, \ldots, \mathbf{h}_k),$$
$$\mathbf{h}_i = \underset{\mathbf{h}_i \geqslant 0}{\operatorname{argmin}} f(\mathbf{w}_1, \ldots, \mathbf{w}_k, \mathbf{h}_1, \ldots, \mathbf{h}_k),$$

for $i = 1, 2, \ldots, k$.

Additionally we chose the Naive Parallel BPP (NBPP) explained in Section 2.2.1. The above C++ baselines are implemented to run on standalone shared memory systems. Hence comparing the wall clock time of the baselines MU and HALS with the distributed implementation of NBPP is unfair. However, we compare the communication and compute time and the speed up of the two distributed algorithm NBPP and CANMF run on same environment are presented in Table 8, Figure 21 and 20. We compared the CANMF relative residual error against MU, HALS and

NBPP for sparse synthetic dataset, dense synthetic dataset, dense video data and sparse text data for different low rank k.

4.3.2.3 Scalability Experiment

We wanted to study the scalability of our algorithm on all the datasets. We swept the number of MPI processes from 1 to 30 in step size of 5. For this scalability study, we considered the matrix is already available in the main memory and ignored the disk access time. For example, in the case of C++ implementation, loading a 35GB dense matrix file into the memory as float array approximately took 2800 seconds that was nearly 4/5 of the total time.

4.3.3 Initialization

To ensure fair comparison among algorithms, the same random seed was used across different methods appropriately. For example., the initial random matrix **W** and **H** was generated with the same random seed when testing with different algorithms. It is important to note that only one of the two matrices **W** and **H**, required to be initialized. This ensures, all the algorithm are started with the same initializations. Similarly, for generating the synthetic input matrix **A**, same random seed was used.

4.3.4 Experimental machines

We experimented the code in two different setup. A very large shared memory multi core system and a cluster of distributed memory of small systems. In the case of former shared memory system, it is a quad socket machine where each socket is mounted with Intel(R) Xeon(R) CPU E7- 8870 processor. Each processor has 10 cores totally making 40 cores available for the conducting the experiments and this machine had 256GB of memory. In the case of distributed memory, the cluster consists of 40 compute nodes, 32 of which are Intel architecture and 8 of which are AMD 64 bit architecture. These nodes are networked together via gigabit ethernet with a frontend

node for control. Each machine has 8 cores and 24GB of memory. We used this cluster for running the sparse real world *webbase* dataset. The rest of the experiments were conducted on shared memory multicore system.

4.3.5 Observation

The graphs of relative error over iterations and low rank k for dense synthetic, sparse synthetic, dense real world and sparse real world are presented in Figure 17,18,19 and 6 respectively. It is very important to define the iteration for CANMF at this juncture. For CANMF algorithm, one iteration defines solving $\mathbf{H}$ in parallel for one configuration as explained in Figure 14. That means the reported error is obtained by seeing only $\frac{1}{P}$ portion of the data.

Also, to begin with, we evaluate the quality of our proposed algorithm CANMF with a set of baselines explained above. Evaluating the goodness of a low rank approximation has a lower bound of at least $O(mn)$. That is., we have to see all the entries in the low rank matrix $\hat{\mathbf{A}} = \mathbf{WH}$. In our case., it is $O(mnk)$, the time for multiplying the two low rank matrix $\mathbf{W}$ and $\mathbf{H}$. This is prohibitively expensive to evaluate the goodness of the low rank approximation for very large matrix. Recall, for all practical applications, even though the input matrix $\mathbf{A}$ is sparse, the low rank factors $\mathbf{W}$ and $\mathbf{H}$ are generally dense and will result in a dense low rank approximation which also takes up huge memory in space $O(mn)$. It is also difficult to obtain information such as qualitative labels for evaluating the goodness of low rank approximation. Hence for evaluating the quality of our proposed CANMF algorithm, we chose a large synthetic matrix of size 100,000 x 50,000, a large dense and sparse real world data. In the case of very large real world datasets *webbase*, we present only the scalability results. We will discuss each of this observation over different datasets in detail.

