SCALABLE DATA MINING VIA CONSTRAINED LOW RANK APPROXIMATION

A Dissertation
Presented to
The Academic Faculty

By

Srinivas Eswar

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

Georgia Institute of Technology

August 2022

SCALABLE DATA MINING VIA CONSTRAINED LOW RANK APPROXIMATION

Thesis committee:

Dr. Richard W. Vuduc, Advisor
Computational Science and Engineering
Georgia Institute of Technology

Dr. Haesun Park, Advisor
Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
Computational Science and Engineering
Georgia Institute of Technology

Dr. Edmond T. Chow
Computational Science and Engineering
Georgia Institute of Technology

Dr. Grey M. Ballard
Department of Computer Science
Wake Forest University

Date approved: July 1, 2022

Never confuse the unusual with the impossible.

Psmith

For Amma and Appa.

ACKNOWLEDGMENTS

I thank Rich Vuduc and Haesun Park for the many years of support, encouragement, patience, and advice. I am immensely fortunate to have been advised by both of these brilliant people. You gave me the freedom to explore many different interests while always being present to overcome any setbacks.

Thank you to my committee members Grey Ballard, Ümit Çatalyürek, and Edmond Chow for your time, insights, comments, and help in making this dissertation possible. I am extremely lucky to follow the research trail set by Ramki Kannan. Thank you, Grey and Ramki for being my mentors since the start of my Ph.D. studies and for blazing the way.

Among the other faculty at Georgia Tech, I would like to thank Polo Chau and Jimeng Sun for serving on my qualifying committee. Special thanks to Jeff Young and Oden Green for helping me navigate the many twists and turns of Ph.D. life.

Thank you to my fellow GT friends, labmates, and colleagues for our amazing time together. The CSE floor and the HPC Garage has been my haunt for the better part of the past decade, and it will always be filled with fun memories. Thank you, Mikhail Isaev, Patrick Lavin, Chunxing Yin, Piyush Sao, Jiajia Li, Koby Hayashi, Ben Cobb, Dongjin Choi, Sara Karamati, Wafa Louhichi, Hannah Kim, and Rundong Du for making me want to show up to the lab. Cheers to the S(h)rivastava group (Ankit and Harsh), Patrick Flick, Elias Khalil, Kasimir Gabert, Jing An, Saurabh Sawlani, Rahul Nihalani, Rob Chen, Aftab Patel, Apo Yasar, Shruti Shivakumar, and Majid Farhadi for making me venture through the rest of the school! I hope our paths cross again in the future.

I am deeply grateful to my parents, sister Sharanya, and wife Sneha for their unconditional love and support. Your constant encouragement, despite the distance, has kept me afloat. Finally, I would like to thank my roommates and dear friends, Prashant Nair and Prasun Gera. You have been my second family throughout this journey. Thank you for showing me the ropes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ADMM	Alternating Direction Method of Multipliers
ANLS	Alternating Nonnegative Least-Squares
BCD	Block Coordinate Descent
BPP	Block Principal Pivoting
CLRA	Constrained Low Rank Approximation
CP	CANDECOMP/PARAFAC
GNCG	Gauss-Newton method using Conjugate Gradients
HALS	Hierarchical Alternating Least-Squares
i.i.d.	independently and identically distributed
ID	Interpolative Decomposition
JointNMF	Joint Nonnegative Matrix Factorisation
KKT	Karush-Kuhn-Tucker
KRP	Khatri-Rao Product
LRA	Low Rank Approximation
MPI	Message Passing Interface
MTTKRP	Matricised-Tensor Times Khatri-Rao Product
MU	Multiplicative Updating
NCP	Nonnegative CANDECOMP/PARAFAC
NLS	Nonnegative Least-Squares
NMF	Nonnegative Matrix Factorisation
NTF	Nonnegative Tensor Factorisation
PCA	Principal Components Analysis

PGD	Projected Gradient Descent
PGNCG	Projected Gauss-Newton method using Conjugate Gradients
PLANC	Parallel Low rank Approximation with Nonnegativity Constraints
SVD	Singular Value Decomposition
SymNMF	Symmetric Nonnegative Matrix Factorisation

SUMMARY

Matrix and tensor approximation methods are recognised as foundational tools for modern data analytics. Their strength lies in their long history of rigorous and principled theoretical foundations, judicious formulations via various constraints, along with the availability of fast computer programs. Multiple Constrained Low Rank Approximation (CLRA) formulations exist for various commonly encountered tasks like clustering, dimensionality reduction, anomaly detection, amongst others. The primary challenge in modern data analytics is the sheer volume of data to be analysed, often requiring multiple machines to just hold the dataset in memory. This dissertation presents CLRA as a key enabler of *scalable* data mining in *distributed-memory* parallel machines.

Nonnegative Matrix Factorisation (NMF) is the primary CLRA method studied in this dissertation. NMF imposes nonnegativity constraints on the factor matrices and is a well-studied formulation known for its simplicity, interpretability, and clustering prowess. The major bottleneck in most NMF algorithms is a distributed matrix-multiplication kernel. We develop the Parallel Low rank Approximation with Nonnegativity Constraints (PLANC) software package, building on the earlier MPI-FAUN library, which includes an efficient matrix-multiplication kernel tailored to the CLRA case. It employs carefully designed parallel algorithms and data distributions to avoid unnecessary computation and communication.

We extend PLANC to include several optimised Nonnegative Least-Squares (NLS) solvers and symmetric constraints, effectively employing the optimised matrix-multiplication kernel. We develop a parallel inexact Gauss-Newton algorithm for Symmetric Nonnegative Matrix Factorisation (SymNMF). In particular PLANC is able to efficiently utilise second-order information when imposing symmetry constraints without incurring the prohibitive memory and computational costs associated with these methods. We are able to observe 70% efficiency while scaling up these methods.

We develop new parallel algorithms for fusing and analysing data with multiple modalities in the Joint Nonnegative Matrix Factorisation (JointNMF) context. JointNMF is capable of knowledge discovery when both *feature-data* and *data-data* information is present in a data source. We extend PLANC to handle this case of simultaneously approximating two different large input matrices and study the various trade-offs encountered in the bottleneck matrix-multiplication kernel.

We show that these ideas translate naturally to the multilinear setting when data is presented in the form of a tensor. A bottleneck computation analogous to the matrix-multiply, the Matricised-Tensor Times Khatri-Rao Product (MTTKRP) kernel, is implemented. We conclude by describing some avenues for future research which extend the work and ideas in this dissertation. In particular, we consider the notion of structured sparsity, where the user has some control over the nonzero pattern, which appears in computations for various tasks like cross-validation, working with missing values, robust CLRA models, and in the semi-supervised setting.

CHAPTER 1

LOW RANK APPROXIMATION AS THE WORKHORSE OF DATA MINING

This dissertation shows that Constrained Low Rank Approximation (CLRA) can be used to *efficiently* extract information from *massive* and *complex* datasets. To achieve efficiency and scalability, parallel algorithms for traditional CLRA methods need to be redesigned, and practical implementations need to be adapted to match the architecture of modern computing platforms.

As a preview of what we will demonstrate, consider the problem of Nonnegative Matrix Factorisation (NMF), an example of a CLRA (and the primary focus of this dissertation). If we characterise NMF computations in a certain way, where a certain algorithmic tuning parameter is allowed to vary, then we can obtain 40% speedup over prior state-of-the-art methods by simply adjusting this parameter as illustrated in Fig. 1.1 (details in Section 2.3). The focus of this dissertation is to systematically develop parallel versions of some popular

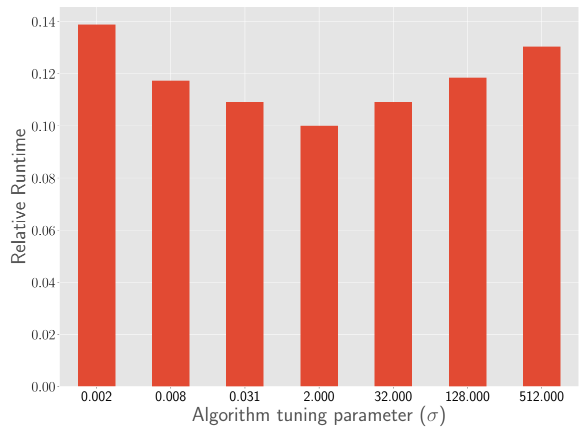


Figure 1.1: Speedups observed when running NMF under different settings. Naïve implementations usually reside in the ends of the parameter space and not near the middle.

CLRA methods. The techniques developed can be adopted to other CLRA methods like the alternating-minimisation algorithms of Udell et al. [1].

We are motivated by CLRA because of its wide applicability in fundamental problems of data mining, which is the process of collecting, cleaning, analysing, and gaining useful insights from large and complex datasets [2]. These insights take the form of interesting patterns in the dataset including but not limited to grouping the various data items (clustering), separating unusual records (anomaly detection), and summarising task-specific structures present in the dataset (embedding). Manual extraction of these patterns has been occurring for centuries (e.g., regression analysis to determine orbits from astronomical observations). However, with the advent of the Internet and the cheap availability of computers, the resulting deluge of data, often unstructured, makes it virtually impossible to manually discover these patterns [3]. While there are many ideas for how to do pattern discovery, CLRA is often the first and fastest way to start any exploratory analysis task.

Our focus on efficient parallelisation and practical implementation stems in part from the dramatic increases in the pace of data generation, as recent reports from various domains indicate [4–6]. The first is from the realm of scientific computing where breakthroughs are anticipated through the modelling and simulation of ever more complex physical phenomena and running massive experiments on larger user facilities like accelerators, colliders, light sources, and neutron sources [4]. ITER, the world's largest fusion experiment, is expected to produce two petabytes of data every day [7]. Many natural language processing models are trained on the entire Wikipedia website which is growing steadily at a rate of approximately 7% per year [5]. The English Wikipedia corpus is currently consisting of 6.4 million articles containing more than 4.9 billion words. The next example is from medical imaging where the average size of MRI, CT, or fMRI datasets have grown 3-10 times between 2011 to 2018 [6]. With advances in computer vision algorithms the research community develops expectations regarding the data samples needed to train, test, and validate these methods resulting in consistently growing benchmark datasets. Achieving work-efficiency and parallel scalability will be paramount to addressing the scale of data.

However, algorithmic innovation is not enough to achieve practical speed and scale on modern computing systems. There is a spike in the diversity of computing platforms available to researchers. To overcome power, memory, and instruction-level-parallelism limitations, hardware vendors embraced parallelism [8]. This takes the form of chips with multiple cores, wider vector processing units, or massively parallel supercomputers with tens of thousands of off-the-shelf processors. Many complications arose with this new paradigm, chief among them being the non-negligible cost of accessing data from memory or over the network. This *communication* cost is higher than that of performing a computation (a flop) with the gap widening over time. A recent study has shown that peak flops per socket has been improving at a rate of 50-60% per year, while memory bandwidth increasing at 23% per year and memory latency *increasing* at 4% per year [9]. Interconnect performance, both bandwidth and latency, are also only improving at 20% year on year. Mitigating these rising communication costs is an important factor to consider while designing algorithms in this computing landscape. Thus, adapting CLRA techniques to address this increasing gap between computation and communication costs will be necessary to be efficient in practice. We show how to do so.

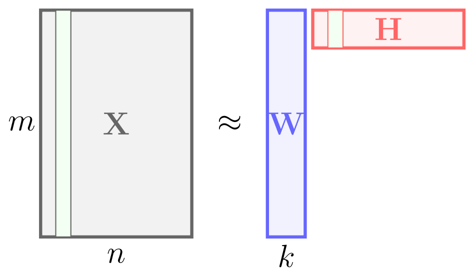


Figure 1.2: Pictorial representation of LRA. Input data items are arranged as columns of $\mathbf{X}$ and their embeddings are columns of $\mathbf{H}$.

1.1 Linear Dimensionality Reduction Methods

Low Rank Approximation (LRA) is one of the oldest approaches to extract the underlying structure within a dataset [10]. Given a set of n data points, $\mathbf{x}_i \in \mathbb{R}^m$, arranged as a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, the goal of LRA is to find a pair of factor matrices, $\mathbf{W} \in \mathbb{R}^{m \times k}$ and $\mathbf{H} \in \mathbb{R}^{k \times n}$, that well-approximate the input:

$$\mathbf{X} \approx \mathbf{W}\mathbf{H} \,. \tag{1.1}$$

The k columns of $\mathbf{W}$, denoted by $\mathbf{w}_i \in \mathbb{R}^m$ for $1 \leq i \leq k$, serve as the “basis vectors” used to linearly approximate the input data points. The i^{th} column of $\mathbf{H}$, which is $\mathbf{h}_i$, contains the contribution of these basis vectors in representing $\mathbf{x}_i$. That is,

$$\mathbf{x}_i \approx \sum_{r=1}^{k} \mathbf{w}_r h_{ri} = \mathbf{W}\mathbf{h}_i \,.$$

Thus, LRA provides a rank-k matrix approximation of $\mathbf{X}$. Typically, the number of basis vectors k is much smaller than the input dimension of the data points m and the number of samples n. Since $k \ll \min(m, n)$, LRA is able to compress mn entries of $\mathbf{X}$ to $mk + nk$ entries of $\mathbf{W}$ and $\mathbf{H}$. Further, the column vector $\mathbf{h}_i \in \mathbb{R}^k$ can also be considered as the lower dimensional representation or “embedding” of the input data point $\mathbf{x}_i$.

In order to compute an LRA we need to define an error measure between $\mathbf{X}$ and $\mathbf{W}\mathbf{H}$. A commonly used measure is the square of the Frobenius norm of the residual,

$\|\mathbf{X} - \mathbf{WH}\|_F^2 = \sum_{ij} (\mathbf{X} - \mathbf{WH})_{ij}^2$. The popular method, Principal Components Analysis (PCA), uses this error measure on an appropriately centered and scaled $\mathbf{X}$ [11]. PCA can be computed via the Singular Value Decomposition (SVD) and is closely related to the eigenvalue decomposition and other factorisations like QR, LU, and Cholesky from numerical linear algebra [12]. Using the squared Frobenius norm loss implicitly assumes independently and identically distributed (i.i.d.) Gaussian noise [1]. However, other error measures exist for different noise statistics, for example ℓ_1 loss in robust PCA [13], Kullback-Leibler divergence for count data [14], and hinge loss [1], among others.

Apart from the choice of loss functions, various constraints can also be placed on the factor matrices. These choices help in interpreting the factors found or restrict them to be meaningful with respect to the input data. For example, in PCA, the orthogonal constraints on the basis matrix make the basis vectors represent the directions of maximum variance [11]. Nonnegativity constraints on the factor matrices often generates a parts-based approximation [14]. The CUR decomposition explicitly constrains the factor matrices to be expressed in terms of a small number of actual columns and actual rows from the input matrix [15]. Therefore, LRA can be used in a principled manner for data analysis by selecting appropriate losses and constraints depending on the task at hand.

Another appealing aspect of LRA is the availability of high performance, scalable, and wide variety of computer programs for numerical linear algebra [16–20]. Dense and sparse linear algebra computations are two of the seven motifs that cover nearly all calculations carried out in scientific computing [21]. The rich history of methods developed for numerical linear algebra can be repurposed for data analysis [12]. The availability of performant code is essential with the advent of the Internet age and large data collections [3].

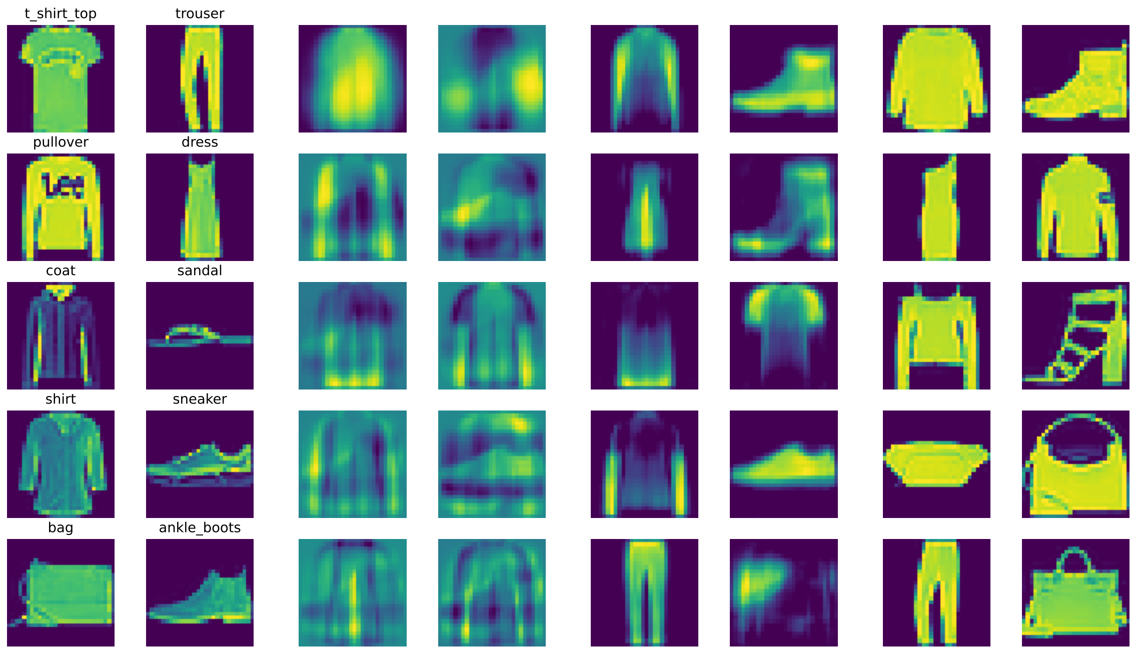
1.1.1 Applications of LRA

LRA has been used in a variety of applications in data analysis [22]. A standard use-case is to use LRA as a *feature extraction* method. PCA and NMF have been used on image data to extract features that can later be used for tasks like face recognition [14, 23]. The basis vectors in $\mathbf{W}$ are linear combinations of the features from the input data which are the representative features discovered. Looking at the columns of $\mathbf{H}$, we get a compressed representation, or *embedding*, of the input data in this new feature space. Clustering these embeddings is computationally cheaper and often produces excellent results [24–27]. A classic example of such a data analysis task is *topic modeling*, where the user aims to discover the various topics that occur in a collection of documents [28]. *Outlier detection* is another task that can be performed by LRA [29, 30]. It can be viewed as a complementary concept to that of clustering where the user aims to determine individual data points that are different from the rest. Fraud detection, network intrusion, and data preprocessing are some common applications of outlier detection.

Taken together, these tasks reveal three different facets of the structure present in datasets. Often, LRA is able to simultaneously model and discover these different aspects of the structure. Performing these disparate tasks within a single framework can speed up the analysis process. LRA methods typically utilise fewer hyperparameters and contain fewer indeterministic components in contrast to other techniques like neural networks. Thus, avoiding the need for expensive hyperparameter tuning and search [31]. In summary, LRA with its desired properties of principled and flexible design, simplicity, and wide availability of fast computer programs makes it the workhorse of data mining.

1.1.2 Some examples of LRA

Let us look at an example of LRA methods used to approximate an image dataset. We use the Fashion-MNIST collection [32], which comprises 60,000 training and 10,000 test examples of clothing articles classified into 10 categories (see Fig. 1.3a). We shall work with the test subset for this illustration. Each grayscale image consists of 784 pixels which are vectorised and placed as columns of a matrix $\mathbf{X} \in \mathbb{R}_{+}^{784\times 10{,}000}$.



(a) Cluster examples. (b) SVD basis vectors. (c) NMF basis vectors. (d) ID basis vectors.

Figure 1.3: Difference in the basis found by various LRA methods. SVD produces orthonormal columns which represent the best rank-10 approximation of the data. They are hard to interpret. The NMF and ID vectors are easier to understand. The ID bases are explicit examples from the dataset whereas the NMF vectors are found by the algorithm and may not correspond to an existing image.

We approximate $\mathbf{X}$ using three different CLRA techniques using the squared Frobenius norm as the error measure and approximation rank of $k = 10$. We look at the different basis vectors discovered, i.e., the columns of $\mathbf{W}$. The formal optimisation problem is

$$\min_{\mathbf{W}\in\mathcal{C}_1,\mathbf{H}\in\mathcal{C}_2} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 \ . \tag{1.2}$$

- SVD: Here $\mathcal{C}_1 = \mathbb{R}^{m\times k}$ and $\mathcal{C}_2 = \mathbb{R}^{k\times n}$, that is, only the low-rank constraints are imposed. This problem is already nonconvex due to the rank constraint but can be solved to global optima via the SVD. In this example, given the rank-k approximation of $\hat{\mathbf{X}}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\top$ we can set $\mathbf{W} = \mathbf{U}_k$ and $\mathbf{H} = \mathbf{\Sigma}_k\mathbf{V}_k^\top$.

- NMF: In addition to the low-rank constraints we also impose the elements of $\mathbf{W}$ and $\mathbf{H}$ to be nonnegative. The constraints are therefore, $\mathcal{C}_1 = \mathbb{R}_+^{m\times k}$ and $\mathcal{C}_2 = \mathbb{R}_+^{k\times n}$.

- Interpolative Decomposition (ID): This model factors the input with $\mathbf{W}$ containing only k selected columns from the original matrix $\mathbf{X}$ [33]. $\mathbf{H}$ contains an $k \times k$ identity submatrix. The absolute value of entries of $\mathbf{H}$ are also constrained to be less than 2.

The different bases discovered by these models are shown in Figs. 1.3b to 1.3d. The SVD bases seem to have mixtures of all the categories in the vectors. The SVD is the best way to compress the information present in $\mathbf{X}$ in this low-rank representation. Unfortunately, the vectors discovered are not that interpretable (see Fig. 1.3b). The NMF bases look more like clothing articles than those of the SVD as seen in Fig. 1.3c. Different vectors capture various parts of the clothes like the sleeves and front parts of shirts. All images are formed as additive combinations of these basis vectors. The ID vectors are the clearest images since they are selected directly from the input. However, it seems difficult to form certain images, for example sandals, as combinations of the current basis.

1.2 Parallel Programming Model

In this dissertation, we are primarily concerned with distributed-memory parallel machines. In this programming model, we are provided a machine with $p \geq 1$ processors that do not share a common main memory. In such a setting, sharing information between different processors is not possible in constant time with respect to p. Thus, the cost of sending information is a key component of analysing distributed parallel algorithms in addition to the cost of computations.

1.2.1 Algorithmic Cost Model

We assume the $\alpha - \beta - \gamma$ model to analyse both the computation and communication costs of parallel algorithms [19]. Since the primary concern of this study is matrix operations, we consider only floating point operations (flops) as the primary computation cost. We assume a fixed cost of γ per flop and do not differentiate between the scalar operations of additions, subtractions, multiplications, divisions, and square roots. The cost of sending a message of

n "words" between two processors is modelled as $\alpha + \beta n$. Here α models the *latency* and β the *bandwidth* costs of sending a message. Although α, β, and γ are modelled as constants, they implicitly depend on the size of the system, p, as well as its topology. Words are typically double precision floating point entries of matrices. Therefore, the running time T of an algorithm performing F flops and transmitting W words in S messages is the sum of these three terms.

$$T = \alpha \cdot S + \beta \cdot W + \gamma \cdot F$$

1.2.2 Assumptions

We assume the following conditions in the simple model of parallel computation.

1. Homogeneous architecture: The cost parameters α, β are same for between every pair of processors and γ is the same on all processors.

2. Bidirectional unicasting: Each processor can only send/receive one message to/from one processor at a time. A processor can send and receive a message simultaneously. Messages travelling on the same link in opposite directions do not conflict.

3. Fully connected networks: We assume that any processor can send a message to any other processor directly through the communication network without any resource contention.

The fully connected network topology might seem stringent at first and not representative of real-world parallel machines. However, this assumption is done to simplify the algorithmic analysis of the various methods encountered in the dissertation. The lower bounds derived on this network are valid for any other topology, but its attainability via an algorithm depends on the details of the network. These bounds are commonly derived by focusing on a single processor's computations. Another reason for such an assumption is that we are often only provided a random subset of a large parallel machine's processors to perform our computations. These issues make designing topology aware algorithms a daunting task taking into account for the inherent variability in the processor assignments.

1.2.3 Collective Communication

The parallel algorithms described in this dissertation use the Message Passing Interface (MPI) library for sending and receiving messages on distributed systems [34]. MPI provides useful ways of organising point-to-point messages into collective communication operations involving more than one processor. Commonly used primitives like broadcast, reductions, gathers, and scatters, amongst others, are implemented in MPI and often optimised for different network topologies [19]. For a concise description of common collectives please refer to the article by Chan et al. [19, Figure 1.].

1.3 Contributions and Overview

In this dissertation, we present distributed-memory parallel algorithms for a certain class of CLRA problems. We focus on NMF, which imposes nonnegativity constraints on factors $\mathbf{W}$ and $\mathbf{H}$. Our algorithms are designed to scale to large problem sizes arising from a variety of sources like satellite images, social networks, text corpora to name a few. While our problems have nonnegativity constraints as the central theme, many parallelisation ideas can be applied to other forms of CLRA.

In Chapter 2, we formally describe the NMF problem and discuss its properties. Distributed-memory parallel algorithms for NMF are presented and studied. We also introduce the Parallel Low rank Approximation with Nonnegativity Constraints (PLANC) software package [35, 36], which is capable of computing NMF and its generalisation to higher order tensors. The computational kernels in PLANC are carefully designed to avoid unnecessary computation and communication enabling them to be scalable and efficient.

Next in Chapter 3, we extend PLANC to include symmetry constraints on the factor matrices in an efficient manner [37]. We implement the first distributed-memory parallel algorithms for Symmetric Nonnegative Matrix Factorisation (SymNMF). We introduce a new method for SymNMF based on the Gauss-Newton algorithm. We observe 70% efficiency while scaling up these methods.

Chapter 4 considers the case when we are presented with multiple input matrices from different modalities. In this setting, every data item is represented by a set of features along with connections to other data items it is similar or "close" to. The Joint Nonnegative Matrix Factorisation (JointNMF) algorithm is capable of handling datasets that contain both feature and connection information. We develop JointNMF in PLANC and study the various trade-offs encountered when approximating two large input matrices [38].

The ideas developed in the previous chapters extend naturally to the multilinear setting in, as we show in Chapter 5. The bottleneck computation Matricised-Tensor Times Khatri-Rao Product (MTTKRP), the higher dimensional analog to matrix multiplication, is developed in PLANC [36]. This kernel allows us to extend most of the machinery developed for NMF to tensors. Finally, we describe some avenues for future research which extend the work and ideas in this dissertation. In particular, we consider the notion of structure sparsity which appears in computations for various tasks like cross-validation, working with missing values, robust CLRA models [39], and in the semi-supervised setting.

The citations refer to the original published versions of this material. This dissertation provides additional details and updated results on new data and clusters.

CHAPTER 2
PRELIMINARIES: NMF AND ITS PARALLEL ALGORITHMS

This chapter provides the reader with some background in NMF including some of its theoretical aspects, geometric interpretation, variants, serial and parallel algorithms, and complexity. In addition, we will introduce PLANC, an open source, scalable, and flexible software package, for handling NMF and its popular variants. In particular it contains an efficient matrix-multiplication kernel which is the primary bottleneck for NMF.

2.1 Introduction

NMF has become a standard tool for analysing high-dimensional datasets. It is a CLRA technique for nonnegative data where we require the factor matrices to be component-wise nonnegative denoted by $\mathbf{W} \geq 0$ and $\mathbf{H} \geq 0$ (see Eq. (1.2)). These nonnegative constraints play an instrumental role in extracting meaningful and interpretable results from nonnegative datasets.

Let us formally define the NMF problem. Given a nonnegative matrix $\mathbf{X} \in \mathbb{R}_+^{m \times n}$ and an input rank k, we need to find factor matrices to solve the optimisation problem,

$$\min_{\mathbf{W} \geq 0, \mathbf{H} \geq 0} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 \ . \tag{2.1}$$

Here $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ are the low-rank basis and embedding matrices, respectively. Sometimes the objective in Eq. (2.1) is simply the Frobenius norm (i.e. $\|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F$), however optimising it is tantamount to solving the squared version.

Unlike other factorisations, NMF only approximates the input matrix and thus the term "factorisation" in the acronym is a misnomer. However, this convention is the standard in the community. Though Eq. (2.1) can be defined using any error measure between $\mathbf{X}$ and $\mathbf{W}\mathbf{H}$, we choose the Frobenius norm for two primary reasons. It corresponds to addition of i.i.d. Gaussian noise to the entries of $\mathbf{X}$, which is a reasonable assumption for many datasets. The second reason is that it leads to a smooth optimisation problem which makes it easier to design algorithms to handle Eq. (2.1).

The first description of NMF in its modern form is by Paatero and Tapper [40]. They termed the model "positive matrix factorisation" even though the factors were constrained to be nonnegative rather than positive. The term NMF was popularised by the seminal paper by Lee and Seung, where they demonstrated the model's ability to produce physically meaningful results for image and text data [14]. After this work, NMF became a standard tool in the modern data analyst's toolkit and has been used in a wide variety of applications,

Table 2.1: Linear algebra notation.

Symbol	Description
$\mathbb{R}_+^n$	Nonnegative orthant.
$\mathbf{X}$	Input matrix (typically $\mathbf{X} \in \mathbb{R}_+^{m \times n}$).
$\mathbf{S}$	Symmetric input matrix (typically $\mathbf{S} \in \mathbb{R}_+^{n \times n}$).
$\mathbf{W}$	Basis matrix (typically $\mathbf{W} \in \mathbb{R}_+^{m \times k}$).
$\mathbf{H}$	Embedding matrix (typically $\mathbf{H} \in \mathbb{R}_+^{k \times n}$).
$\mathbf{I}_n$	$n \times n$ identity matrix.
$\mathbf{1}$	Matrix or vector of all ones (of appropriate size).
$\mathbf{0}$	Matrix or vector of all zeros (of appropriate size).
$\mathrm{vec}(\cdot)$	Vectorises a matrix by stacking its columns on top of each other.
$[\cdot]_+$	Projects negative entries of input to 0.

including feature extraction in images, topic modelling, hyperspectral unmixing, amongst others [10]. For a more complete history of the origins of NMF and its earlier roots in analytical chemistry, geosciences, and linear algebra, we refer the reader to the excellent textbook by Gillis [10].