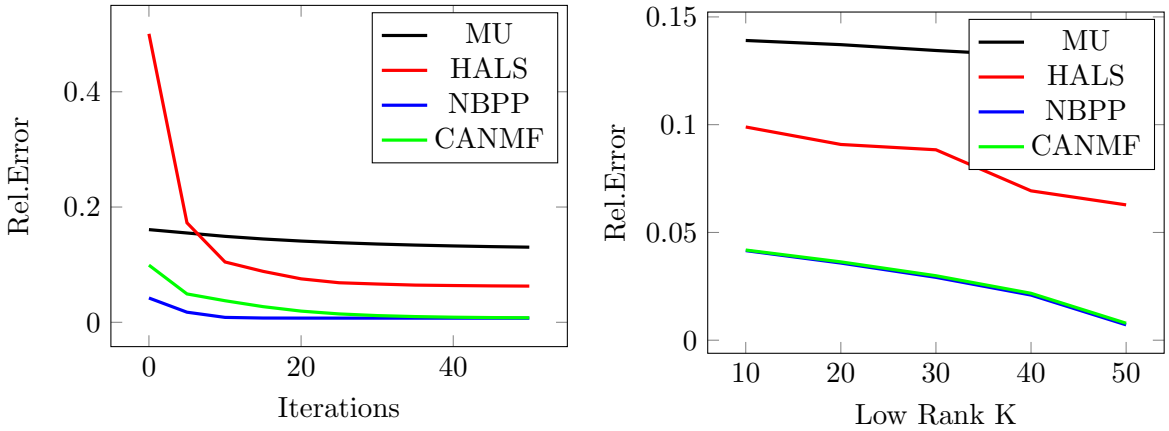


Figure 17: Relative Error Comparison of CANMF with Baselines – Dense Synthetic

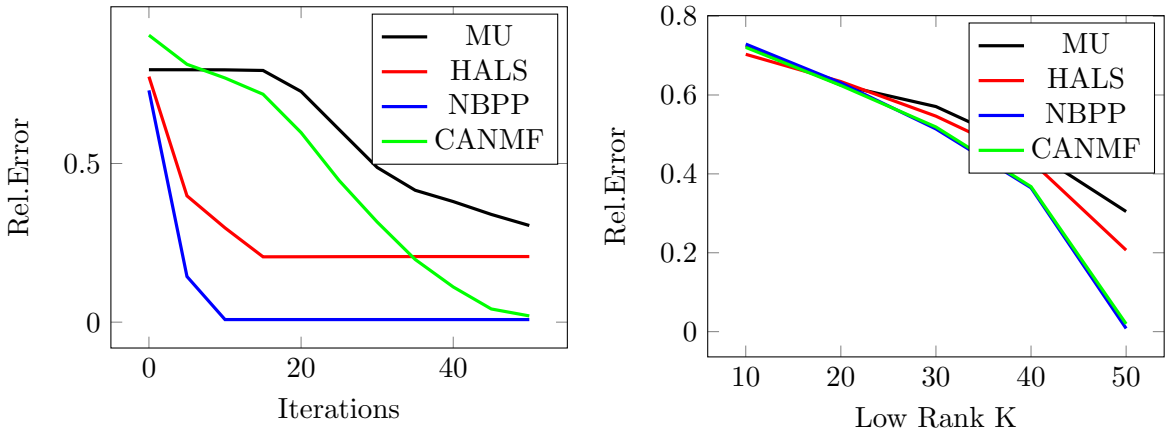


Figure 18: Relative Error Comparison of CANMF with Baselines –Sparse Synthetic

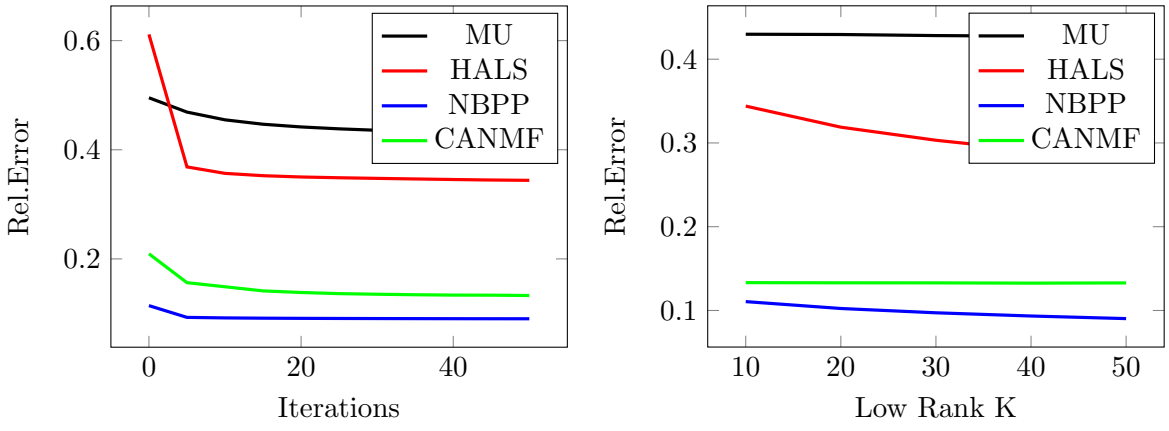


Figure 19: Relative Error Comparison of CANMF with Baselines – Video Realworld

4.3.6 Relative Error

In the case of dense synthetic dataset, we chose the low rank $k = 50$ and we used 40 MPI processes for NBPP and CANMF. The relative error of the four algorithms MU, HALS, NBPP and CANMF are presented in Figure 17 by running for 50 iterations. From the experiments, we observe that the improvement in error over iterations of MU was really slow. Even though HALS, started with higher error, it's relative error was much better than MU over iterations. HALS surpassed MU's relative error in as low as 10 iterations. NBPP was better than MU and HALS in all the iterations. Even though CANMF started with a slightly higher rate, it was fast to match with NBPP and produced a similar result over 50 iterations. As explained before, this difference was mainly because of percentage of data coverage for CANMF is different from NBPP. That is, CANMF sees all the data once only after 40 iterations when we use 40 MPI processes. However NBPP sees all the data for every iteration and would have seen the entire data for 40 times at the end of 40 iterations. For, CANMF, it took 50 iterations to achieve the error of 0.007 that was achieved by NBPP in 15 iterations. At the end of the 50 iterations, the relative error of MU, HALS, NBPP and CANMF, where 0.13034, 0.06279, 0.00712 and 0.00787 respective. Even though CANMF's relative error was slightly more than NBPP, this difference is observable only for higher low rank k. In low k's such as 10 and 20, after 50 iterations CANMF was equally as good as NBPP. It is important to understand that, in the case of matrices with completely known entries, higher the k, better the relative error. Our proposed algorithm CANMF, reflected this fact and it performed better over higher k's over MU and HALS. For eg., in the case of low rank 10, the difference of relative error between CANMF - MU was 0.09. This margin increased with higher k. That is., at low rank 50, this difference between CANMF and MU improved to 0.12247. Over all the performance of CANMF was very much similar with NBPP even though the speed up of CANMF was much superior over

NBPP. That is., the quality of CANMF was not compromised for better speed up.

In the case of sparse matrices, one of the major challenge for the CANMF algorithm is introduction of zero rows and columns in a particular block. For example, the sub block of the sparse input matrix $\mathbf{A}_{I,J}$ in Equation (32) can have zero rows or columns even though there need not be one in $\mathbf{A}_{I,*}$ or $\mathbf{A}_{*,J}$. To over come this problem, we chose to solve only the sub block with non-zero rows and columns. That is.,

$$\begin{aligned} \mathbf{H}'_{J*} &\leftarrow \underset{\mathbf{H}'_{J*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}'_{I,J} - \mathbf{W}_{I*}\mathbf{H}_{J*}^{T'} \right\|_F^2, \\ \mathbf{W}'_{I*} &\leftarrow \underset{\mathbf{W}'_{I*} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A}'_{I,J} - \mathbf{W}'_{I*}\mathbf{H}_{J*}^{T} \right\|_F^2. \end{aligned} \tag{39}$$

where, $\mathbf{A}'_{I,J}$ is a sub block of $\mathbf{A}_{I,J}$, that is without zero columns for $\mathbf{H}'_{J*}$ and zero rows for $\mathbf{W}'_{I*}$ and use this to update the corresponding rows of $\mathbf{H}_{J*}$ and $\mathbf{W}_{*I}$. With this modification CANMF ran seamlessly for very large sparse matrices. As shown in the Figure 18 and 6 for synthetic and real world data respectively, the performance of HALS and NBPP with relative error over number of iterations was almost similar to that of dense matrix. However CANMF's, behavior was different from dense. In the case of CANMF, this behavior is attributed to the zero rows and columns that gets introduced in the sub block of $\mathbf{A}'_{I,J}$ and for every iteration, the CANMF processes only $\frac{1}{P}$ *really sparse portion of the data when ran with P distributed processes.* CANMF surpasses HALS and matches with NBPP after two or more times of completely visiting the data. After 50 iterations over 40 MPI processes the relative error of MU, HALS, NBPP and CANMF on synthetic data were 0.30461, 0.20672, 0.00810 and 0.01967 respectively. With respect to low rank k, as opposed to dense, in the case of sparse all the four algorithms had a similar behavior with respect to relative error. This is because of the fact that with higher low rank k, we have more degrees of freedom to closely approximate with the more zero entries and less non-zero entries.

Table 8: Error Vs Time

	NBPP			CANMF		
Dataset	Rel.Err	Compute	Comm	Rel.Err	Compute	Comm
DSYN	0.0292	239.10	99.75	0.0299	46.54	4.415
SSYN	0.5139	258.54	91.38	0.5188	225.10	16.07
Video	0.0972	366.97	449.77	0.1329	445.50	64.88
Wiki	0.5188	1774.23	489.55	1.2129	99.43	51.28

Finally, the behavior of the relative error over iterations and the low rank k on real world video dataset as shown in Figure 19 was almost similar to the dense synthetic dataset.

In all the datasets, that we experimented CANMF was more accurate than MU and HALS and almost matched NBPP after few iterations over all low rank k. Also the running time and scalability of CANMF was far superior over NBPP. Even though in some cases of the presented results, NBPP is more accurate than CANMF , in few of our experiments on dense dataset, we observed that CANMF achieved the same error with little more iterations than NBPP, which took much lesser clock time than NBPP over same number of parallel processes P.

4.3.7 Scalability

To appreciate the difference in scalability between sparse and dense matrix, it is important to understand the computational complexity of BPP algorithm. The BPP requires the computation of four matrix matrix products once and a bunch of least squares for each iteration. The matrix matrix multiplication or level-3 BLAS cost is $O(mkn + k^2m)$ and the time spent to find the solution of length k and we are solving for m NNLS vectors for $\mathbf{W}$ and n NNLS vectors for $\mathbf{H}$. Each iteration of the active set method requires at most a $k \times k$ Cholesky decomposition and two $k \times k$ triangular solves.

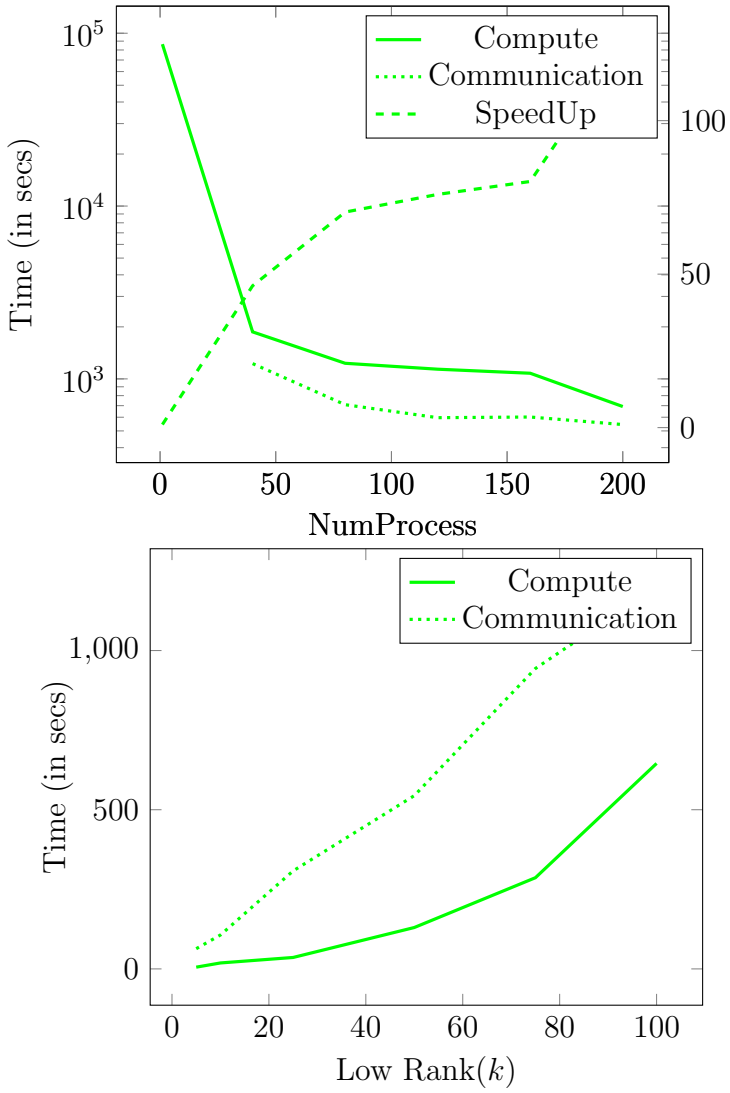


Figure 20: Very Large Sparse Real World Scalability

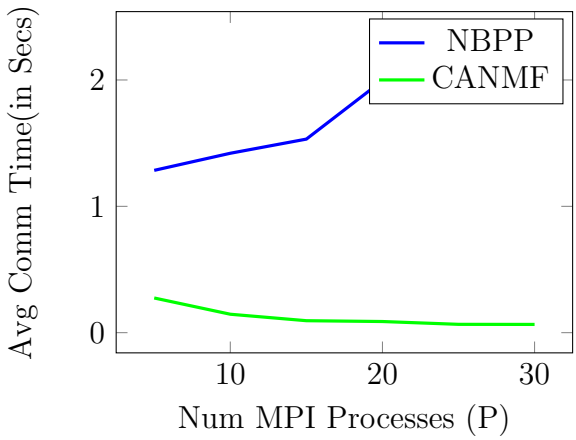


Figure 21: Average Communication Time Per Iteration

The difference between sparse and dense matrices lies in the matrix-matrix multiplication. To be more specific the the sparse input matrix $\mathbf{A}$ and one of the low rank factors $\mathbf{W}$ or $\mathbf{H}$ which takes only $O(m \cdot nnz(\mathbf{A}))$, that reduces to quadratic from cubic for dense. All the rest of the computation, mainly the low rank factor's inner products ($\mathbf{W}^T\mathbf{W}$ and $\mathbf{H}^T\mathbf{H}$) and the active set based NNLS remain the same between sparse and dense input matrix. When the complexity of solving the active set NNLS is more than the complexity of the sparse-dense matrix multiplication(SPMM), the characteristics of speed up will be similar to dense with an over all reduced running time.