2.1.1 Notation

Table 2.1 summarises the notation used throughout the document. We use bold lowercase fonts for representing vectors (e.g., $\mathbf{x}$) and bold uppercase for matrices (e.g., $\mathbf{A}$). For $\mathbf{A}$, its ith column is represented as $\mathbf{a}_i$. $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ are also used for denoting the ith row and jth column of $\mathbf{A}$, respectively. Elements of $\mathbf{A}$ is shown as a_{ij} and $\mathbf{A}(i,j)$ interchangeably.

2.1.2 Geometrical Interpretation

The *conical hull* is the set of all *conic* combinations of a finite set of vectors $S = \{\mathbf{v}_i\}$, denoted by cone (S), where

$$\text{cone}\,(S) = \{\mathbf{x} : \mathbf{x} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n, \alpha_i \geq 0\} \ .$$

The set of vectors often comprise the columns of a matrix and we abuse notation to refer to its conical hull as $\text{cone}\,(S) = \text{cone}\,(\{\mathbf{v}_i\}) = \text{cone}\,([\mathbf{v}_1\ \mathbf{v}_2\ \ldots\ \mathbf{v}_n]) = \text{cone}\,(\mathbf{V})$. NMF computes a conic combination of the nonnegative basis vectors. For NMF, every input data item $\mathbf{x}_i$ is approximated by the conical combination of the columns of $\mathbf{W}$, i.e. $\mathbf{x}_i \approx \mathbf{W}\mathbf{h}_i = \sum_{r=1}^{k} \mathbf{w}_r h_{ri}$. Equivalently, for the exact factorisation $\mathbf{X} = \mathbf{W}\mathbf{H}$, we can see for all i,

$$\mathbf{x}_i \in \text{cone}\,(\mathbf{W}) \subseteq \mathbb{R}_+^m \ ,$$

and

$$\text{cone}\,(\mathbf{X}) \subseteq \text{cone}\,(\mathbf{W}) \subseteq \mathbb{R}_+^m \ .$$

The subset inequalities shown above is known as the nested cone problem, where we need to find the cone, namely $\text{cone}\,(\mathbf{W})$, nested between $\text{cone}\,(\mathbf{X})$ and $\mathbb{R}_+^m = \text{cone}\,(\mathbf{I}_m)$.

The NMF basis vectors tend to "wrap" the data from the "outside" because it tries to nest the cone of the input inside the cone generated by the basis matrix like in the exact factorisation. These vectors can be contrasted to the more "holistic" bases vectors generated by the SVD. We demonstrate this by constructing a synthetic dataset with two clusters shown in Fig. 2.1. Each cluster is generated from a multivariate normal distribution with

$$\boldsymbol{\mu}_1 = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.3 \end{bmatrix} \ , \ \boldsymbol{\mu}_2 = \begin{bmatrix} 0.5 \\ 0.7 \\ 0.9 \end{bmatrix} \ , \ \sigma^2 = 0.001 \ ,$$

and identical covariance matrices $\sigma^2 \mathbf{I}_3$. They are depicted as red and green data clouds in Fig. 2.1 with the means ($\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$) shown as blue lines. The SVD relative error, $\frac{\|\mathbf{X}-\mathbf{W}\mathbf{H}\|_F}{\|\mathbf{X}\|_F}$, is 0.031626 whereas NMF has one of 0.031644. SVD achieves the global minimum in terms of the objective. Its basis vectors, capture holistic information about the data as seen by the first left singular vector being a combination of the data clouds in the middle. The second vector is orthogonal to first and captures the next direction with most

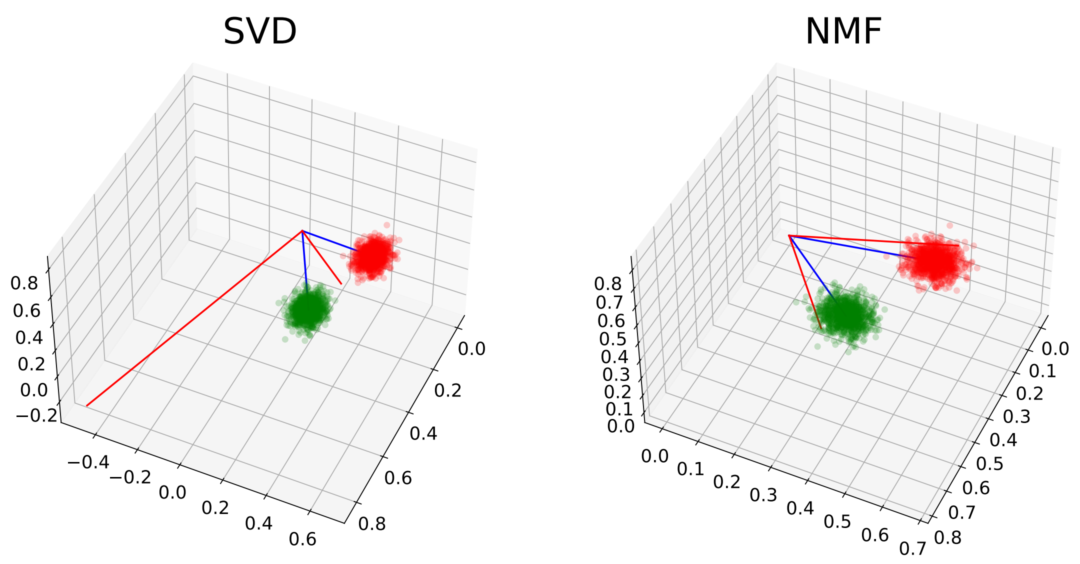
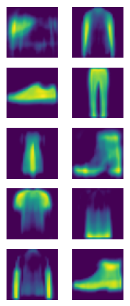
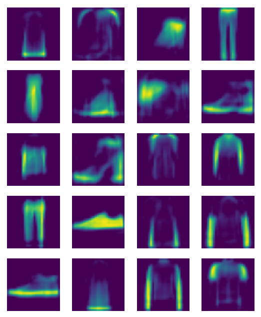


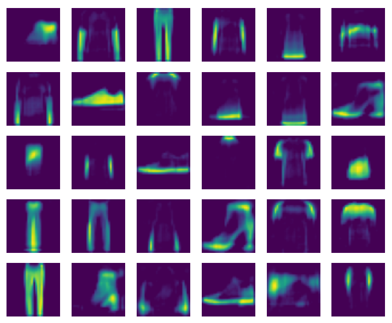
Figure 2.1: Comparison between NMF and SVD bases. The two clusters, of 1,000 points each, are shown as data clouds of red and green respectively. The blue lines correspond to the means of the clusters. The SVD and NMF bases are shown as red lines. Notice that the SVD bases capture more holistically information with regards to the data stored in $\mathbf{X}$ whereas NMF tends to wrap the input data from the outside.

information. NMF on the other hand, tends to "wrap" around the data as stated earlier. Its two bases correspond to each of two different clusters present in the data albeit, two of the more outermost points in the clusters respective data clouds. As expected, its relative error is also higher than that of the SVD.

The nonnegativity constraints typically produce "sparser" solutions than the input data. This claim can be mathematically understood by the complementary slackness conditions on smooth optimisations problem over $\mathbf{x}$ (see Eq. (2.16)). We must have $x_i = 0$ whenever $(\nabla f(\mathbf{x}))_i > 0$ which tends to produce solutions with many zeros in them. The sparser solutions manifest as the parts-based approximations discovered by NMF. We revisit the image collection from Fig. 1.3. We run NMF with varying embedding ranks and look at the basis vectors found. Notice that the image vectors become sparser as the rank increases. Unfortunately, these properties of NMF come at the price of computational tractability, which we shall see in the next section.


(a) $k = 10$.


(b) $k = 20$.


(c) $k = 30$.

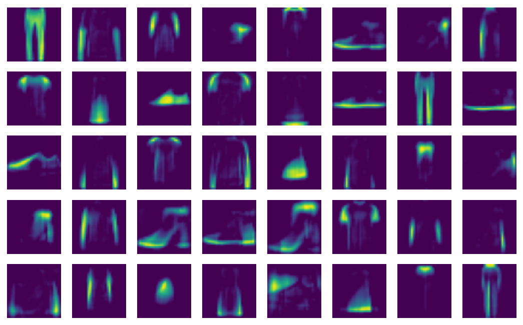

(d) $k = 40$.

Figure 2.2: Parts-based representation by NMF on the Fashion-MNIST dataset. Notice how the clusters break into smaller “parts” as we increase k. For example, the shoes from (a) change into soles, heels, and different upper sections for sneakers and boots.

2.1.3 Hardness of NMF

The NMF optimisation problem Eq. (2.1) is nonconvex. This fact can be seen via the following counterexample to the convexity property, namely,

$$f\left(\theta \mathbf{a} + (1-\theta)\mathbf{b}\right) \le \theta f\left(\mathbf{a}\right) + (1-\theta) f\left(\mathbf{b}\right) \ .$$

Here $f\left(\mathbf{a}\right) = f\left(\mathbf{W}, \mathbf{H}\right) = \left\|\mathbf{X} - \mathbf{W}\mathbf{H}\right\|_F^2$ is the function we are trying to minimise with $\mathbf{a} = \begin{bmatrix} \text{vec}\left(\mathbf{W}\right) \\ \text{vec}\left(\mathbf{H}\right) \end{bmatrix}$ being the inputs to f reshaped as a long vector.

Consider the rank-3 input matrix[1]

$$\mathbf{X} = \begin{bmatrix} 2 & 1 & 2 \\ 3 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix} \ ,$$

and the following two rank-2 candidate cases

$$\mathbf{W}_1 = \begin{bmatrix} 4 & 3 \\ 4 & 3 \\ 1 & 2 \end{bmatrix} \quad \mathbf{H}_1 = \begin{bmatrix} 3 & 2 & 1 \\ 1 & 3 & 3 \end{bmatrix} \quad \text{and} \quad \mathbf{W}_2 = \begin{bmatrix} 1 & 3 \\ 1 & 2 \\ 2 & 4 \end{bmatrix} \quad \mathbf{H}_2 = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 3 \end{bmatrix} \ .$$

Computing the convexity condition, with $\theta = 0.8$, we get

$$\begin{aligned} f\left(\theta \mathbf{a} + (1-\theta)\mathbf{b}\right) &= f\left(\theta \mathbf{W}_1 + (1-\theta)\mathbf{W}_2, \theta \mathbf{H}_1 + (1-\theta)\mathbf{H}_2\right) &&= 1{,}384.1136 \\ \theta f\left(\mathbf{a}\right) + (1-\theta) f\left(\mathbf{b}\right) &= \theta f\left(\mathbf{W}_1, \mathbf{H}_1\right) + (1-\theta) f\left(\mathbf{W}_2, \mathbf{H}_2\right) &&= 1{,}382.2000 \end{aligned}$$

which violates the convexity condition and provides us our counterexample.

[1]These values were found by computer simulation.

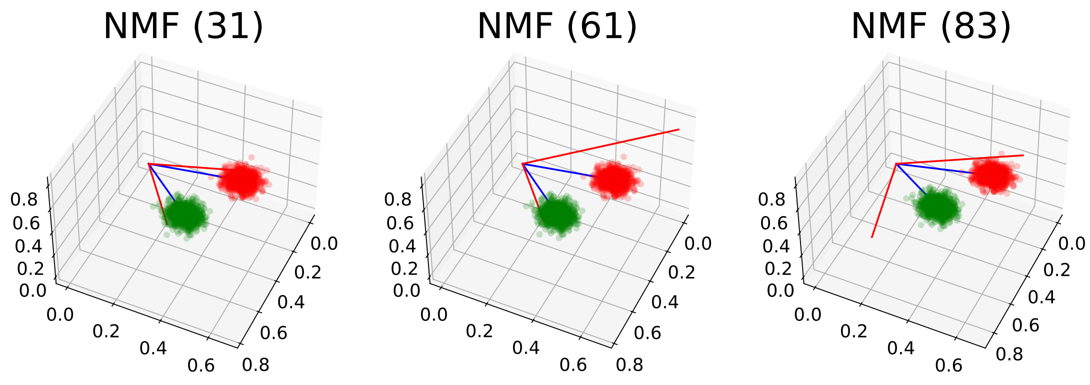


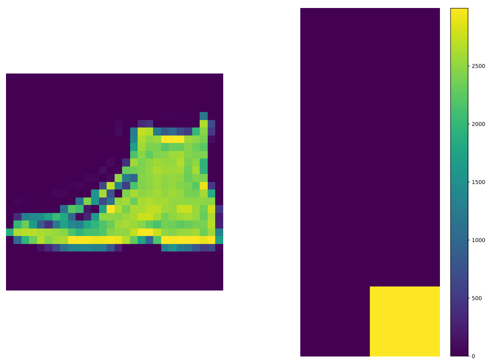
Figure 2.3: NMF computed with different starting points. The blue and red lines are the cluster centroids and NMF bases respectively. Notice that we get different basis vectors each time. The relative errors for the 3 cases are 0.031659, 0.031664, and 0.031626.

Vavasis showed that the exact NMF factorisation $\mathbf{X} = \mathbf{WH}$ is NP-hard when k is part of the input [41]. This result also applies to the standard NMF optimisation problem. Thus, when using NMF, it is reasonable to expect a solution which is a stationary point or local minimum rather than one which finds the global optimum of Eq. (2.1) . Most NMF algorithms are iterative in nature, where we start from an initial "guess" solution and successively find better approximations of $\mathbf{X}$. We shall discuss these algorithms in more detail in the next section. Fig. 2.3 shows the basis vectors discovered when we run NMF starting from different initial guesses $\mathbf{W}_0$ and $\mathbf{H}_0$. Notice that bases for all starting points attempt to wrap around the input data from the outside[2].

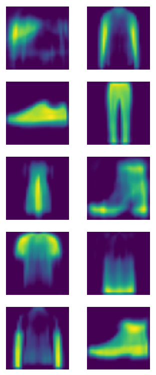
2.1.4 Clustering via NMF

NMF has seen great success as a clustering tool in many different domains, like images, text corpora, and audio signals, to name a few [10, 14]. This section offers some intuition for this behaviour. Typically NMF is run with the low-rank parameter k chosen to be the desired number of clusters. The hope is that each basis vector discovered corresponds to one of the data clusters, i.e., the basis vectors behave as cluster representatives. The ten-

[2]The error for the final case, with seed 83, seems to match the SVD error from the previous section but they differ in the last few significant digits. SVD relative error: 0.03162598913926414 and NMF relative error: 0.031625989139264725.



(a) Test image along with its corresponding $\mathbf{h}$.



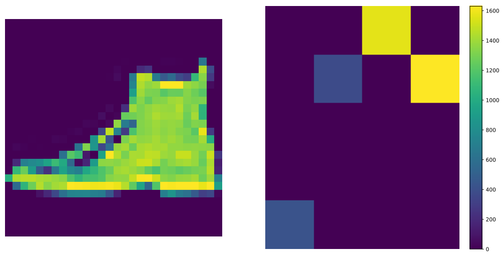
(b) NMF bases with $k = 10$.

Figure 2.4: The Fashion-MNIST dataset contains 10 categories of clothing. Running NMF with $k = 10$ allows us to cluster these images. The query image and its corresponding column in $\mathbf{H}$ is shown in (a) along with the bases in (b). We see a high activation value corresponding to the basis vector that looks a like boot.

dency of the columns of $\mathbf{W}$ to wrap the input data from outside, rather than take a holistic view of the entire input like in SVD, helps in this aspect. We can intuitively attribute this tendency to the nested cone property and sparse solutions obtained in the NMF optimisation (see Section 2.1.2). This property is seen in all the synthetic cases in Fig. 2.3 as well in the image dataset (Figs. 2.4b and 2.5b).

With this wrapping assumption, we can assign an input data point $\mathbf{x}_i$ to its corresponding cluster by examining its lower-dimensional representation $\mathbf{h}_i$. This column vector contains the contributions from each of the cluster representatives. Taking the index of the maximum entry allows us to cluster $\mathbf{x}_i$. An example of such a clustering is shown in Fig. 2.4. Another popular way to cluster the input data is to run a secondary method, like K-Means, on the columns of $\mathbf{H}$ [25, 42]. Since $k \ll \min{(m, n)}$, this procedure would run much faster than running the same method directly on the input.

This two-step procedure allows us to cluster even when the selected k is different from the number of clusters present. Choosing k larger than the number of clusters is often desirable to get the parts-based decompositions on the input. The nonnegative constraints on $\mathbf{H}$ force the approximation of $\mathbf{x}_i$ to be done only via additive, not subtractive, combinations


(a) Test image along with its corresponding $\mathbf{h}$.

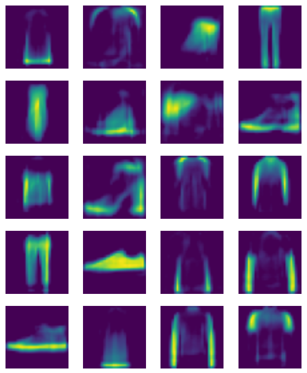

(b) NMF bases with $k = 20$.

Figure 2.5: Parts-based representation of an image via NMF. The query image of the boot is recreated combining the different parts of a shoe (sole, heel, middle, and upper sections). Running a different clustering method, like K-Means, on the columns of $\mathbf{H}$ is another way to cluster the input.

of the basis vectors. $\mathbf{h}_i$ is the "embedding" of $\mathbf{x}_i$ computed via NMF. This property often simplifies interpreting the result of the approximation, as seen in Fig. 2.5.

2.2 Serial Algorithms

We now provide an overview of algorithms to solve Eq. (2.1) from the perspective of Block Coordinate Descent (BCD) methods. BCD methods are a popular tool in nonlinear optimisation and are the basis of some of the most successful NMF algorithms. This section borrows heavily from other sources [10, 27].

2.2.1 BCD framework for NMF

The BCD framework is an iterative process to solve an optimisation problem. It groups the variables in the problem into several disjoint subgroups and iteratively solves the variable in a subgroup keeping the others fixed. It is particularly useful when these groupings allow the subproblems to solved quickly or exactly.

Consider the general optimisation problem for $\mathbf{x} \in \mathbb{R}^n$,

$$\min_{\mathbf{x} \in \mathcal{C}} \ f(\mathbf{x}) \ . \tag{2.2}$$

We assume that $\mathcal{C} \subseteq \mathbb{R}^n$ is closed and convex and can be represented as the Cartesian product,

$$\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2 \times \ldots \mathcal{C}_m \ , \tag{2.3}$$

where $\mathcal{C}_i \subseteq \mathbb{R}^{n_i}$ is closed and convex with $\sum_{i=1}^{m} n_i = n$. The input vector is similarly partitioned as $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix}$. The BCD method, also known as the *nonlinear Gauss-Seidel* method, generates the next iterate $\mathbf{x}^{(t+1)}$ given the current iterate $\mathbf{x}^{(t)}$ according to the update rule

$$\mathbf{x}_i^{(t+1)} = \min_{\mathbf{z} \in \mathcal{C}_i} f\left(\mathbf{x}_1^{(t+1)}, \mathbf{x}_2^{(t+1)}, \ldots, \mathbf{x}_{i-1}^{(t+1)}, \mathbf{z}, \mathbf{x}_{i+1}^{(t)}, \ldots, \mathbf{x}_m^{(t)}\right) \ . \tag{2.4}$$

Notice that it uses the most recently updated values of all the fixed blocks. This approach ensures that the objective value never increases after every update.

Convergence guarantees for BCD

There are two main convergence guarantees for BCD methods which we state here without proof. We direct readers to the appropriate source materials for more information [10, 43–45]. The original sources for the theorems are cited in the statements below but the corresponding text is from Gillis [10]. The order in which blocks are updated in the BCD algorithm are arbitrary as long as every block is updated once every fixed constant number of iterations. This updating order is known as essentially cyclic block updates.

Theorem 2.2.1 ([43, 44, Proposition 2.7.1])**.** *The limit points of the iterates of an exact BCD algorithm are stationary points provided that the following conditions hold.*

1. *The objective function is continuously differentiable.*
2. *Each block is required to belong to a closed convex set.*
3. *The minimum computed at each iteration for a given block of variables is uniquely attained.*
4. *The objective function values in the interval between all iterates and the next (which is obtained by updating a single block of variables) is monotonically nonincreasing.*

In the context of NMF, the objective function of Eq. (2.1) is continuously differentiable. The nonnegative orthant, and its subsets, are closed convex sets. The final two conditions hold true if the subproblems are strongly convex per block variable [10].

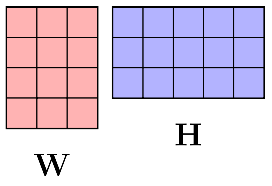
Theorem 2.2.2 ([45, Corollary 2])**.** *The limit points of the iterates of an exact two-block BCD algorithm are stationary points provided that the following two conditions hold.*

1. *The objective function is continuously differentiable.*
2. *Each block of variables is required to belong to a closed convex set.*

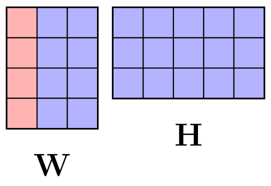
This special case for two blocks gets rid of the last two constraints from Theorem 2.2.1. It is particularly useful for NMF as it corresponds to the natural partitioning of the variables into the factor matrices, $\mathbf{W}$ and $\mathbf{H}$.

Blocking strategies for NMF

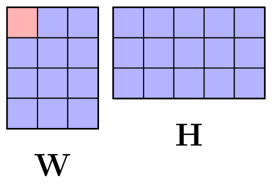
For NMF, the most common blocking strategies are shown in Fig. 2.6. We can either update entire factor matrices at a time in the two-block approach (Fig. 2.6a), or update columns of $\mathbf{W}$ and rows of $\mathbf{H}$ (Fig. 2.6b), or individual entries of both factor matrices (Fig. 2.6c).



(a) Two blocks.



(b) $2k$ blocks.



(c) $(m+n)\,k$ blocks.

Figure 2.6: Different BCD blocking options for NMF. The block being updated are shown in red with the blue blocks remaining fixed. We go from the largest blocks where the entire factor matrices are updated in the two-block's case to when only single scalars are updated in the $(m+n)\,k$-block's case.

The two-block BCD is the most popular strategy amongst iterative NMF algorithms. This approach of updating $\mathbf{H}$ while keeping $\mathbf{W}$ fixed and vice versa is also known as alternating optimisation. The subproblems formed in this natural partitioning of the variables are Nonnegative Least-Squares (NLS) ones. Therefore, this approach is also known as Alternating Nonnegative Least-Squares (ANLS). There are a few primary advantages to this partitioning.

1. The relation, $\|\mathbf{X}-\mathbf{WH}\|_F^2 = \left\|\mathbf{X}^\mathsf{T}-\mathbf{H}^\mathsf{T}\mathbf{W}^\mathsf{T}\right\|_F^2$, is symmetric. Therefore, if an efficient algorithm is available for updating $\mathbf{H}$ we can reuse it for updating $\mathbf{W}$.

2. The problem of minimising $\min\limits_{\mathbf{H}\geq 0}\|\mathbf{X}-\mathbf{WH}\|_F^2$ can be broken down into independent subproblems across the columns of $\mathbf{X}$. That is, we get n different NLS minimisations,

$$\min_{\mathbf{h}_i\geq 0}\|\mathbf{W}\mathbf{h}_i-\mathbf{x}_i\|_2^2 \ .$$

   An added benefit is that we can reuse certain computations, like $\mathbf{W}^\mathsf{T}\mathbf{W}$ for the gradient with respect to $\mathbf{H}$ during the update of all columns $\mathbf{h}_i$.

3. The two-block case has less restrictive conditions for convergence than the general many-block BCD case (see Theorem 2.2.2).

Algorithm 1 Two-block coordinate descent for NMF

```
Require: Input matrix $\mathbf{X} \in \mathbb{R}_+^{m \times n}$ and low rank parameter $k$.
Ensure: $\mathbf{W} \geq 0, \mathbf{H} \geq 0$, is a rank-$k$ approximation $\mathbf{X} \approx \mathbf{WH}$.
1: Initialise $\mathbf{W}^{(0)}$ and $\mathbf{H}^{(0)}$.
2: while $t = 1, 2, \ldots$ do                    ▷ Till some stopping criteria is met.
3:     $\mathbf{W}^{(t)} \leftarrow \underset{\mathbf{W} \geq 0}{\operatorname{argmin}} \left\| \mathbf{X}^\mathsf{T} - {\mathbf{H}^{(t-1)}}^\mathsf{T} \mathbf{W}^\mathsf{T} \right\|_F^2$
4:     $\mathbf{H}^{(t)} \leftarrow \underset{\mathbf{H} \geq 0}{\operatorname{argmin}} \left\| \mathbf{X} - \mathbf{W}^{(t)} \mathbf{H} \right\|_F^2$
5: end while
```

2.2.2 Algorithms

We can now describe the standard two-block BCD algorithm for NMF in Algorithm 1. The key computation kernel in this algorithm is the NLS solves in Lines 3 and 4. We can solve this subproblem in a variety of different ways to get different NMF algorithms. If the BCD employs a blocking scheme other than the two-block one described above we can simply arrange the blocks in alternating optimising manner. That is, we update an entire factor matrix before moving to the next one. This block ordering is no longer a two-block BCD method, and Lines 3 and 4 will be replaced by $\mathbf{W}^{(t)} \leftarrow \operatorname{update}\left(\mathbf{X}, \mathbf{H}^{(t-1)}\right)$ and $\mathbf{H}^{(t)} \leftarrow \operatorname{update}\left(\mathbf{X}, \mathbf{W}^{(t)}\right)$ respectively. The $\operatorname{update}(\cdot, \cdot)$ function updates a factor given the most recent iterate of the other factor matrix. Some popular NLS solvers are described later in Section 2.2.2.

We shall briefly comment on the stopping criteria for NMF. The gradient of the objective Eq. (2.1) with respect to the two factor matrices are

$$\nabla_\mathbf{W} f = 2\left(\mathbf{W}\mathbf{H}\mathbf{H}^\mathsf{T} - \mathbf{X}\mathbf{H}^\mathsf{T}\right) \,, \tag{2.5}$$

$$\nabla_\mathbf{H} f = 2\left(\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H} - \mathbf{W}^\mathsf{T}\mathbf{X}\right) \,. \tag{2.6}$$

We define the projected gradient with respect to $\mathbf{W}$ as

$$\nabla^p_{\mathbf{W}} f = \begin{cases} \nabla_{\mathbf{W}} f_{ij} & \text{if } \nabla_{\mathbf{W}} f_{ij} < 0 \text{ or } \mathbf{W}_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{2.7}$$

Similarly, we can define the projected gradient for $\mathbf{H}$. Next, we define $\delta\left(t\right)$ for iteration t of Algorithm 1 as

$$\delta\left(t\right) = \sqrt{\left\|\nabla^p_{\mathbf{W}^{(t)}} f\right\|_F^2 + \left\|\nabla^p_{\mathbf{H}^{(t)}} f\right\|_F^2} . \tag{2.8}$$

Then a stopping criteria of $\frac{\delta(t)}{\delta(0)} \leq \epsilon$ guarantees the stationarity of the final solutions [27]. However, one must be careful when implementing such a solution since they are quite sensitive to scaling degeneracy, which refers to the fact that if $\left(\mathbf{W}, \mathbf{H}\right)$ is a solution to Eq. (2.1) then so is $\left(\alpha\mathbf{W}, \frac{\mathbf{H}}{\alpha}\right)$ [10]. In practice, putting limits on the number of iterations, time, change in objective value, and change in iterate norms are simple but effective stopping criteria.

Update Algorithms

In this subsection, we consider updating algorithms for the NLS updates of the factor at each inner iteration of the algorithm (Lines 3 and 4 of Algorithm 1). The general problem to be solved in each inner iteration is a constrained least-squares problem of the form

$$\mathbf{C} \leftarrow \underset{\mathbf{C} \geq 0}{\operatorname{argmin}} \left\|\mathbf{A}\mathbf{C} - \mathbf{B}\right\|_F^2 . \tag{2.9}$$

We shall list popular updating methods that (approximately) solve Eq. (2.9) by first forming $\mathbf{A}^\mathsf{T}\mathbf{A}$ and $\mathbf{A}^\mathsf{T}\mathbf{B}$ These matrix terms appear in the gradient of the objective function. In the case of updating the factor matrix $\mathbf{H}$ we need to solve Eq. (2.9) with $\mathbf{C} = \mathbf{H}$, $\mathbf{A} = \mathbf{W}$, and $\mathbf{B} = \mathbf{X}$. Computing $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{W}^\mathsf{T}\mathbf{X}$ takes mk^2 and $2mnk$ flops, respectively (assuming we use standard matrix multiplication and accounting for symmetry). There is an analogous computation for updating $\mathbf{W}$ taking $(nk^2 + 2mnk)$ flops. These matrix multiplications, $\mathbf{W}^\mathsf{T}\mathbf{W}, \mathbf{H}\mathbf{H}^\mathsf{T}, \mathbf{W}^\mathsf{T}\mathbf{X}$, and $\mathbf{H}\mathbf{X}^\mathsf{T}$, are often the main bottleneck for NMF, costing

$$F_{\text{matmuls}} = \big((m+n)\,k^2 + 4mnk\big) \text{ flops} . \tag{2.10}$$

When $\mathbf{X}$ is sparse, the flop count changes to

$$F_{\text{matmuls}} = \big((m+n)\,k^2 + 4\text{nnz}(\mathbf{X})\big) \quad . \tag{2.11}$$

We briefly describe the different solvers and update algorithms along with their computational complexity below.

Multiplicative Updating (MU): The MU solve is an elementwise operation [14]. The update rule for element (i, j) of $\mathbf{C}$ is

$$\mathbf{C}(i,j) \leftarrow \mathbf{C}(i,j)\frac{\mathbf{A}^\mathsf{T}\mathbf{B}(i,j)}{\big(\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{C}\big)\,(i,j)} \ . \tag{2.12}$$

While this rule does not solve Eq. (2.9) to optimality it ensures a reduction in the objective value from the initial value of $\mathbf{C}$. Note that Eq. (2.12) breaks down if the denominator becomes zero. In practice, a small value is added to the denominator to prevent this situation. The costs of computing $\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{C}$, namely $\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{W}$ and $\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H}$, adds $2\,(m+n)\,k^2$ flops to the runtime of a single iteration of Algorithm 1.

Hierarchical Alternating Least-Squares (HALS): HALS updates are performed on individual rows of $\mathbf{C}$ [46, 47]. The update rule for row i can be written in closed form as

$$\mathbf{C}(i,:) \leftarrow \left[\mathbf{C}(i,:) + \frac{(\mathbf{A}^\mathsf{T}\mathbf{B})(i,:) - \big(\mathbf{A}^\mathsf{T}\mathbf{A}\big)\,(i,:)\mathbf{C}}{(\mathbf{A}^\mathsf{T}\mathbf{A})(i,i)}\right]_+ \ , \tag{2.13}$$

where $[\cdot]_+$ is the projection operator onto $\mathbb{R}_+$.