We present the improvement in communication of our proposed CANMF over the baseline NBPP in Figure 21. We averaged the communication time over 50 iterations for NBPP and CANMF by sweeping number of MPI processes from 5 to 30 in steps of 5. Since the size of the low rank factors are same in both the sparse and dense matrix, we conducted this experiment only on dense synthetic dataset. We can observe that, for NBPP, the communication time increases with increase in number of processes P and in the case of CANMF decreased with increases in P. Even though the matrix size gets smaller with increase in P, the communication gain

in CANMF has a diminishing advantage. This is due to the latency in MPI Send and Receive calls. Generally, in our experiments, we found the collective MPI calls such as $MPI\_BROADCAST$ and $MPI\_ALLGATHER$ are more communication effective over non blocking MPI_SEND and $MPI\_RECEIVE$. Inspite of these observations, the gains of CANMF is more pronounced over NBPP as P increases.

Similarly, we studied the cost benefits of the proposed CANMF against the baseline NBPP on all the datasets. For this experiment, we ran both CANMF and NBPP on all datasets over 20 MPI processes with the low rank $k = 30$ for 50 iterations. and show the error, communication and compute time after 50 iterations in Table 8. In the case of both large real world dense and sparse datasets, CANMF demonstrated significant reduction in both compute and communication cost over baseline NBPP.

Finally, we would like to present the scalability result of very large sparse real world matrix *webbase* on distributed cluster. We collected both the running time and communication time by varying the number of distributed processes for a low rank $k = 50$ over 5 iterations. The reason for choosing lower iterations is for determining speedup of the CANMF algorithm on distributed cluster. We have to run with only one process for the *webbase* dataset which took almost one complete day for running 5 iterations. Also, we know that the running time of the algorithm is linear over number of iterations and the behavior will not be much of a difference with running over more iterations. Similarly, we also collected the communication cost by varying the low rank k, with 200 distributed processes. The results of both of these experiments are presented in Figure 20. In the case of very large sparse matrix, we achieved a linear speedup and the advantage started diminishing by introducing more and more processes. The communication cost of the CANMF algorithm is $O(nk/P)$, that is for a given low rank k, as the number of processes increases, the communication time of the algorithm must reduce. According to Figure 20, we could validate that for the given low rank $k = 50$, the communication cost was linearly

reducing over the number of processes. Similarly, for a given $P = 200$, as we increase the low rank k, the both the communication and the total running time was increasing almost linearly.

Experimentally, we could validate that CANMF is most sophisticated distributed NMF algorithm considering the relative error and the overall running time of the algorithm for both the sparse and dense datasets.

In this chapter, we presented a new distributed algorithm for Nonnegative Matrix Factorization called CANMF. The algorithm partitions the data matrix and the factor matrices into blocks then optimizes parallely in different machines. The CANMF algorithm utilizes a communication scheme that avoids the communication bottleneck in naively parallel implementation of alternating NMF methods. The communication cost of CANMF is $O(nk/P)$ an improvement over $O((m+n)klogP)$ for naive implementation. The CANMF algorithm performs well qualitatively and computationally for both sparse and dense matrices. Apart from these benefits, it also ensures preserving privacy, where the input data from one machine will not be shared to either a central server or other machines in the cluster.

CHAPTER V

HIGH PERFORMANCE COMPUTING NON-NEGATIVE MATRIX FACTORIZATION

In this chapter, we propose a high-performance distributed-memory parallel algorithm that computes the factorization by iteratively solving alternating non-negative least squares (NLS) subproblems for $\mathbf{W}$ and $\mathbf{H}$. 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 $\mathbf{W}$ and $\mathbf{H}$ within the alternating iterations. We demonstrate the scalability of our algorithm and compare it with baseline implementations, showing significant performance improvements.

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.

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 chapter, 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. [13] 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.

5.1 Preliminaries

5.1.1 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 [51, 6].

5.1.2 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 [51, 6]), 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 $p = 1$.

5.2 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 [37, 14, 56, 17, 39]. Liu et al. propose running Multiplicative Update (MU) for KL divergence, squared loss, and "exponential" loss functions [39]. 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 [37]. Also, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [56]. Gemmula et al. propose a *Generic algorithm* that works on different loss functions, often involving the distributed computation of the gradient [17]. 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 [14]. The authors also provide an implementation that can enforce non-negative constraints on the factor matrices.

We note that Spark [58] 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 [46], 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 [21] and on a GPU [45].

5.3 Foundations

In this section, we will introduce the Alternating-Updating NMF (AU-NMF) framework, multiple methods for solving NMF. We also present a straightforward approach to parallelization of the framework.

5.3.1 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 6.

input : $\mathbf{A}$ is an $m \times n$ matrix, k is rank of approximation
output: $\mathbf{W} \in \mathbb{R}_+^{m \times k}, \mathbf{H} \in \mathbb{R}_+^{k \times n}$
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 $\mathbf{W}^T\mathbf{A}$;

Algorithm 6: $[\mathbf{W}, \mathbf{H}] = \text{AU-NMF}(A, k)$

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 $\mathbf{W}$:

$$\begin{aligned} \mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \tilde{\mathbf{W}}\mathbf{H} \right\|_F, \\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}} \geqslant 0}{\operatorname{argmin}} \left\| \mathbf{A} - \mathbf{W}\tilde{\mathbf{H}} \right\|_F. \end{aligned} \tag{40}$$

Since each subproblem involves nonnegative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [28]. Block Principal Pivoting (BPP), discussed more in detail at Section 2.2.1, 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 (40) to optimality but simply improve the overall objective function (29). Such methods include Multiplicative Update (MU) [50] and Hierarchical Alternating Least Squares (HALS) [10], which was also independently proposed as Rank-one Residual Iteration (RRI) [22]. 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

$$f(\mathbf{w}^1, \cdots, \mathbf{w}^k, \mathbf{h}_1, \cdots, \mathbf{h}_k) = \left\| \mathbf{A} - \sum_{i=1}^{k} \mathbf{w}^i \mathbf{h}_i \right\|_F, \tag{41}$$

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:

$$\begin{aligned} \mathbf{w}^i &\leftarrow \left[\mathbf{w}^i + \frac{(\mathbf{A}\mathbf{H}^T)^i - \mathbf{W}(\mathbf{H}\mathbf{H}^T)^i}{(\mathbf{H}\mathbf{H}^T)_{ii}} \right]_+, \\ \mathbf{h}_i &\leftarrow \left[\mathbf{h}_i + \frac{(\mathbf{W}^T\mathbf{A})_i - (\mathbf{W}^T\mathbf{W})_i\mathbf{H}}{(\mathbf{W}^T\mathbf{W})_{ii}} \right]_+. \end{aligned} \tag{42}$$

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 $\mathbf{H}\mathbf{H}^T$, $\mathbf{A}\mathbf{H}^T$, $\mathbf{W}^T\mathbf{W}$, 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

$$\begin{aligned} w_{ij} &\leftarrow w_{ij} \frac{(\mathbf{A}\mathbf{H}^T)_{ij}}{(\mathbf{W}\mathbf{H}\mathbf{H}^T)_{ij}}, \\ h_{ij} &\leftarrow h_{ij} \frac{(\mathbf{W}^T\mathbf{A})_{ij}}{(\mathbf{W}^T\mathbf{W}\mathbf{H})_{ij}}. \end{aligned} \tag{43}$$

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 [28]. 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 5.4) can be extended to these methods (or any other AU-NMF method) with only a change in local computation.

5.3.2 Naive Parallel NMF Algorithm

In this section we present a naive parallelization of NMF algorithms [13]. 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. (10) 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.

Table 9: Leading order algorithmic costs for Naive and HPC-NMF (per iteration). Note that the computation and memory costs assume the data matrix $\mathbf{A}$ is dense, but the communication costs (words and messages) apply to both dense and sparse cases.