The rows of $\mathbf{C}$ are updated in order so that the latest values are used in every update step. HALS has been observed to produce unbalanced results with either very large or very small values appearing in the factor matrices [27, 46]. Normalising the rows of $\mathbf{C}$ after every update via Eq. (2.13) has been proposed to alleviate this problem [27, 46]. The cost for updating via HALS is $2\,(m+n)\,k^2$ after forming the different matrix products.

Projected Gradient Descent (PGD): Another method to tackle Eq. (2.9) is via PGD, first described for NMF by Lin [48]. The gradient for Eq. (2.9) is given by

$$\begin{aligned} f(\mathbf{C}) &:= \|\mathbf{AC}-\mathbf{B}\|_F^2 \ , \\ \nabla f(\mathbf{C}) &= 2\left(\mathbf{A}^\mathsf{T}\mathbf{AC}-\mathbf{A}^\mathsf{T}\mathbf{B}\right) \ . \end{aligned}$$

The Hessian for f is $\mathbf{A}^\mathsf{T}\mathbf{A}$ and therefore its gradient is Lipschitz continuous with constant $L=\lambda_1\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)=\sigma_i\left(\mathbf{A}\right)^2=\|\mathbf{A}\|_2^2$ [10]. The PGD update is given by,

$$\mathbf{C}=\left[\mathbf{C}-\frac{2}{L}\left(\mathbf{A}^\mathsf{T}\mathbf{AC}-\mathbf{A}^\mathsf{T}\mathbf{B}\right)\right]_+ \tag{2.14}$$

where $[\cdot]_+$ is the projection operator onto $\mathbb{R}_+$. Computing L takes $O\left(k^3\right)$ flops for the Gram matrices $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}\mathbf{H}^\mathsf{T}$ and computing and applying the gradient step takes $O\left((m+n)\left(k^2+k\right)\right)$ flops. Quite often, a fixed stepsize is used instead of computing L every iteration to avoid computing eigenvalues.

PGD can also be accelerated if $\mathbf{H}$ is updated several times before $\mathbf{W}$ and can improve from a linear to a quadratic convergence rate [10]. However, acceleration often loses the property of monotonically decreasing the objective.

Block Principal Pivoting (BPP): BPP is an active-set like method for solving NLS problems. The main subroutine of BPP is the single right-hand side version of Eq. (2.9),

$$\mathbf{c} \leftarrow \underset{\mathbf{c} \geq 0}{\operatorname{argmin}} \|\mathbf{A}\mathbf{c} - \mathbf{b}\|_2^2 \ . \tag{2.15}$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for Eq. (2.15) are specified by $\mathbf{c}$ and the gradient $\mathbf{y} = \mathbf{A}^{\mathsf{T}}\mathbf{A}\mathbf{c} - \mathbf{A}^{\mathsf{T}}\mathbf{b}$. $\mathbf{c} \geq 0$, $\mathbf{y} \geq 0$, and $\mathbf{x} * \mathbf{y} = \mathbf{0}$, where $*$ is the Hadamard or elementwise product.

$$\mathbf{y} \geq 0 \tag{2.16a}$$

$$\mathbf{c} \geq 0 \tag{2.16b}$$

$$c_i y_i = 0 \quad \forall i \tag{2.16c}$$

The complementary slackness criteria from the KKT conditions forces the support sets, i.e., the nonzero elements, of $\mathbf{c}$ and $\mathbf{y}$ to be disjoint. This disjoint property makes Eq. (2.16) an instance of the *Linear Complementarity Problem* which arises frequently in quadratic programming. When $k \ll \min(m.n)$, active-set methods are preferred since they deal with small matrices of size $m \times k$, $n \times k$ and $k \times k$. In the optimal solution, the active set is the set of indices where $c_i = 0$, and the remaining indices are referred to as the passive set. Once the active set is found, we can find the optimal solution to Eq. (2.15) by solving an unconstrained least-squares problem on the passive set of indices. The BPP algorithm attempts to find the active set by greedily swapping indices between the intermediate active and passive sets until it finds a solution that satisfies the KKT conditions. Unconstrained least-squares is solved using the normal equations. Kim and Park discuss the method in greater detail [49].

Solving n NLS problems in k variables with the BPP algorithm takes about $O\left(nk^3 s\left(k\right)\right)$ flops, where $s(k) \leq 2^k$ is the number of active sets explored. In practice, $s(k)$ is typically much smaller than 2^k. Overall, using BPP to update both $\mathbf{W}$ and $\mathbf{H}$ takes $O\left(\left(m+n\right) k^3 s(k)\right)$

flops [10]. If k is small, BPP should be the method of choice since it solves the subproblems exactly. However, as k becomes large BPP tends to become slow when compared to other optimisation methods.

Alternating Direction Method of Multipliers (ADMM): In the ADMM solver the optimisation problem Eq. (2.9) is reformulated by introducing an auxiliary variable $\hat{\mathbf{C}}$,

$$\begin{aligned} &\min_{\mathbf{C},\hat{\mathbf{C}}} \quad \frac{1}{2}\left\|\mathbf{A}\hat{\mathbf{C}} - \mathbf{B}\right\|_F^2 + r(\mathbf{C}) \\ &\text{subject to } \mathbf{C} = \hat{\mathbf{C}} \ , \end{aligned} \tag{2.17}$$

where $r(\cdot)$ is the penalty function for nonnegativity [50]. It is 0 if $\mathbf{C} \geq 0$ and ∞ otherwise. The updates for the ADMM algorithm are given by

$$\begin{aligned} \hat{\mathbf{C}} &\leftarrow \left(\mathbf{A}^{\mathsf{T}}\mathbf{A} + \rho\mathbf{I}\right)^{-1}\left(\mathbf{A}^{\mathsf{T}}\mathbf{B} + \rho\left(\mathbf{C} + \mathbf{U}\right)^{\mathsf{T}}\right) \\ \mathbf{C} &\leftarrow \operatorname*{argmin}_{\mathbf{C}} r(\mathbf{C}) + \frac{\rho}{2}\left\|\mathbf{C} - \hat{\mathbf{C}} + \mathbf{U}\right\|_F^2 \\ \mathbf{U} &\leftarrow \mathbf{U} + \mathbf{C} - \hat{\mathbf{C}} \ , \end{aligned} \tag{2.18}$$

where $\mathbf{U}$ is the scaled version of the dual variables corresponding to the equality constraints, $\mathbf{C} = \hat{\mathbf{C}}$, and ρ is a step size specified by the user. $\mathbf{U}$ is initialised as a matrix of all zeros. The advantage of using ADMM is the clever splitting of the nonnegativity constraints into updates of two blocks of variables $\mathbf{C}$ and $\hat{\mathbf{C}}$. This splitting allows for an unconstrained least-squares solve for $\hat{\mathbf{C}}$ and element-wise projections onto $\mathbb{R}_+$ for $\mathbf{C}$. One set of Eq. (2.18) takes about $O\left(k^3 + (m+n)\,k\right)$ flops to update both $\mathbf{W}$ and $\mathbf{H}$.

We can accelerate this solve by repeating the updates given by Eq. (2.18) more than once. One important fact to notice is that the same matrix $\mathbf{A}^{\mathsf{T}}\mathbf{B}$ and matrix inverse $\left(\mathbf{A}^{\mathsf{T}}\mathbf{A} + \rho\mathbf{I}\right)^{-1}$ are used for all the updates. We can therefore cache $\mathbf{A}^{\mathsf{T}}\mathbf{B}$ and the Cholesky decomposition of $\left(\mathbf{A}^{\mathsf{T}}\mathbf{A} + \rho\mathbf{I}\right)$ to save some computations during subsequent updates. The stopping criteria described by Huang et al., is based on $\|\mathbf{C}\|_F$, $\left\|\hat{\mathbf{C}}\right\|_F$, and $\|\mathbf{U}\|_F$ [50] . By default, a good choice for ρ is $\|\mathbf{A}\|_F^2/k$, where k is the number of columns of $\mathbf{A}$ (rank of the NMF decomposition) [50]. For a comprehensive guide to the ADMM method, including convergence properties and selection of optimal ρ, please refer to article by Boyd et al. [51].

Nesterov-type algorithm: There are many different Nesterov-type algorithms for NMF [10] but we outline the one by Liavas et al. [52]. Their method solves a modified version of NLS, Eq. (2.9), with the introduction of a proximal term with an auxiliary matrix $\mathbf{C}_*$. The proximal term is useful to handle ill-conditioned instances and guarantee strong convexity. The objective function tackled is

$$\min_{\mathbf{C}\geq 0} f_p(\mathbf{C}) := \frac{1}{2}\|\mathbf{B}-\mathbf{A}\mathbf{C}\|_F^2 + \frac{\lambda}{2}\|\mathbf{C}-\mathbf{C}_*\|_F^2 \ , \tag{2.19}$$

where $\mathbf{C}$ is constrained to be nonnegative. The gradient of f_p is given by the expression

$$\nabla f_p(\mathbf{C}) = (\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{C} - \mathbf{A}^\mathsf{T}\mathbf{B}) + \lambda(\mathbf{C}-\mathbf{C}_*) \ .$$

Updates to $\mathbf{C}$ are performed using the gradient of f_p,

$$\begin{aligned} \nabla f_p(\mathbf{D}_k) &= -\left(\mathbf{A}^\mathsf{T}\mathbf{B} + \lambda\mathbf{C}_*\right) + \left(\mathbf{A}^\mathsf{T}\mathbf{A} + \lambda\mathbf{I}\right)\mathbf{D}_k \\ \mathbf{C}_{k+1} &\leftarrow \left[\mathbf{D}_k - \alpha\nabla f_p(\mathbf{D}_k)\right]_+ \\ \mathbf{D}_{k+1} &\leftarrow \mathbf{C}_{k+1} + \beta_{k+1}\left(\mathbf{C}_{k+1} - \mathbf{C}_k\right) \ , \end{aligned} \tag{2.20}$$

where $[\cdot]_+$ is the projection operator onto $\mathbb{R}_+$. Notice that we can update $\mathbf{C}$ multiple times reusing $\mathbf{A}^\mathsf{T}\mathbf{A}$ and $\mathbf{A}^\mathsf{T}\mathbf{B}$. They are repeated until a termination criteria is triggered; different criteria are discussed in the original paper [52]. The termination criteria are bounds checks on the minimum and absolute maximum values of $\mathbf{C}$.

The selection of hyperparameters λ, α, and β depends on the singular values of $\mathbf{A}$ and is necessary for developing a Nesterov-like method for solving Eq. (2.19). The matrix $\mathbf{C}_*$ is generally $\mathbf{C}$ from the previous outer iteration (Line 2 of Algorithm 1). Details of the selection procedure and different cases can be found in the original paper [52]. The cost to take one step for both factor matrices take at most $O\left(k^2 + (m+n)\,k^2 + (m+n)\,k\right)$ flops.

In addition to the acceleration performed during each NLS solve, Eq. (2.20), we can also perform an acceleration step for every outer iteration in the while loop (Line 2 of Algorithm 1). In this step, all factor matrices are updated using the previous outer iteration values until the objective stops decreasing. The outer acceleration step for iteration t will be

$$\begin{aligned} \mathbf{W}_{\text{new}} &\leftarrow \mathbf{W}^{(t)} + s_t\left(\mathbf{W}^{(t)} - \mathbf{W}^{(t-1)}\right) \\ \mathbf{H}_{\text{new}} &\leftarrow \mathbf{H}^{(t)} + s_t\left(\mathbf{H}^{(t)} - \mathbf{H}^{(t-1)}\right) \end{aligned} \tag{2.21}$$

The results of Eq. (2.21) will be accepted as the next iterate only if the overall objective error with the new factor matrices, $(\mathbf{W}_{\text{new}}, \mathbf{H}_{\text{new}})$, is lower than that of $\left(\mathbf{W}^{(t)}, \mathbf{H}^{(t)}\right)$. In order to compute the relative error, we need an extra function objective computation per outer acceleration. Typically, $s_t = \sqrt{t}$ but its value can change as the overall algorithm progresses [52].

BPP and HALS are exact BCD methods with two and $2k$ blocks, respectively. All other algorithms only approximately solve Eq. (2.9) or approach the solution in the limit. MU, while appearing like a scalar $(m+n)\,k$ block BCD, does not quite solve the objective [27]. In general, most of the update algorithms are sufficient for exploratory data analysis. Providing conclusive rankings is not possible since the performance of each method is dependent on tuning various hyperparameters. This dissertation is primarily concerned with scaling NMF up to large input sizes, and we point the interested reader to other sources to learn more about the convergence and relative performance of different update algorithms [10, 27, 53].

Regularisation on $\mathbf{C}$

Often, we would like to incorporate prior information about the factors into the optimisation problem, Eq. (2.1), resulting in the new regularised formulation,

$$\min_{\mathbf{W}\geq 0, \mathbf{H}\geq 0} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 + \phi\left(\mathbf{W}\right) + \psi\left(\mathbf{H}\right) \ . \tag{2.22}$$

Frobenius-norm regularisation is a popular way to prevent the elements of the factor matrices from growing large in their absolute values. This corresponds, for $\mathbf{H}$ say, to $\psi\left(\mathbf{H}\right) = \alpha \|\mathbf{H}\|_F^2$. Another regularisation is to promote column-wise (or row-wise) sparsity in $\mathbf{H}$ (or $\mathbf{W}$). This regularisation results in $\psi\left(\mathbf{H}\right) = \beta \sum_{j=1}^{n} \|\mathbf{h}_j\|_1^2$ which, is also known as the squared ℓ_{12}- norm. Other regularisations exist [27], but we focus on these two as they can easily be incorporated into the NLS framework described in Section 2.2.2.

Frobenius-norm regularisation: In this case we modify Eq. (2.9) to be

$$\mathbf{C} \leftarrow \operatorname*{argmin}_{\mathbf{C}\geq 0} \|\mathbf{A}\mathbf{C} - \mathbf{B}\|_F^2 + \alpha \|\mathbf{C}\|_F^2 \ , \tag{2.23}$$

which can be recast as

$$\mathbf{C} \leftarrow \operatorname*{argmin}_{\mathbf{C}\geq 0} \left\| \begin{bmatrix} \mathbf{A} \\ \sqrt{\alpha}\mathbf{I}_k \end{bmatrix} \mathbf{C} - \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \right\|_F^2 \ . \tag{2.24}$$

Forming the matrix products for the gradient, $\mathbf{A}^\mathsf{T}\mathbf{A} + \alpha\mathbf{I}_k$ and $\mathbf{A}^\mathsf{T}\mathbf{B}$, we see that only the Gram matrix has to be modified. Thus, by simply adding α along the diagonals of the unmodified Gram matrix we can reuse any of the methods developed in Section 2.2.2.

ℓ_{12}**-norm regularisation:** Similarly, we can modify the Eq. (2.9) for this case.

$$\mathbf{C} \leftarrow \underset{\mathbf{C}\geq 0}{\operatorname{argmin}} \|\mathbf{AC} - \mathbf{B}\|_F^2 + \beta \|\mathbf{C}\|_{1,2}^2 \tag{2.25}$$

$$\mathbf{C} \leftarrow \underset{\mathbf{C}\geq 0}{\operatorname{argmin}} \left\| \begin{bmatrix} \mathbf{A} \\ \sqrt{\beta}\mathbf{1}_{1\times k} \end{bmatrix} \mathbf{C} - \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \right\|_F^2 \tag{2.26}$$

This reformulation too results in only modifying the Gram computation to $\mathbf{A}^{\mathsf{T}}\mathbf{A} + \beta\mathbf{1}_{k\times k}$, which is equivalent to adding β to all entries of unmodified Gram matrix.