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

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 $\mathbf{W}_i \leftarrow \text{SolveBPP}(\mathbf{H}\mathbf{H}^T, \mathbf{A}_i\mathbf{H}^T)$

;

/* Compute H given W */

5: collect $\mathbf{W}$ on each processor using all-gather

6: p_i computes $(\mathbf{H}^i)^T \leftarrow \text{SolveBPP}(\mathbf{W}^T\mathbf{W}, (\mathbf{W}^T\mathbf{A}^i)^T)$

7:

Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geqslant 0, \tilde{\mathbf{H}} \geqslant 0}{\text{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$

Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

Algorithm 7: $[\mathbf{W}, \mathbf{H}] = \text{Naive}(\mathbf{A}, k)$

Algorithm 7 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 $\mathbf{A}_i$

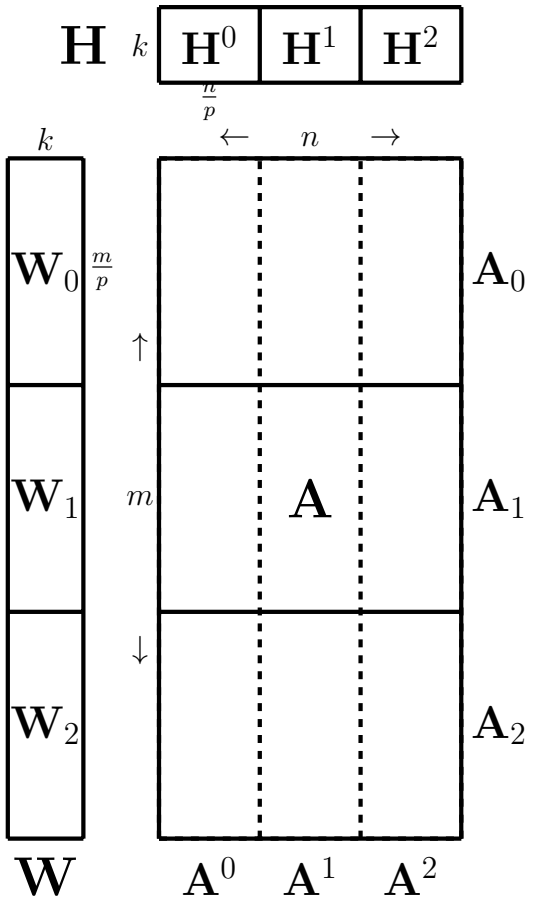


Figure 22: Distribution of matrices for Naive (Algorithm 7), for $p = 3$. Note that $\mathbf{A}_i$ is $m/p \times n$, $\mathbf{A}^i$ is $m \times n/p$, $\mathbf{W}_i$ is $m/p_r \times k$, and $\mathbf{H}^i$ is $k \times n/p$.

and $\mathbf{A}^i$ (see Figure 22). 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 7 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 5.1.2 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 7 in Table 9. This naive algorithm [13] 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 $\mathbf{HH}^T$ and $\mathbf{W}^T\mathbf{W}$ (each processor computes it redundantly), and (3) as we will see in Section 5.4, it communicates more data than necessary.

5.4 High Performance Parallel NMF

We present our proposed algorithm, HPC-NMF , as Algorithm 8. The main ideas of the algorithm are to (1) exploit the independence of NLS problems for rows of $\mathbf{W}$ and columns of $\mathbf{H}$ and (2) use communication-optimal matrix multiplication algorithms to set up the NLS problems. The naive approach (Algorithm 7) shares the first property, by parallelizing over rows of $\mathbf{W}$ and columns of $\mathbf{H}$, 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. [12]. 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. For convenience, 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. (10) 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.

The algorithm uses a 2D distribution of the data matrix $\mathbf{A}$ across a $p_r \times p_c$ grid of processors (with $p = p_r p_c$), as shown in Figure 23. Algorithm 8 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 $\mathbf{A}$ 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 8 are given in Figure 23; we use a 2D distribution of $\mathbf{A}$ and 1D distributions of $\mathbf{W}$ and $\mathbf{H}$. Recall from Table 1 that $\mathbf{M}_i$ and $\mathbf{M}^i$ denote row and column blocks of $\mathbf{M}$, respectively. Thus, the notation $(\mathbf{W}_i)_j$

denotes the jth row block within the ith row block of $\mathbf{W}$. Lines 3–8 compute $\mathbf{W}$ for a fixed $\mathbf{H}$, and lines 9–14 compute $\mathbf{H}$ for a fixed $\mathbf{W}$; 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 9 summarizes the results of this section and compares them to Naive.

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 $k \times n/p$

p_{ij} initializes $(\mathbf{H}_j)_i$

while *stopping criteria not satisfied* **do**

/* Compute W given H */

p_{ij} computes $\mathbf{U}_{ij} = (\mathbf{H}_j)_i (\mathbf{H}_j)_i^T$

compute $\mathbf{H}\mathbf{H}^T = \sum_{i,j} \mathbf{U}_{ij}$ using all-reduce across all procs

▷ $\mathbf{H}\mathbf{H}^T$ is $k \times k$ and symmetric

p_{ij} collects $\mathbf{H}_j$ using all-gather across proc columns

p_{ij} computes $\mathbf{V}_{ij} = \mathbf{A}_{ij}\mathbf{H}_j^T$

▷ $\mathbf{V}_{ij}$ is $m/p_r \times k$

compute $(\mathbf{A}\mathbf{H}^T)_i = \sum_j \mathbf{V}_{ij}$ using reduce-scatter across proc row to achieve row-wise distribution of $(\mathbf{A}\mathbf{H}^T)_i$

▷ p_{ij} owns $m/p \times k$ submatrix $((\mathbf{A}\mathbf{H}^T)_i)_j$

p_{ij} computes $(\mathbf{W}_i)_j \leftarrow \text{SolveBPP}(\mathbf{H}\mathbf{H}^T, ((\mathbf{A}\mathbf{H}^T)_i)_j)$

/* Compute H given W */

p_{ij} computes $\mathbf{X}_{ij} = (\mathbf{W}_i)_j^T (\mathbf{W}_i)_j$

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

p_{ij} collects $\mathbf{W}_i$ using all-gather across proc rows

p_{ij} computes $\mathbf{Y}_{ij} = \mathbf{W}_i^T \mathbf{A}_{ij}$

▷ $\mathbf{Y}_{ij}$ is $k \times n/p_c$

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 $((\mathbf{W}^T\mathbf{A})^j)^i$

p_{ij} computes $((\mathbf{H}^j)^i)^T \leftarrow \text{SolveBPP}(\mathbf{W}^T\mathbf{W}, (((\mathbf{W}^T\mathbf{A})^j)^i)^T)$

Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geqslant 0, \tilde{\mathbf{H}} \geqslant 0}{\text{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$

Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

Algorithm 8: $[\mathbf{W}, \mathbf{H}] = \text{HPC-NMF}\ (\mathbf{A}, k)$

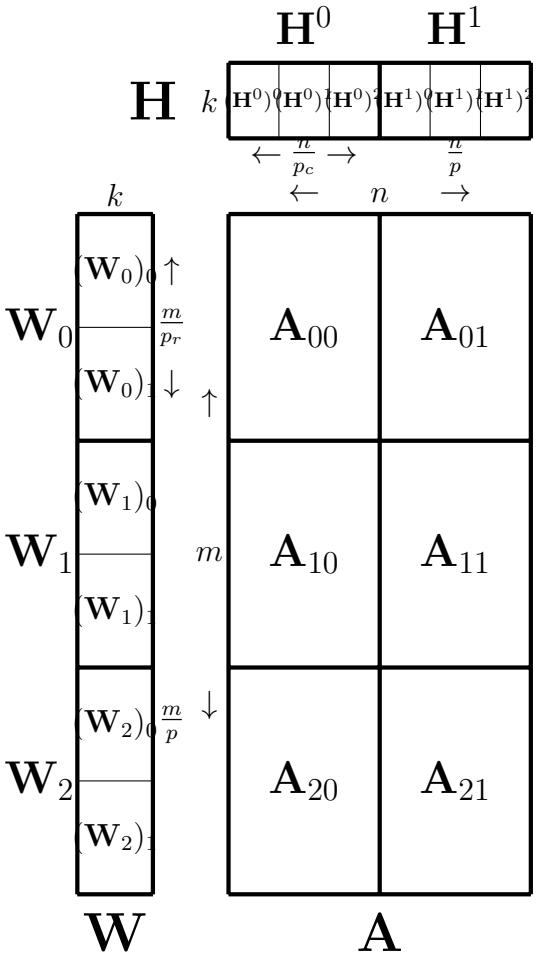


Figure 23: Distribution of matrices for HPC-NMF (Algorithm 8), for $p_r = 3$ and $p_c = 2$. Note that $\mathbf{A}_{ij}$ is $m/p_r \times m/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$ is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}_j)_i$ is $k \times n/p$.

Computation Cost Local matrix computations occur at lines 3, 6, 9, and 12. In the case that $\mathbf{A}$ 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 $\mathbf{A}$ 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 nonnegative 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 5.1.2 for these collectives. The communication cost of the all-reduces (lines 4 and 10) is $\alpha \cdot 4 \log p + \beta \cdot 2k^2$; 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 $\text{nnz}(\mathbf{A}_{ij})$ 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 $((\mathbf{A}\mathbf{H}^T)_i)_j$ 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 8 provably minimizes communication costs. Theorem 7 establishes the bandwidth cost lower bound for any algorithm that computes $\mathbf{W}^T\mathbf{A}$ or $\mathbf{A}\mathbf{H}^T$ each iteration. A latency lower bound of $\Omega(\log p)$ exists in our communication model for any algorithm that aggregates global information [6], 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 7 ([12])**.** *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 $\mathbf{A}\mathbf{H}^T$ must communicate at least $\Omega(\min\{\sqrt{mnk^2/p}, nk\})$ words along its critical path.*

Proof. The proof follows directly from [12, Section II.B]. Each matrix multiplication $\mathbf{W}^T\mathbf{A}$ and $\mathbf{A}\mathbf{H}^T$ 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 8 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 $\mathbf{A}$ [2]. 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) [25].

5.5 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, 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 [17, 39, 37]. 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.

5.5.1 Experimental Setup

5.5.1.1 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 $\times$ 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 $\times$ 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 [15], 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 [28]. 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 $\times$ 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. [53]. 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.

5.5.1.2 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.

5.5.1.3 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 [49]. 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 [54]. We use Armadillo's own implementation of sparse matrix-dense matrix multiplication, the default GNU C++ Compiler and MPI library on Edison.

5.5.1.4 *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 chapter, we used number of iterations as the stopping criteria for all the algorithms. For fair comparison, all the algorithms were executed for 10 iterations.

5.5.2 Algorithms

For each of our data sets, we benchmark and compare three algorithms: (1) Algorithm 7, (2) Algorithm 8 with $p_r = p$ and $p_c = 1$ (1D processor grid), and (3) Algorithm 8 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 5.4: the performance of a naive parallelization of Naive (Algorithm 7) will be severely limited by communication overheads, and the correct choice of processor grid within Algorithm 8 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 [37], [39], [17], [56] and [14]. We stress that Hadoop is not designed for high performance computing of iterative numerical algorithms, requiring disk I/O between steps, so

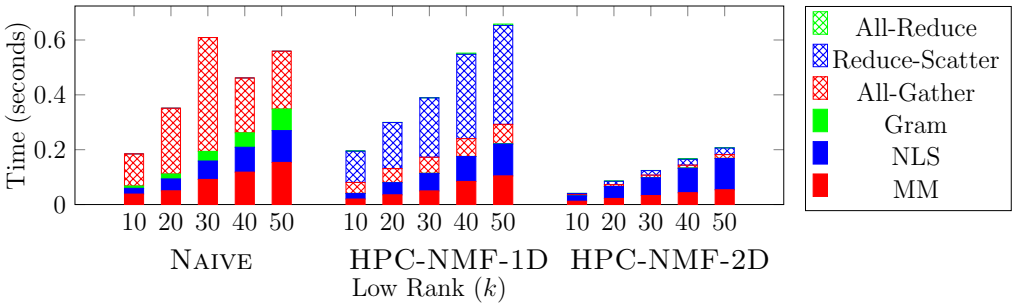
a run time comparison between a Hadoop implementation and a C++/MPI implementation is not a fair comparison of parallel algorithms. A qualitative example of differences in run time is that a Hadoop implementation of the MU algorithm on a large sparse matrix of dimension $2^{17} \times 2^{16}$ with 2×10^8 nonzeros (with k=8) takes on the order of 50 minutes per iteration [39], 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.

5.5.3 Time Breakdown

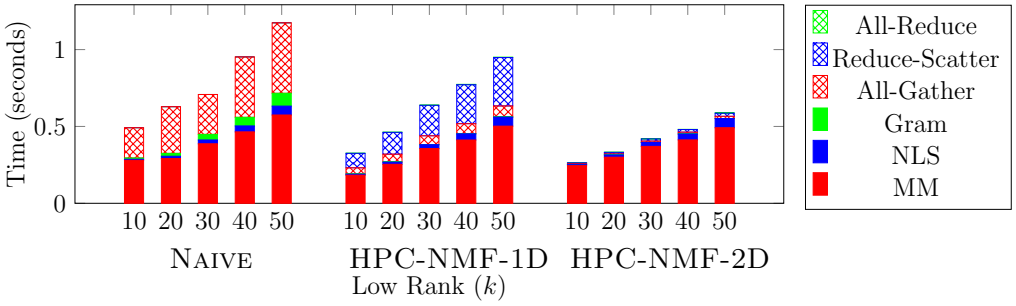
To differentiate the computation and communication costs among the algorithms, we present the time breakdown among the various tasks within the algorithms for both performance experiments. For Algorithm 8, there are three local computation tasks and three communication tasks to compute each of the factor 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.

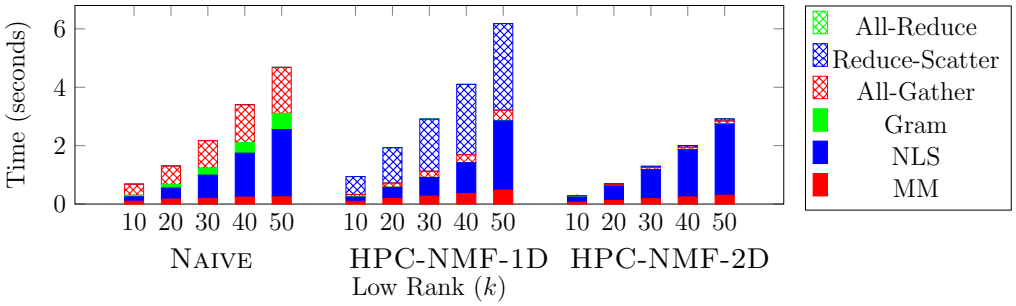
In our results, we do not distinguish the costs of these tasks for $\mathbf{W}$ and $\mathbf{H}$ separately; we report their sum, though we note that we do not always expect balance between the two contributions for each task. Algorithm 7 performs all of these tasks except Reduce-Scatter and All-Reduce; all of its communication is in All-Gather.



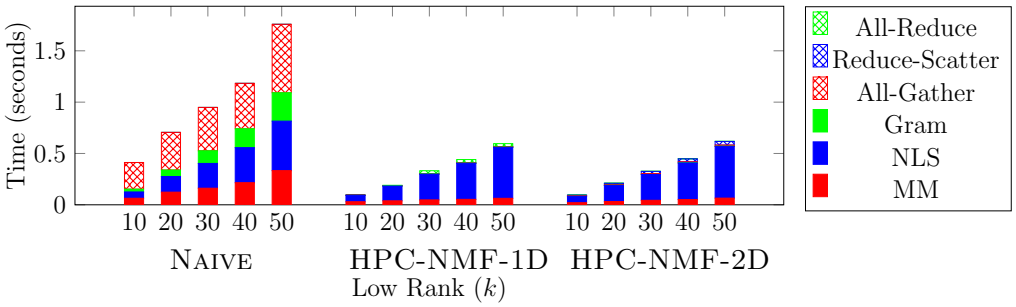
(a) Sparse Synthetic (SSyn) Comparison



(b) Dense Synthetic (DSyn) Comparison

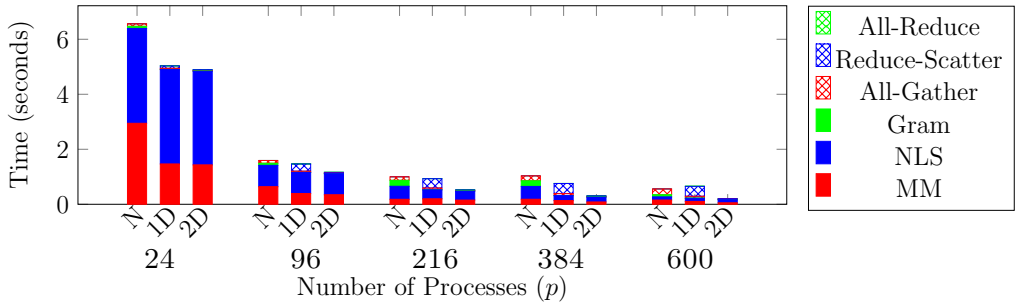


(c) Webbase Comparison

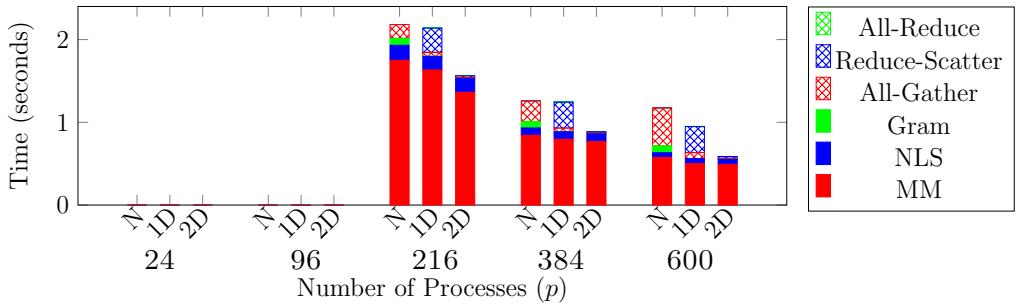


(d) Video Comparison

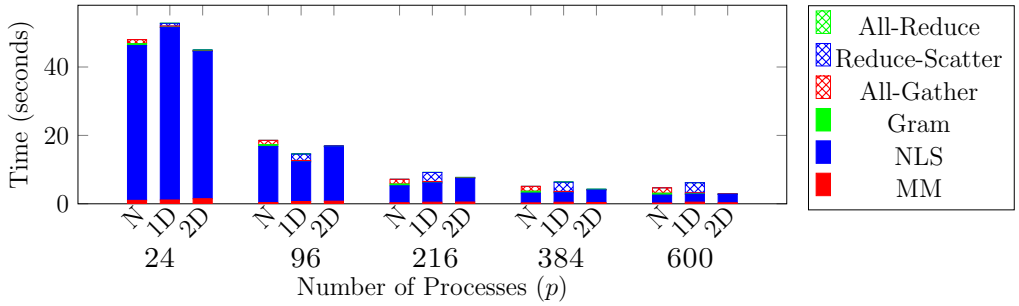
Figure 24: Comparison experiments on sparse and dense data sets for three algorithms: Naive (N), HPC-NMF-1D (1D), HPC-NMF-2D (2D). We vary the low rank k for fixed $p = 600$. The reported time is the average over 10 iterations.