2.3 Parallel Algorithms

We switch tack now to deal with *parallelising* and *scaling up* the algorithms for NMF. Early versions of parallel NMF relied on the MapReduce framework [54–58]. These frameworks require data to be written to disk at every iteration and suffer from expensive communication intensive global data shuffles [59]. The first "communication-avoiding" algorithm and library for NMF was proposed by Kannan et al. [35, 59], which later became the PLANC library [36]. The implementations, with a careful choice of data distribution, ensure that once the input matrix is read into memory it is never communicated. Only the smaller factor matrices and other temporaries are exchanged between the processors.

2.3.1 Notation

Throughout the dissertation, we shall be assuming that there are p processors arranged in logical grids of 1, 2, 3, or > 3 dimensions. Subscripts are used to indicate a particular processor in the 1D, 2D, or 3D case, e.g., p_i, p_{ij} and p_{ijk}. For higher dimensions we represent a particular processor by a tuple $\mathbf{p} = (p_1, p_2, \ldots, p_N)$. We assume data items are distributed in a load-balanced across the processors. If there are n items to be distributed, each processor has either $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ items. Let $r = n \mod p$, then the first r processors get $\lceil \frac{n}{p} \rceil$ items and the remaining have $\lfloor \frac{n}{p} \rfloor$ items. However, for a simpler presentation

Table 2.2: Costs of certain MPI collectives.

Operation	Cost
All-Gather	$\alpha \cdot \log p + \beta \cdot \frac{p-1}{p} n$
Reduce-Scatter	$\alpha \cdot \log p + (\beta + \gamma) \cdot \frac{p-1}{p} n$
All-Reduce	$2\alpha \cdot \log p + (2\beta + \gamma) \cdot \frac{p-1}{p} n$

we assume p divides n and each processor contains exactly n/p items unless explicitly specified.

MPI collectives cost

The distributed-memory algorithms we consider here utilise the All-Gather, Reduce-Scatter, and All-Reduce MPI Collectives [19], which we briefly describe here using the $\alpha - \beta - \gamma$ model (see Section 1.2). Our analysis assumes that optimal collective algorithms are used, but our algorithms rely on the underlying MPI implementation [19]. 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. This single collective is a more efficient way of implementing a reduce followed by a scatter. (The reduction can be computed with other associative operators besides addition.) The cost of 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 $2\alpha \cdot \log p + (2\beta + \gamma) \cdot \frac{p-1}{p} n$. The costs of each of the collectives are zero when $p = 1$. These costs are summarised in Table 2.2.

Notions of scalability

We use the metrics of *speedup* and *efficiency* to evaluate parallel algorithms. Let $T(n,p)$ denote the runtime of a parallel algorithm on an input of size n on p processors. The speedup is defined as

$$\text{Speedup}(n,p) = \frac{T_{\text{seq}}(n)}{T(n,p)}$$

where $T_{\text{seq}}(n)$ is the best sequential algorithm or implementation. This is sometimes referred to as absolute speedup due to the requirement of using the best sequential algorithm. *Relative speedup* is the comparison of the runtimes against using the same algorithm on a single processor, i.e.,

$$\text{Relative-Speedup}(n,p) = \frac{T(n,1)}{T(n,p)}\ .$$

Another measure of parallel efficiency is

$$\text{Efficiency}(n,p) = \frac{T_{\text{seq}}(n)}{pT(n,p)} = \frac{\text{Speedup}(n,p)}{p}\ .$$

This measure quantifies how well the processors are utilised by the parallel algorithm. If relative speedup is used instead of absolute speedup in the formula, we term this measure relative efficiency. We analyse our algorithms in two ways:

1. *Strong scaling*: Here the input problem size n is held fixed and we vary the number of processors p.

2. *Weak scaling*: The per-processor problem size $\frac{n}{p}$ is held fixed while varying p.

2.3.2 Parallel Matrix Multiplication

From Section 2.2, we can see that the computational complexity of a single outer iteration of NMF (i.e. Line 2 of Algorithm 1) can be split into two blocks.

$$T_{\text{NMF}} = T_{\text{matmuls}} + T_{\text{nls}} \tag{2.27}$$

$$T_{\text{NMF}} \in \begin{cases} O\left(mnk + (m+n)\,k^2\right) + T_{\text{nls}}\left(k, m+n\right) & \mathbf{X} \text{ is dense} \\ O\left(\text{nnz}(\mathbf{X})k + (m+n)\,k^2\right) + T_{\text{nls}}\left(k, m+n\right) & \mathbf{X} \text{ is sparse} \end{cases} \tag{2.28}$$

Here, $T_{\text{nls}}(a,b)$ denotes the time taken to update an $a \times b$ matrix. In general, we have $T_{\text{nls}}\left(k, m+n\right) \in O\left(k^3 + (m+n)\,k^2 + (m+n)\,k\right)$, for all the NLS updates discussed in Section 2.2.2. We assume that BPP is in the average case scenario and does not take 2^k flops. Since we are in the CLRA setting, we can assume $k \ll \min\left(m,n\right)$. We can see that the $O\left(mnk\right)$ cost is the bottleneck computation for the dense case. The $O\left(\text{nnz}(\mathbf{X})k\right)$ term also dominates for the sparse case if $\text{nnz}(\mathbf{X}) > (m+n)\,k$. This cost is associated with the large matrix multiplies involving the input matrix $\mathbf{X}$, namely $\mathbf{W}^{\mathsf{T}}\mathbf{X}$ and $\mathbf{H}\mathbf{X}^{\mathsf{T}}$. Employing a state-of-the-art parallel matrix multiplication algorithm would significantly improve the running time of NMF.

Matrix multiplication is one of the most fundamental computational kernels, and its communication complexity has been studied extensively in both the sequential and parallel settings [60–62]. These results establish asymptotic lower bounds for matrix multiplication, which were realised to be tight with the development of algorithms achieving them [63, 64]. We provide a quick overview of this area and highlight the cases that apply to matrix multiplications in NMF. These results are shown for dense classic matrix multiplication; similar bounds are known for "fast" algorithms (e.g., Strassen's method), but we assume standard or classic matrix multiplications [8]. For sparse computations these lower bounds do not apply.

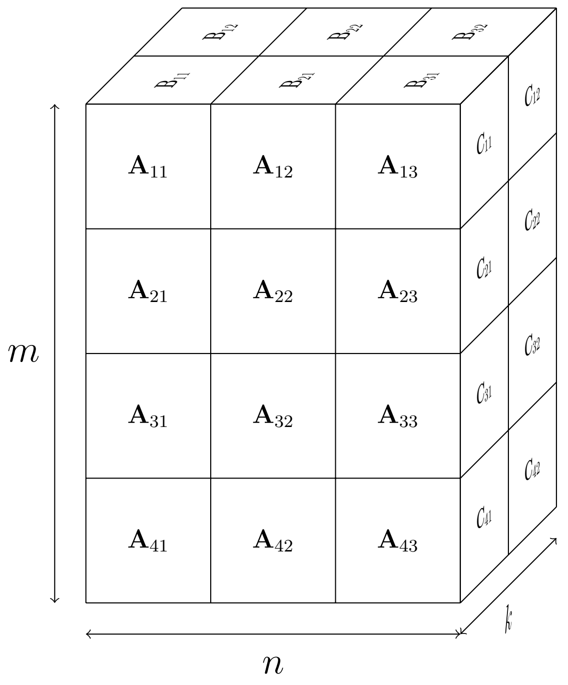


Figure 2.7: Visualisation of the iteration space of classical matrix multiplication for computing $\mathbf{C} = \mathbf{AB}$ with $\mathbf{C} \in \mathbb{R}^{m\times k}, \mathbf{A} \in \mathbb{R}^{m\times n}$, and $\mathbf{B} \in \mathbb{R}^{n\times k}$. The prism of mnk scalar multiplications is arranged in a $4 \times 3 \times 2$ processor grid.

Lower Bounds

There are two types of lower bounds studied for matrix multiplication. The first is the "memory-dependent" kind where we assume that each of the processors have a cache of size M. Consider the case of multiplying two square $n \times n$ matrices. Irony et al. reproduce an earlier result that $\Omega\left(\frac{n^3}{\sqrt{M}}\right)$ transfers occur between disk and cache [62]. They extended this result to p processors as requiring $\Omega\left(\frac{n^3}{p\sqrt{M}}\right)$ transfers between the processors and to rectangular matrix multiplications (replacing n^3 by mnk for a $m \times n$ by $n \times k$ multiply).

The second kind of lower bounds, also shown by Irony et al., are known as "memory-independent" bounds. In this case the factor M does not appear in the bounds. They show that a parallel algorithm for matrix multiplications must communicate $\Omega\left(\frac{n^2}{p^{2/3}}\right)$ words under the reasonable assumption of computational load balance. Interestingly, when $p \gg \frac{n^3}{M^{3/2}}$ the memory-dependent bounds no longer apply since the memory-independent ones are larger. Demmel et al. extend these results to general rectangular case and describe the different regimes of computation [63]. Daas et al. tightened these lower bounds and provide explicit constants for the leading terms and unify the different results under a single optimisation framework [64, 65].

The lower bounds were developed using geometric arguments on the “prism” of mnk scalar multiplications needed by a classic matrix multiplication algorithm (see Fig. 2.7). To perform a multiplication, a processor must have access to the corresponding entries on the $m \times n$, $n \times k$, and $m \times k$ faces of the prism. Otherwise, it needs to communicate data to perform the multiply. Relating the volume of this iteration space “cuboid” with respect to the areas of its faces allow us to develop lower bounds for the computation [62–65].

Theorem 2.3.1 ([65, Theorem 1])**.** *Consider a classical matrix multiplication involving matrices of size $n_1 \times n_2$ and $n_2 \times n_3$. Let $m = \max(n_1, n_2, n_3)$, $n = \operatorname{median}(n_1, n_2, n_3)$, and $k = \min(n_1, n_2, n_3)$, so that $m \geq n \geq k$. Any parallel algorithm using p processors that starts with one copy of the two input matrices and ends with one copy of the output matrix and load balances either the computation or the data must communicate at least*

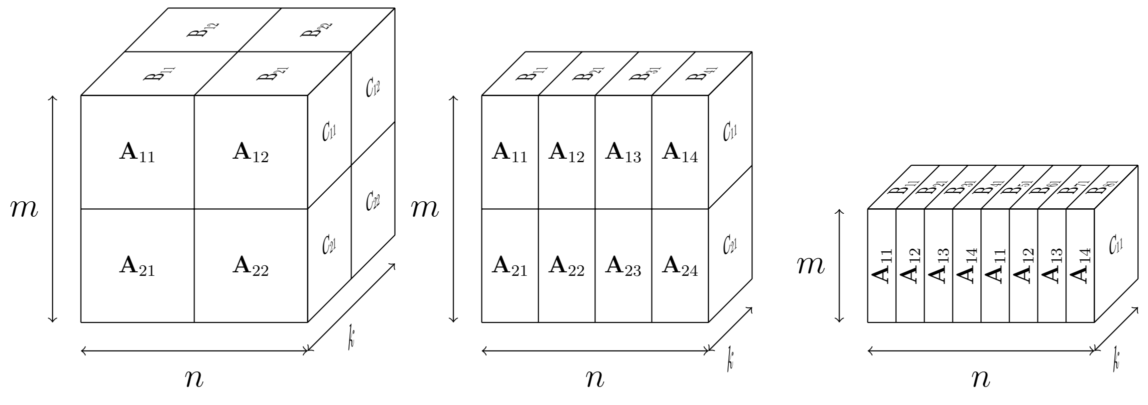
$$D - \frac{mn + nk + mk}{p}$$

words of data, where

$$D = \begin{cases} \frac{mn+mk}{p} + nk & \text{if } 1 \leq p \leq \frac{m}{n} \\ 2\left(\frac{mnk^2}{p}\right)^{1/2} + \frac{mn}{p} & \text{if } \frac{m}{n} \leq p \leq \frac{mn}{k^2} \\ 3\left(\frac{mnk}{p}\right)^{2/3} & \text{if } \frac{mn}{k^2} \leq p \end{cases} .$$

Optimal Algorithms

3D matrix multiplication algorithms assign the computation space (Fig. 2.7) to processors according to their location in the grid [63, 66]. Visualising the algorithm as this iteration



(a) Three large dimensions. (b) Two large dimensions. (c) One large dimension.

Figure 2.8: Examples of the three different aspect ratios of a processor grid with $p = 8$. In (a) all the dimensions are split, only m and n are split in (b), and finally in (c) only n is divided. In the last two cases, only one processor needs access to the largest matrix's entries. Therefore, with the correct data layout the matrix multiplication algorithm will only need to transfer the smaller matrices.

space prism also allows us to realise that there are three different regimes depending on the aspect ratios, i.e., the ratio of number of rows to columns, of the matrices involved. Demmel et al. classify these as the one-large, two-large, and three-large dimension cases [63]. By large dimension, they mean that it benefits the algorithm to split up that dimension. They are also known as 3D, 2D, and 1D algorithms based on the dimension of the processor grid involved. They show that optimal algorithms, which attain the lower bound, can be realised by carefully partitioning the data layouts to make each sub-prism as cubical as possible (see Fig. 2.8).

The optimal parallel algorithm (Algorithm 2) and its visualisation (Fig. 2.9) are shown below [64]. The algorithm is written in the style of MPI programs, which is used throughout the dissertation. The pseudocode in Algorithm 2 is the set of instructions executed in parallel by every processor $\mathbf{p}$ in the grid. Synchronisations are handled either by the MPI collective calls or by the user explicitly. The notation $(p_1, :, p_3)$ is used to refer to the slice of the processor grid keeping p_1 and p_3 constant. For a comprehensive review of this topic, we direct the reader to following sources [62–64].

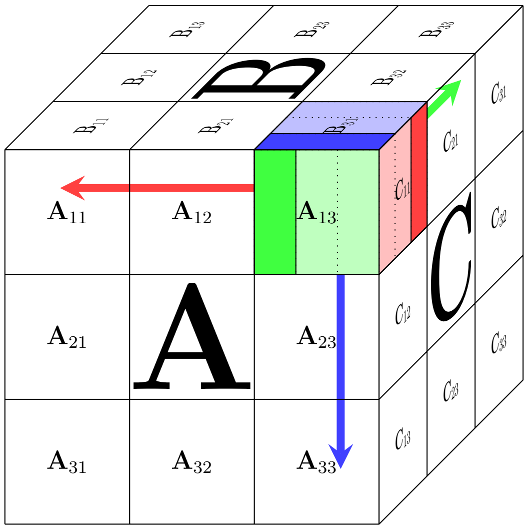


Figure 2.9: Visualisation of Matrix multiply (Algorithm 2) with a $3 \times 3 \times 3$ processor grid. The 3D iteration space is mapped onto the processor grid, and the matrices are mapped to the faces of the grid. The dark highlighting corresponds to the input data initially owned and the output data finally owned by processor $(1, 3, 1)$, and the light highlighting signifies the data of other processors it uses to perform the local computation. The arrows show the sets of processors involved in the three collective operations involving processor $(1, 3, 1)$.