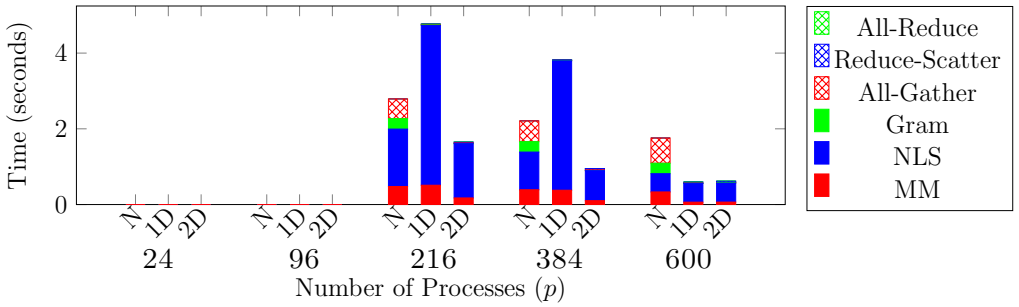
(a) Sparse Synthetic (SSyn) Strong Scaling



(b) Dense Synthetic (DSyn) Strong Scaling

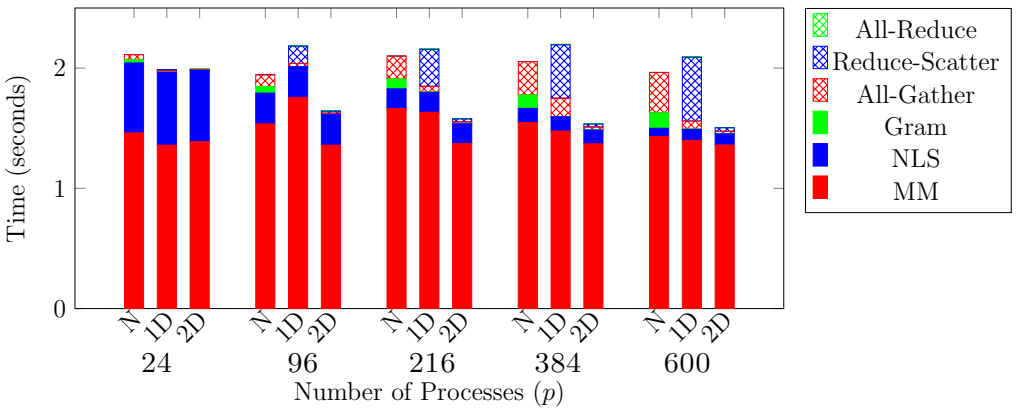


(c) Webbase Strong Scaling

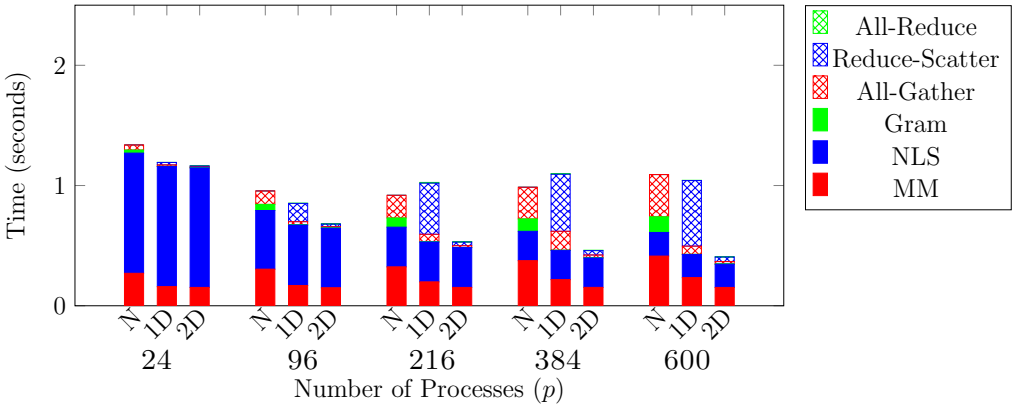


(d) Video Strong Scaling ,

Figure 25: Strong-scaling experiments on sparse and dense data sets for three algorithms: Naive (N), HPC-NMF-1D (1D), HPC-NMF-2D (2D). We vary the number of processes (cores) p for fixed $k = 50$. The reported time is the average over 10 iterations.



(a) Dense Synthetic (DSyn) Weak Scaling



(a) Sparse Synthetic (SSyn) Weak Scaling

Figure 26: Weak-scaling experiments on dense (top) and sparse (bottom) 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.

5.5.4 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 25 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; 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 24d, 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 24c 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 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 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.

Table 10: Average per-iteration running times (in seconds) of parallel NMF algorithms for $k = 50$.

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

5.5.5 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 25 shows the scaling results for all four data sets, and

Table 10 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 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.

5.5.6 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 26 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 26: in the dense case, since most of the time is in MM, the time generally holds steady as the 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 9): 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 and HPC-NMF -1D, we see that communication costs increase as we scale up. Again, Table 9 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 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.

In this chapter, we proposed 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 [30]. 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.

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.

CHAPTER VI

CONCLUSION AND FUTURE WORKS

In this thesis, we looked at a novel constrained low rank approximation called Bounded Matrix Low Rank Approximation (BMA) which imposes a lower and an upper bound on every element of the lower rank matrix. For very large input matrices, we extended our BMA algorithm to Block BMA that can scale to a large number of processors. In applications, such as HD video, where the input matrix to be factored is extremely large, distributed computation is inevitable and the network communication becomes a major performance bottleneck. To overcome the communication overhead in non-negative constrained low rank approximation, we proposed a novel distributed Communication Avoiding NMF (CANMF) algorithm that communicates only the right low rank factor to its neighboring machine. Finally, a general distributed HPC-NMF framework that uses HPC techniques in communication intensive NMF operations and suitable for broader class of NMF algorithms.

6.1 Future Works

To explain some of our future directions, we would like to summarize our AU-NMF algorithm in this chapter again. The details of this can be found in Chapter 2. We restrict attention to the class of NMF algorithms that use the Gram matrix – a matrix that is formed by the inner products of the individual vectors; 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 9.

In the above problem, computing the dense sparse multiplications $\mathbf{AH}^T$ in line 3 and $\mathbf{W}^T\mathbf{A}$ in line 4 are computationally expensive steps. Towards this end we paid our attention to accelerate this multiplication and try to experiment this for NMF.

input : $\mathbf{A}$ is an $m \times n$ matrix, k is rank of approximation
output: $\mathbf{W} \in \mathbb{R}_+^{m \times k}, \mathbf{H} \in \mathbb{R}_+^{k \times n}$
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 $\mathbf{H}\mathbf{H}^T$ and $\mathbf{A}\mathbf{H}^T$;
4 Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$;

Algorithm 9: $[\mathbf{W}, \mathbf{H}] = \text{AU-NMF}(A, k)$

We called such an NMF as Distributed Accelerated NMF.

In some cases, the sparse representation of the very large input matrix takes very less memory compared to the low rank factors. Hence for sufficiently very large sparse matrices, we can distribute the matrix between the accelerator and the cpu to peak the computational resource. We designed a load balanced distributed accelerated NMF that distributes the NMF computation between the accelerators such as Xeon Phi or GPU and the CPU.

6.1.1 Future Work - Distributed Accelerated NMF

The shared memory systems such as a multi-core systems is improving its computing ability significantly year-on-year. A multi-core processor is a single computing component with two or more independent high end central processing units (called cores). Recently for scientific application, there is a commercial low-cost accelerator available called Many Integrated Core (MIC)[1].The energy efficient MIC, packs many small end processors such as the atom processor in one CPU. It is a highly scalable architecture that is easy to program as they use the similar instruction set of existing computers. Hence many core are suitable for highly scalable parallel algorithms like Block Principal Pivoting (BPP). This many core accelerator can also work in tandem with the main multi core CPU as well. Currently, we are experimenting distributing

[1]The commercial name for this accelerator is called Xeon Phi

the BPP by performing the computationally expensive Sparse-Dense matrix multiplications (AH and AtW) in both CPU and many core accelerator simultaneously. Alternatively, we can also use traditional accelerators such as GPU. We are using the following strategies for distributing the sparse-dense multiplication.

- Static Partitioning - Statically Partition the sparse and dense matrix to accelerator and the CPU. The partitions can be row and column partitions.
- Dynamic Partitioning - In the case of static partitioning, discovering a partition, balancing the load between CPU and accelerator that is suitable for all general matrices can be difficult. To over come this issue, a synchronized computation between accelerator and CPU can significantly boost the load balancing. That is., before computing the multiplication of a partition, accelerator will wait for a signal from CPU. CPU will decide whether a block of matrix multiplication will be performed by the accelerator or CPU during runtime.
- Model driven - Dynamic partitioning requires communication for synchronization between the CPU and accelerator. This can be avoided, by determining a model driven partitioning between CPU and accelerator. The model can decide on a partition based on various parameters such as the size, density, nnz etc., of the sparse matrix and the some properties of the dense matrix.
- Adaptive - It is an extension of model driven partitioning. In iterative algorithm, we have the scope for improving the load balance over the progress of iteration. For eg., if the previous iteration poorly load balanced, we can take corrective action and determine a suitable partition in the next iteration.

We will investigate the different partition strategies for sparse dense matrix multiplication to understand the load distribution between CPU and accelerator to peak the total compute capacity in a computer. We will compare the above partition strategies between two different accelerator - Xeon Phi and GPU.