Algorithm 2 Communication-optimal parallel matrix multiplication

```
Require: $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2}$ and $\mathbf{B} \in \mathbb{R}^{n_2 \times n_3}$.
Ensure: $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{n_1 \times n_3}$
  1: $\mathbf{p} = (p_1, p_2, p_3)$                                        ▷ Processor rank
  2:                                   ▷ Gather input matrices needed for multiply
  3: $\mathbf{A}_{p_1,p_2} = $ All-Gather $(\mathbf{A}_{p1,p2,p3}, (p_1, p_2, :))$
  4: $\mathbf{B}_{p_2,p_3} = $ All-Gather $(\mathbf{B}_{p1,p2,p3}, (:, p_2, p_3))$
  5:                                          ▷ Perform the local matrix multiply
  6: $\hat{\mathbf{C}}_{p_1,p_2,p_3} = \mathbf{A}_{p_1,p_2}\mathbf{B}_{p_2,p_3}$
  7:                                             ▷ Sum the rest of the parts of C
  8: $\mathbf{C}_{p_2,p_3} = $ Reduce-Scatter $\left(\hat{\mathbf{C}}_{p1,p2,p3}, (p_1, :, p_3)\right)$
```

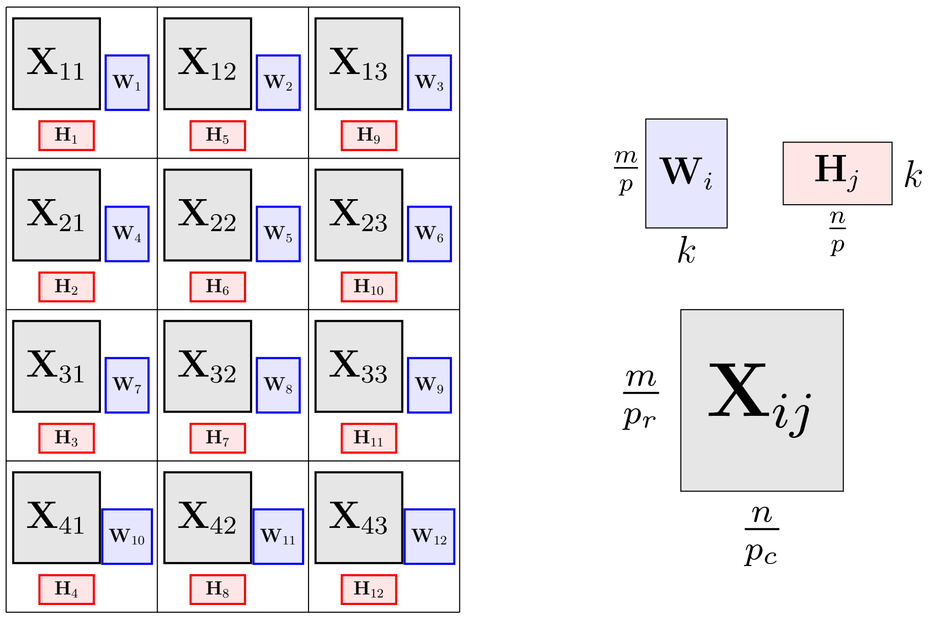
Now let us categorise the different matrix multiplications arising in NMF: $\mathbf{W}^{\mathsf{T}}\mathbf{W}$, $\mathbf{H}\mathbf{H}^{\mathsf{T}}$, $\mathbf{W}^{\mathsf{T}}\mathbf{X}$, and $\mathbf{H}\mathbf{X}^{\mathsf{T}}$ (see Eqs. (2.10) and (2.11)). Note that for NMF, and for CLRA in general, we have $k \ll \min(m, n)$. From these discussions, it is clear that the Gram matrix computations ($\mathbf{W}^{\mathsf{T}}\mathbf{W}$ and $\mathbf{H}\mathbf{H}^{\mathsf{T}}$) fall under the one-large dimension category. The other multiplies, $\mathbf{W}^{\mathsf{T}}\mathbf{X}$ and $\mathbf{H}\mathbf{X}^{\mathsf{T}}$, are in the two-large dimensions case. Notice that both these cases be designed in a manner such that the largest matrices participating in the multiplications are never communicated (see Fig. 2.8). In the next section we go into the details of designing such an algorithm for NMF.

2.3.3 2D NMF Algorithms

The PLANC software package is designed to utilise the communication-optimal matrix multiplication algorithm for each of the four different multiplies encountered in NMF. Furthermore, it is also able to exploit the parallelism afforded by the NLS subproblems by separating out the independent right-hand sides across the processors.

Data Distributions

Let us first look at the how the three different matrices involved in NMF are distributed. Since we are working in the one-large dimension and two-large dimensions regime we need a logical 2D grid of processors. Let us assume we have $p = p_r \times p_c$ grid of processors to work with. The larger dimensions m and n are split between the processors whereas the low-rank dimension k is kept intact. Thus, the input matrix $\mathbf{X}$ is 2D distributed across the processors, whereas $\mathbf{W}$ is distributed in a block-row manner and $\mathbf{H}$ in a block column (since their other dimension is of size k). The choice for p_r and p_c is to make the input $\mathbf{X}$ as square as possible to be as communication efficient as possible. Therefore, we pick $\frac{p_r}{p_c} \approx \frac{m}{n}$. We analytically justify this choice when we calculate the computational and communication costs of this distributed matrix multiplication.



$\mathbf{X}_{11}$ $\mathbf{W}_1$ $\mathbf{H}_1$
$\mathbf{X}_{12}$ $\mathbf{W}_2$ $\mathbf{H}_5$
$\mathbf{X}_{13}$ $\mathbf{W}_3$ $\mathbf{H}_9$
$\mathbf{X}_{21}$ $\mathbf{W}_4$ $\mathbf{H}_2$
$\mathbf{X}_{22}$ $\mathbf{W}_5$ $\mathbf{H}_6$
$\mathbf{X}_{23}$ $\mathbf{W}_6$ $\mathbf{H}_{10}$
$\mathbf{X}_{31}$ $\mathbf{W}_7$ $\mathbf{H}_3$
$\mathbf{X}_{32}$ $\mathbf{W}_8$ $\mathbf{H}_7$
$\mathbf{X}_{33}$ $\mathbf{W}_9$ $\mathbf{H}_{11}$
$\mathbf{X}_{41}$ $\mathbf{W}_{10}$ $\mathbf{H}_4$
$\mathbf{X}_{42}$ $\mathbf{W}_{11}$ $\mathbf{H}_8$
$\mathbf{X}_{43}$ $\mathbf{W}_{12}$ $\mathbf{H}_{12}$
$\frac{m}{p}$ $\mathbf{W}_i$ k
$\mathbf{H}_j$ k $\frac{n}{p}$
$\frac{m}{p_r}$ $\mathbf{X}_{ij}$ $\frac{n}{p_c}$

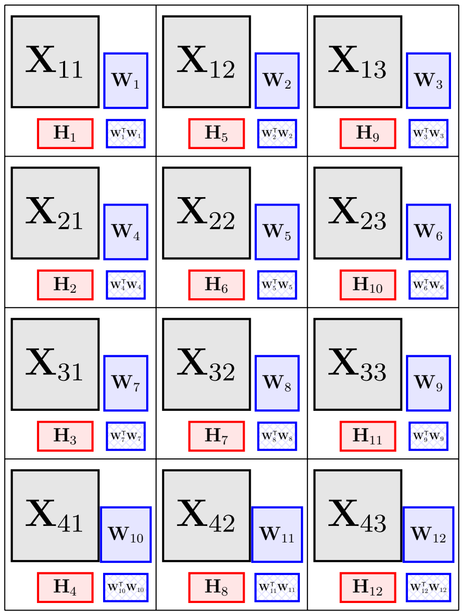
Figure 2.10: PLANC data distribution of the NMF matrices on a 4×3 processor grid. The large $m \times n$ input matrix $\mathbf{X}$ is 2D partitioned across the grid, whereas as the factor matrices are 1D partitioned. The rows of $\mathbf{W}$ are placed in row-major order and the columns of $\mathbf{H}$ in column-major order across the grid. The smallest dimension k is never split.

Fig. 2.10 shows the data distribution for NMF in PLANC. Every processor possesses an $\frac{m}{p_r} \times \frac{n}{p_c}$ block of the input matrix $\mathbf{X}$. Since the column dimension of $\mathbf{W}$ is k and considered to be small, we block-distribute the rows of $\mathbf{W}$ across the entire grid in a row-major order. Each processor has a $\frac{m}{p}$ contiguous block of rows from $\mathbf{W}$. The columns of $\mathbf{H}$ are distributed analogously in a column-major order across the grid. Each processor has a $k \times \frac{n}{p}$ column-block of $\mathbf{H}$. The reason for the two different orderings is because we apply $\mathbf{W}^\mathsf{T}$ to $\mathbf{X}$ whereas $\mathbf{H}$ is applied on the transpose of input matrix, i.e., $\mathbf{X}^\mathsf{T}$.

Parallel matrix multiplications in PLANC

Let us describe the two different types of matrix multiplications in detail.

Gram calculations: The Gram matrix calculations of $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}\mathbf{H}^\mathsf{T}$ are used to form the LHS of the normal equations, $\mathbf{A}^\mathsf{T}\mathbf{A}$, of the NLS update algorithms (see Section 2.2.2). The sizes of the matrices involved in the computation are $k \times m$, and $m \times k$, resulting in


(a) Every processor computes its local $\mathbf{W}_i^\mathsf{T}\mathbf{W}_i$.

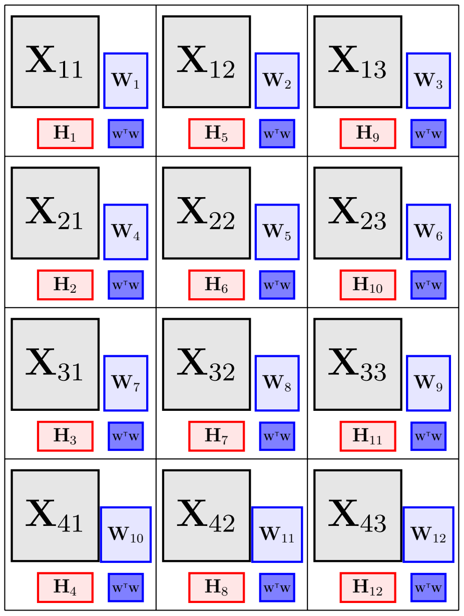
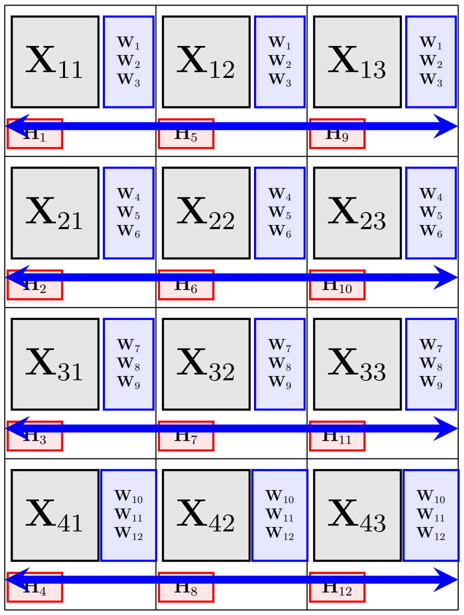
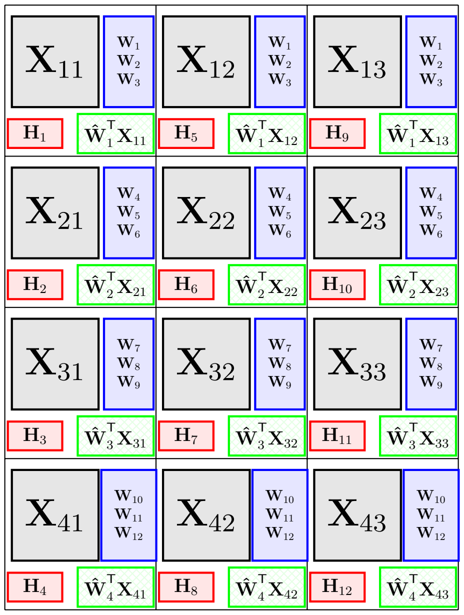

(b) All-Reduce to compute the full $\mathbf{W}^\mathsf{T}\mathbf{W}$.

Figure 2.11: Parallel Gram computation for computing $\mathbf{W}^\mathsf{T}\mathbf{W}$. All processors redundantly store the $k \times k$ Gram matrix $\mathbf{W}^\mathsf{T}\mathbf{W}$ in the end. Computing $\mathbf{H}\mathbf{H}^\mathsf{T}$ is the exact same algorithm, and it is redundantly stored as well.

a $k \times k$ matrix. This is a one-large dimension category problem under Demmel's taxonomy [63]. Therefore, we should be able to compute it without communicating any of the larger matrices. The pictorial description of the multiplication is shown in Fig. 2.11. Each processor computes its local Gram matrix, either $\mathbf{W}_i^\mathsf{T}\mathbf{W}_i$ or $\mathbf{H}_j\mathbf{H}_j^\mathsf{T}$, to form a $k \times k$ partial Gram matrix (see Fig. 2.11a). This computation is followed by an All-Reduce of the partial Gram matrix to compute and redundantly store the $k \times k$ Gram matrix on all the processors. Redundant storage is required for each of the processors to have access to the LHS of the independent NLS subproblems. Once each processor's RHS portion is computed, they can update their portions of the factor matrices in parallel. For handling regularisation as described in Section 2.2.2, each processor can simply add $\alpha\mathbf{I}_k$ or $\beta\mathbf{1}_{k\times k}$ locally to the Gram matrices once they are computed.

Multiplication with X: We use the 2D algorithm to compute $\mathbf{W}^\mathsf{T}\mathbf{X}$ and $\mathbf{H}\mathbf{X}^\mathsf{T}$. This product corresponds to computing $\mathbf{A}^\mathsf{T}\mathbf{B}$ in the updating algorithms for the NLS subprob-


(a) All-Gather stage.


(b) Every processor computes its local $\hat{\mathbf{W}}_i^\top \mathbf{X}_{ij}$.

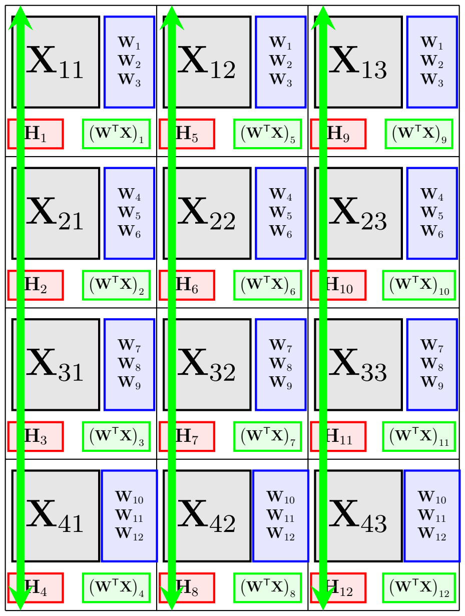

(c) Reduce-Scatter to compute the full $\mathbf{W}^\top\mathbf{X}$.