6.1.2 Other Constraints

We are currently exploring enforcing integer constraints on individual low rank factors $\mathbf{W}$ and $\mathbf{H}$ as well as on the product $\mathbf{WH}$. Formally the former problem can be explained as in Equation (44) and the latter through Equation (46).

$$\underset{\mathbf{W}\in\mathbb{Z},\mathbf{H}\in\mathbb{Z}}{\operatorname{argmin}} \|\mathbf{M}\cdot*(\mathbf{A}-\mathbf{WH})\|_F^2 \tag{44}$$

Similar to 2 Block BCD ANLS, we can solve the problem (44) as two different subproblems of finding integer low rank factors $\mathbf{W}$ given $\mathbf{H}$ and vice-versa as explained below.

$$\begin{aligned}\mathbf{W} &\leftarrow \operatorname{argmin}_{\mathbf{W}\in\mathbb{Z}}\|\mathbf{H}^\intercal\mathbf{W}^\intercal-\mathbf{A}^\intercal\|_F^2,\\ \mathbf{H} &\leftarrow \operatorname{argmin}_{\mathbf{H}\in\mathbb{Z}}\|\mathbf{WH}-\mathbf{A}\|_F^2,\end{aligned} \tag{45}$$

In this case, every subproblem is an Integer Least Squares problem comprising of two steps - Reduction and Search. We have completed reduction for multiple right hand side but the computationally expensive search problem is still open. We are leveraging the insight that in the case of iterative alternating integer least squares formulation, the search process can bootstrap from the previous iteration. This will reduce the search time significantly.

The latter problem of enforcing integer constraints on the product $\mathbf{WH}$ is relatively computationally inexpensive. However, the major challenge is holding the big matrix $\mathbf{Z}$ in memory and this restricts the practical applicability of solution for very large input matrices. Alternatively, we can consider $\mathbf{Z}$ as a logical place holder that can be computed whenever needed. But this will increase the computational cost drastically slowing down the algorithm.

$$\underset{\mathbf{W}\in\mathbb{R}^{m\times k},\mathbf{H}\in\mathbb{R}^{k\times n},\mathbf{Z}\in\mathbb{Z}^{m\times n}}{\operatorname{argmin}} \|\mathbf{M}.*(\mathbf{A}-\mathbf{Z})\|_F^2+\|\mathbf{Z}-\mathbf{WH}\|_F^2 \tag{46}$$

We are currently exploring the integer constraint in both the directions.

6.1.3 Open Source Software

The current state-of-the-art approaches for NMF with large-scale data have primarily focused on the Map-Reduce programming model with implementations in systems like Hadoop and Spark. These implementations are much too slow because (a) map-reduce are not ideal for all operations in the NMF algorithm involving communication intensive global, input-data shuffles across machines (b) sometimes they perform more data movement than necessary and because each step of computation involves reading and writing data from disk and (c) to run for very large matrices we need sophisticated clusters and sometime machine learning experts for parameter tuning these algorithms. To address these problems, we focus our attention for high-performance NMF algorithms explained in Chapter 5. 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. In particular, our proposed framework ensures that after the input data is initially read into memory, it is never communicated to any other machine in the network. We communicate only the factor matrices and other smaller temporary matrices, which guarantees privacy of the original data and, in some cases, communication optimality. 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, to perform very few messages per iteration, the minimum achievable for aggregating global data.

As explained in Chapter 5, currently we have implemented the NMF algorithm based on ANLS-BPP and have witnessed tremendous improvement for very large matrices. For example, a state-of-the-art implementation of MU [39] using a Hadoop system takes 50 minutes per iteration whereas our algorithm takes less than a second

on a cluster with 24 machines. In addition to implementing and comparing the aforementioned standard ANLS algorithms at large scale, we would also like to investigate alternative NMF methods based on hierarchical factorizations, which may be more appropriate for certain applications like topic modeling.

Also, our framework HPC-NMF based on AU-NMF, is suitable for implementing the iterative NMF algorithms such as Hierarchical Alternating Least Squares (HALS), Multiplicative Update (MU), Projected Gradient methods, and active set algorithms such as Block Principal Pivoting (BPP). Thus, the framework allows the end users to try many different NMF algorithms, all implemented efficiently, on very large real-world data matrices such as those arising in recommender systems, social network analyis, text mining, etc.

We are currently in the process of developing an open source NMF and NTF library with objective of reusability, extensibility and off the shelf algorithms for conducting large scale experiments. The entire software stack is shown in the Figure 27. The major components in the library are the Matrix/Tensor library and BLAS/LAPACK library. In this thesis, we used Armadillo [49] as matrix library and OpenBLAS [54] for the BLAS and LAPACK operations. Armadillo is a C++ library having interfaces for both sparse and dense matrices. It allows users to perform matrix operations similar to Matlab. That is., $\mathbf{W} * \mathbf{H}$ will multiply matrix $\mathbf{H}$ with $\mathbf{W}$ by calling the *gemm* in BLAS. The one major short coming of OpenBLAS is unavailability of sparse-dense BLAS operations. Under such cases, either we used baseline implementation from Armadillo or used Intel MKL.

We would like to extend the algorithms for NMF to Non-negative Tensor Factorization (NTF). Tensors are generalization of matrices, representing data sets with more than two dimensions. The canonical decomposition (CANDECOMP) or the parallel factorization (PARAFAC), is one of the natural extensions of the singular value decomposition to higher order tensors. The CP decomposition with nonnegativity

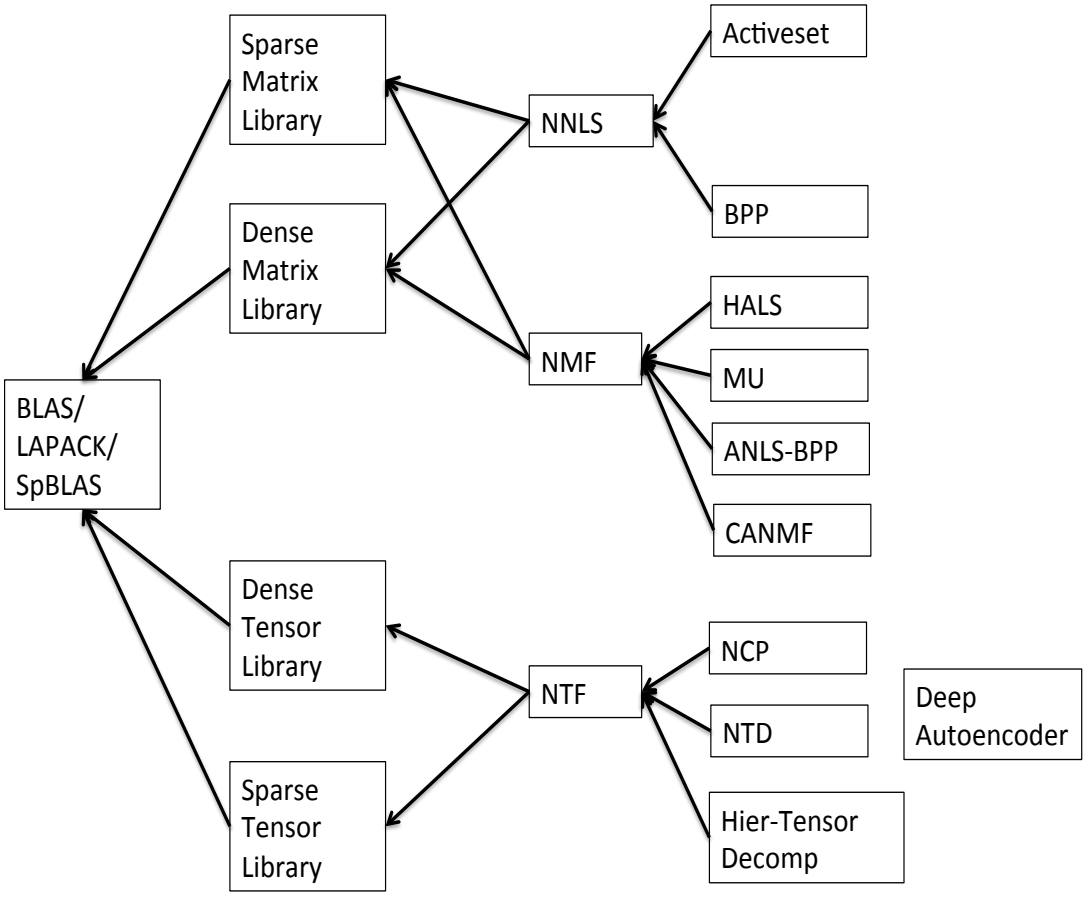


Figure 27: Constrained Low Rank Approximation Software Stack

constraints imposed on the factor matrices is denoted as nonnegative CP (NCP), can be computed in a way that is similar to the NMF computation. In the case of Non-negative Tucker Decomposition (NTD), also called as Tucker3 model, apart from the factor matrices, we also determine a core tensor that defines the scaling factor for the rank-k approximate tensor. Cichocki et.al., [10], discuss the formal definition and the details of the various NTF algorithms. By respecting the multi-way relationships among data points and treating the data as a tensor, we can often compute much better approximations, but maintaining the efficiency of algorithms is more difficult. In particular, optimizing data distributions and data movement becomes more complicated and more important for high performance than in the matrix case.

In the case of shared memory implementation, we were using OpenMP for scalable implementations. For, distributed implementations, we are now surveying the right libraries for intended algorithms. We will investigate and develop an open source library for scalable and distributed constrained low rank approximation problems with special focus on non-negativity constraint. Finally, we would like to package all our MPI-based distributed HPC-NMF and HPC-NTF algorithms into an open-source library, basing our implementations on other open-source libraries like BLAS and LAPACK. We have to carefully choose the distributed matrix and tensor libraries that are easy to extend by any developers and can run seamlessly on varied platforms. We will implement our package so that it can run on systems ranging from supercomputers to commodity clusters or workstations and always provide high efficiency.

REFERENCES

[1] "Movielens dataset." `http://movielens.umn.edu`, 1999. [Online; accessed 6-June-2012].

[2] Ballard, G., Druinsky, A., Knight, N., and Schwartz, O., "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication," in *Proceedings of SPAA*, pp. 86–88, 2015.

[3] Bertsekas, D. P., *Nonlinear Programming*. Belmont, MA: Athena Scientific, 1999.

[4] Boldi, P. and Vigna, S., "The webgraph framework i: Compression techniques," in *Proceedings of the*, WWW '04, (New York, NY, USA), pp. 595–602, 2004.

[5] Brozovsky, L. and Petricek, V., "Recommender system for online dating service," in *Proceedings of Conference Znalosti 2007*, (Ostrava), VSB, 2007.

[6] Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R., "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[7] Chen, D. and Plemmons, R., "Nonnegativity constraints in numerical analysis," in *Proc. of Symposium on the Birth of Numerical Analysis*, p. 3, World Scientific, 2009.

[8] Cichocki, A. and Phan, A.-H., "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E92-A, pp. 708–721, 2009.

[9] Cichocki, A., Zdunek, R., and Amari, S., "Hierarchical als algorithms for nonnegative matrix and 3d tensor factorization," *Lecture Notes in Computer Science*, vol. 4666, pp. 169–176, 2007.

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

[11] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, pp. 391–407, 1990.

[12] Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., and Spillinger, O., "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proceedings of IPDPS*, pp. 261–272, 2013.

[13] Fairbanks, J. P., Kannan, R., Park, H., and Bader, D. A., "Behavioral clusters in dynamic graphs," *Parallel Computing*, vol. 47, pp. 38–50, 2015.

[14] Faloutsos, C., Beutel, A., Xing, E. P., Papalexakis, E. E., Kumar, A., and Talukdar, P. P., "Flexi-fact: Scalable flexible factorization of coupled tensors on hadoop," in *Proceedings of the SDM*, pp. 109–117, 2014.

[15] Fujimoto, R., Guin, A., Hunter, M., Park, H., Kanitkar, G., Kannan, R., Milholen, M., Neal, S., and Pecher, P., "A dynamic data driven application system for vehicle tracking," *Procedia Computer Science*, vol. 29, pp. 1203–1215, 2014.

[16] Funk, S., "Stochastic gradient descent." `http://sifter.org/~simon/journal/20061211.html`, 2006. [Online; accessed 6-June-2012].

[17] Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y., "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the KDD*, pp. 69–77, ACM, 2011.

[18] Goldberg, K., "Jester collaborative filtering dataset." `http://goldberg.berkeley.edu/jester-data/`, 2003. [Online; accessed 6-June-2012].

[19] Golub, G. H. and Van Loan, C. F., *Matrix Computations*. The Johns Hopkins University Press, 3rd ed., 1996.

[20] Grippo, L. and Sciandrone, M., "On the convergence of the block nonlinear gauss-seidel method under convex constraints," *Oper. Res. Lett.*, vol. 26, pp. 127–136, Apr. 2000.

[21] Grove, D., Milthorpe, J., and Tardieu, O., "Supporting array programming in X10," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pp. 38:38–38:43, 2014.

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

[23] Kannan, R., Ishteva, M., and Park, H., "Bounded matrix low rank approximation," in *Proceedings of the 12th IEEE International Conference on Data Mining(ICDM-2012)*, pp. 319–328, 2012.

[24] Kannan, R., Ishteva, M., and Park, H., "Bounded matrix factorization for recommender system," *Knowledge and Information Systems*, vol. 39, no. 3, pp. 491–511, 2014.

[25] Kaya, O. and Uçar, B., "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of SC*, pp. 77:1–77:11, ACM, 2015.

[26] Kim, H. and Park, H., "Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 2, pp. 713–730, 2008.

[27] Kim, J., He, Y., and Park, H., "Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework," *Journal of Global Optimization*, pp. 1–35, 2013.

[28] Kim, J., He, Y., and Park, H., "Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework," *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, 2014.

[29] Kim, J. and Park, H., "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011.

[30] Kim, J. and Park, H., "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011.

[31] Korattikara, A., Boyles, L., Welling, M., Kim, J., and Park, H., "Statistical optimization of non-negative matrix factorization," in *Proceedings of AISTATS, JMLR: W &CP*, vol. 15, pp. 128–136, 2011.

[32] Koren, Y., "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '08*, pp. 426–434, 2008.

[33] Koren, Y., "Collaborative filtering with temporal dynamics," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*, p. 447, 2009.

[34] Koren, Y., Bell, R., and Volinsky, C., "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, pp. 30–37, Aug. 2009.

[35] Kyrola, A., Blelloch, G., and Guestrin, C., "Graphchi: large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[36] Lee, D. D. and Seung, H. S., "Learning the parts of objects by non-negative matrix factorization.," *Nature*, vol. 401, pp. 788–791, Oct. 1999.

[37] Liao, R., Zhang, Y., Guan, J., and Zhou, S., "Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets," *Genomics, proteomics & bioinformatics*, vol. 12, no. 1, pp. 48–51, 2014.

[38] Lin, C. J., "Projected Gradient Methods for Nonnegative Matrix Factorization," *Neural Comput.*, vol. 19, pp. 2756–2779, Oct. 2007.

[39] Liu, C., Yang, H.-c., Fan, J., He, L.-W., and Wang, Y.-M., "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce," in *Proceedings of the WWW*, pp. 681–690, ACM, 2010.

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

[41] Luk, F. T. and Park, H., "On parallel jacobi orderings," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 18–26, 1989.

[42] Luk, F. T. and Park, H., "A proof of convergence for two parallel jacobi svd algorithms," *Computers, IEEE Transactions on*, vol. 38, no. 6, pp. 806–811, 1989.

[43] Mackey, L. W., Weiss, D., and Jordan, M. I., "Mixed membership matrix factorization," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 711–718, 2010.

[44] Markovsky, I., "Algorithms and literate programs for weighted low-rank approximation with missing data," in *Approximation Algorithms for Complex Systems* (Levesley, J., Iske, A., and Georgoulis, E., eds.), pp. 255–273, Springer-Verlag, 2011. Chapter: 12.

[45] Mejía-Roa, E., Tabas-Madrid, D., Setoain, J., García, C., Tirado, F., and Pascual-Montano, A., "NMF-mGPU: non-negative matrix factorization on multi-GPU systems," *BMC bioinformatics*, vol. 16, no. 1, p. 43, 2015.

[46] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., and Talwalkar, A., "MLlib: Machine Learning in Apache Spark," May 2015.

[47] Paterek, A., "Improving regularized singular value decomposition for collaborative filtering," in *Proceedings of 13th ACM International Conference on Knowledge Discovery and Data Mining - KDD'07*, pp. 39–42, 2007.

[48] Salakhutdinov, R. and Mnih, A., "Bayesian probabilistic matrix factorization using markov chain monte carlo," in *ICML*, pp. 880–887, 2008.

[49] Sanderson, C., "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," tech. rep., NICTA, 2010.

[50] Seung, D. and Lee, L., "Algorithms for non-negative matrix factorization," *NIPS*, vol. 13, pp. 556–562, 2001.

[51] Thakur, R., Rabenseifner, R., and Gropp, W., "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[52] Wang, Y.-X. and Zhang, Y.-J., "Nonnegative matrix factorization: A comprehensive review," *TKDE*, vol. 25, pp. 1336–1353, June 2013.

[53] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009.

[54] Xianyi, Z., "Openblas," Last Accessed 03-Dec-2015.

[55] Xiong, L., Chen, X., Huang, T.-K., Schneider, J. G., and Carbonell, J. G., "Temporal collaborative filtering with bayesian probabilistic tensor factorization," in *Proceedings of the SIAM International Conference on Data Mining-SDM'10*, pp. 211–222, 2010.

[56] Yin, J., Gao, L., and Zhang, Z., "Scalable nonnegative matrix factorization with block-wise updates," in *Machine Learning and Knowledge Discovery in Databases*, vol. 8726 of *LNCS*, pp. 337–352, 2014.

[57] Yu, H.-F., Hsieh, C.-J., Si, S., and Dhillon, I. S., "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Proceedings of the IEEE International Conference on Data Mining-ICDM'12*, pp. 765–774, 2012.

[58] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pp. 10–10, USENIX Association, 2010.

[59] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R., "Large-scale Parallel Collaborative Filtering for the Netflix Prize," *Algorithmic Aspects in Information and Management*, vol. 5034, pp. 337–348, 2008.

[60] Ziegler, C.-N., McNee, S. M., Konstan, J. A., and Lausen, G., "Improving recommendation lists through topic diversification," in *Proceedings of the 14th international conference on World Wide Web-WWW'05*, pp. 22–32, 2005.