Figure 2.12: Parallel multiplication involving $\mathbf{X}$. We show the case for $\mathbf{W}^\top\mathbf{X}$.

lems Eq. (2.9). To compute $\mathbf{W}^\mathsf{T}\mathbf{X}$, we first gather all the parts of $\mathbf{W}$ needed for the local matrix multiply. This All-Gather operation across the processor rows (Fig. 2.12a) combines the rows of $\mathbf{W}$ as follows

$$\hat{\mathbf{W}}_i = \begin{bmatrix} \mathbf{W}_{(i-1)p_c+1} \\ \mathbf{W}_{(i-1)p_c+2} \\ \vdots \\ \mathbf{W}_{(i-1)p_c+p_c} \end{bmatrix}.$$

Then each processor computes its local product, shown as $\hat{\mathbf{W}}_i^\mathsf{T}\mathbf{X}_{ij}$ in Fig. 2.12b. Next to compute the full $\mathbf{W}^\mathsf{T}\mathbf{X}$ we Reduce-Scatter the partial products across the processor columns (Fig. 2.12c). The distribution of $\left(\mathbf{W}^\mathsf{T}\mathbf{X}\right)_i$, which are block rows of the product, is in column-major order across the grid. This ordering matches the distribution of $\mathbf{H}$, and we can now update its block columns as we have the RHS needed to update $\mathbf{H}_j$ block each processor owns. The operation for $\mathbf{H}\mathbf{X}^\mathsf{T}$ is analogous on the transposed grid. The gathers happen across the processor columns and the scatters across the rows.

Parallel NMF

Now we are able to describe the parallel algorithm for two-block NMF. We denote the NLS update algorithm to update $\mathbf{C}$ given $\mathbf{A}^\mathsf{T}\mathbf{A}$ and $\mathbf{A}^\mathsf{T}\mathbf{B}$ by $\mathrm{update}\left(\mathbf{A}^\mathsf{T}\mathbf{A}, \mathbf{A}^\mathsf{T}\mathbf{B}\right)$ (see Section 2.2.2 for more details). Algorithm 3 shows the NMF algorithm employing the communication-avoiding matrix multiplications described in the previous section, along with parallelising over the independent RHS in the NLS subproblems.

Computational Cost: We can now analyse the cost of a single outer iteration Line 5 of Algorithm 3 for a single processor. Let us use $T_{\text{nls}}\left(a, b\right)$ to denote the time taken to update a matrix of size $a \times b$ using one of the NLS update algorithms. The Gram computations, Lines 6 and 12 of Algorithm 3, take $\frac{(m+n)k^2}{p}$ flops. The RHS computations, Lines 9 and 15, take $\frac{4mn}{p_r p_c}k = \frac{4mnk}{p}$ flops. This cost changes to $4\frac{\mathrm{nnz}(\mathbf{X})}{p}k$ flops in the sparse case.

Algorithm 3 Communication-avoiding parallel NMF [59]

```
Require: $\mathbf{X} \in \mathbb{R}_+^{m \times n}$ 2D distributed across a $p_r \times p_c$ processor grid.
Require: Low-rank parameter $k$.
Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\mathbf{W} \geq 0, \mathbf{H} \geq 0}{\operatorname{argmin}} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2$ .
Ensure: $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ is row-wise distributed across the grid.
Ensure: $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ is column-wise distributed across the grid.
 1: $\mathbf{p} = (i, j)$                                        ▷ Processor rank
 2: $r = (i-1)\, p_c + j$                                     ▷ Row-major rank
 3: $c = (j-1)\, p_r + i$                                     ▷ Column-major rank
 4: Initialise $\mathbf{W}_r^{(0)}$ and $\mathbf{H}_c^{(0)}$
 5: while $t = 1, 2, \ldots$ do                         ▷ Till some stopping condition is met.
        % Compute W given H
 6:     $\hat{\mathbf{G}}_{\mathbf{H}_c} = \mathbf{H}_c^{(t)} \left(\mathbf{H}_c^{(t)}\right)^\mathsf{T}$          ▷ Gram computation.
 7:     $\mathbf{G}_\mathbf{H}$ = All-Reduce($\hat{\mathbf{G}}_{\mathbf{H}_c}$)
 8:     $\hat{\mathbf{H}}_j^{(t)}$ = All-Gather($\mathbf{H}_c^{(t)}$,$(:, j)$)          ▷ RHS computation.
 9:     $\hat{\mathbf{R}}_{\mathbf{H}_c} = \hat{\mathbf{H}}_j^{(t)} \mathbf{X}_{ij}^\mathsf{T}$
10:     $\mathbf{R}_{\mathbf{H}_c}$ = Reduce-Scatter($\hat{\mathbf{R}}_{\mathbf{H}_c}$, $(i, :)$)
11:     $\mathbf{W}_r^{(t)}$ = update($\mathbf{G}_\mathbf{H}, \mathbf{R}_{\mathbf{H}_c}$)          ▷ NLS update.
        % Compute H given W
12:     $\hat{\mathbf{G}}_{\mathbf{W}_r} = \left(\mathbf{W}_r^{(t)}\right)^\mathsf{T} \mathbf{W}_r^{(t)}$          ▷ Gram computation.
13:     $\mathbf{G}_\mathbf{W}$ = All-Reduce($\hat{\mathbf{G}}_{\mathbf{W}_r}$)
14:     $\hat{\mathbf{W}}_i^{(t)}$ = All-Gather($\mathbf{W}_r^{(t)}$,$(i, :)$)          ▷ RHS computation.
15:     $\hat{\mathbf{R}}_{\mathbf{W}_r} = \left(\hat{\mathbf{W}}_i^{(t)}\right)^\mathsf{T} \mathbf{X}_{ij}$
16:     $\mathbf{R}_{\mathbf{W}_r}$ = Reduce-Scatter($\hat{\mathbf{R}}_{\mathbf{W}_r}$, $(:, j)$)
17:     $\mathbf{W}_r^{(t)}$ = update($\mathbf{G}_\mathbf{W}, \mathbf{R}_{\mathbf{W}_r}$)          ▷ NLS update.
18: end while
```

There an additional $\left(\frac{m}{p_r}+\frac{n}{p_c}\right)k$ flops arising from the Reduce-Scatters (Lines 10 and 16). There are two NLS updates of $k\times\frac{m}{p}$ and $k\times\frac{n}{p}$ matrices, which combined take $T_{\text{nls}}\left(k,\frac{m+n}{p}\right)$ times (Lines 11 and 17).

Thus, the per iteration flop counts are

$$T_{\text{comp}}\left(\text{NMF}\right)\in\begin{cases}O\left(\frac{mnk+(m+n)k^2+(p_cm+p_rn)k}{p}\right)+T_{\text{nls}}\left(k,\frac{m+n}{p}\right) & \mathbf{X}\text{ is dense}\\ O\left(\frac{\text{nnz}(\mathbf{X})k+(m+n)k^2+(p_cm+p_rn)k}{p}\right)+T_{\text{nls}}\left(k,\frac{m+n}{p}\right) & \mathbf{X}\text{ is sparse}\end{cases}\tag{2.29}$$

Comparing to Eq. (2.28), we can see a reduction of costs by a factor of p.

Communication cost: There are six collective calls in Algorithm 3. We use the costs associated with them from Table 2.2 to calculate the per iteration communication costs. The costs of the All-Reduce operations (Lines 7 and 13) are $4\alpha\log p+2\beta k^2$. The All-Gathers of Lines 8 and 14 cost $2\alpha\log p+\beta\left(\frac{mk}{p}\left(p_c-1\right)+\frac{nk}{p}\left(p_r-1\right)\right)$. The Reduce-Scatters cost $2\alpha\log p+\beta\left(\frac{mk}{p}\left(p_c-1\right)+\frac{nk}{p}\left(p_r-1\right)\right)$.

We can assume without loss of generality $m\geq n$. These costs can be classified into two cases. If $m/p<n$ we can choose $\frac{m}{p_r}\approx\frac{n}{p_c}\approx\sqrt{mn/p}$, which simplifies the communication cost to $\alpha\cdot O(\log p)+\beta\cdot O(k^2+\frac{mk}{p_r}+\frac{nk}{p_c})=\alpha\cdot O(\log p)+\beta\cdot O\left(k^2+\sqrt{mnk^2/p}\right)$. In the tall-skinny case, $m/p>n$, we can set $p_c=1$, and the costs simplify to $\alpha\cdot O(\log p)+\beta\cdot O(nk)$. This choice of grid gives a total communication cost of

$$T_{\text{comm}}\left(\text{NMF}\right)=\begin{cases}\alpha\cdot O(\log p)+\beta\cdot O\left(\sqrt{mnk^2/p}\right) & m/p<n\\ \alpha\cdot O(\log p)+\beta\cdot O\left(nk\right) & m/p>n\end{cases}.\tag{2.30}$$

Note that we assume the NLS update algorithm does not communicate here. This assumption is not true for some algorithms like, for example HALS.

Comparing Eq. (2.30) with Theorem 2.3.1 we can see that that Algorithm 3 achieves the lower bounds for both the cases (1D or 2D distribution of $\mathbf{X}$). Notice also that unlike computation, communication scales only as a factor $\sqrt{p}$ when $\mathbf{X}$ is 2D distributed and is not affected by p in the 1D case.

Memory requirements: The local memory requirements for each processor is mn/p entries of $\mathbf{X}$ (or $\mathrm{nnz}(\mathbf{X})/p$ in the sparse case), mk/p entries of $\mathbf{W}$ and nk/p entries of $\mathbf{H}$. The temporary matrices for computing the RHS products consume $2\left(\frac{mk}{p_r}+\frac{nk}{p_c}\right)$ words. In the dense case, if we assume $k < \min(m/p_r, n/p_c)$ then the local memory requirements is just a constant times the size of the input matrix. For optimal choices of p_r and p_c we can simplify this assumption to $k < \max\left(\sqrt{mn/p}, m/p\right)$.

2.3.4 Experiments

We present a few experiments to demonstrate the parallel performance of Algorithm 3. These are shown to give a basic idea of how the different components of the algorithm behave. We defer more thorough studies with respect to different updating algorithms and other configurations to later chapters.

Experimental Setup

All of our experiments at scale were carried out on Phoenix, a supercomputer hosted at Georgia Tech [67]. Each of the 852 compute nodes of Phoenix, in the basic node class, contains two 2.7 GHz Intel® Xeon® 6226 12-core processors, one per socket. Every node has a main memory of 192 GB (DDR4-2,933 MHz) and are interconnected via a HDR100 Infiniband interconnect.

PLANC uses Armadillo for matrix operations [68]. Armadillo stores dense matrices in column-major order and sparse matrices in the Compressed Sparse Column (CSC) format. We link Armadillo (version 11.2.3) with OpenBLAS (0.3.13) for dense BLAS and LAPACK operations and compile using GNU C++ version 8.3.0. All our scaling operations are conducted in the flat MPI setting. By "flat" we mean that each core is assigned to a different MPI process as opposed to assigning a single MPI process to a socket or node and enabling multithreading within the socket or node. This sample experiment was run on two synthetic inputs. Dense matrices are generated as nonnegative low-rank matrices by multiplying a randomly generated $\mathbf{W}$ and $\mathbf{H}$. Sparse matrices are generated in a uniform random manner with a density of 0.05.

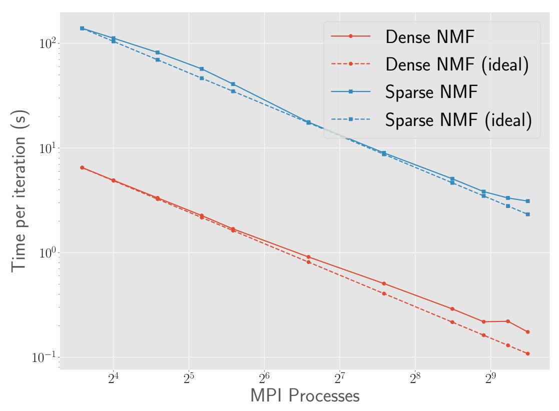
We capture different stages of the algorithm in our experiments. We divide the time into computation and communication sections with three different parts in each. The local matrix multiplications with $\mathbf{X}$ are grouped together as the category `Matmul`. The Gram computations and NLS updates, for both $\mathbf{W}$ and $\mathbf{H}$, are called `Gram` and `NLS` respectively. These three form the computation section. The three collective communications, All-Reduce, All-Gather, and Reduce-Scatter, are individually timed. For this sample experiment we only run the BPP update algorithm, therefore, there isn't any communication involved in the NLS updates.

We vary the number of processors from 12 to 720 which is from 1 node to 30 nodes of Phoenix. We use a single socket of a node as the starting pointing of scaling. The different grid configurations used for the scaling studies are 4×3, 4×4, 6×4, 9×4, 8×6, 16×6, 16×12, 30×12, 30×16, 30×20, and 30×24. The NMF algorithm is run 5 times for 10 outer iterations in the dense case (5 for sparse) and we average the different categories to get the per-iteration timings.

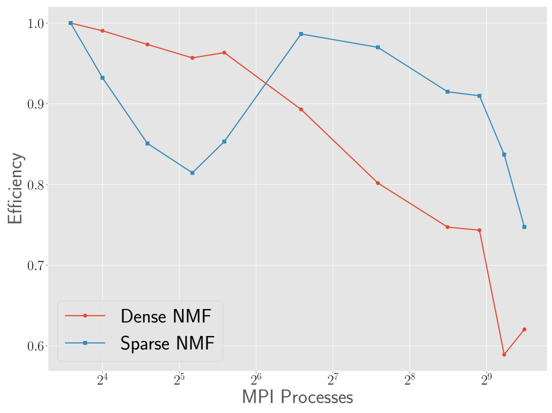
Strong Scaling

The strong-scaling results are presented in Figs. 2.13 to 2.15. We use synthetic matrices of dimensions 140,000$\times$70,000 for the dense case and 440,000$\times$220,000 for sparse. We use a low rank parameter of 50, which is a typical size for CLRA.

Fig. 2.13 show the runtime (Fig. 2.13a) and relative efficiency (Fig. 2.13b) for both the sparsity types. The dense case runs an order of magnitude faster than the sparse case. We can see the timing reduce as we increase p but not gracefully. Efficiency shoots up for both

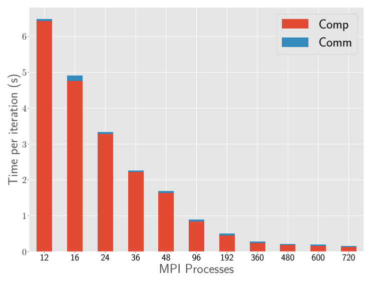


(a) Strong scaling.

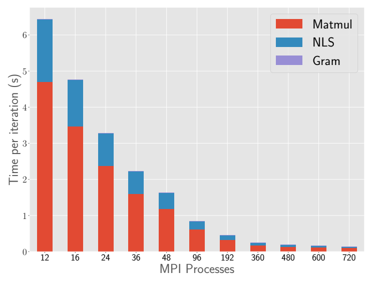


(b) Scaling efficiency.

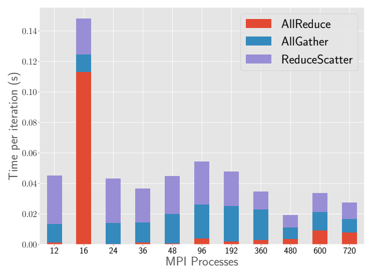
Figure 2.13: Strong scaling for both the dense ($140{,}000 \times 70{,}000$) and sparse ($440{,}000 \times 220{,}000$) cases with $k = 50$. Erratic but high efficiency observed.



(a) Total breakdown.

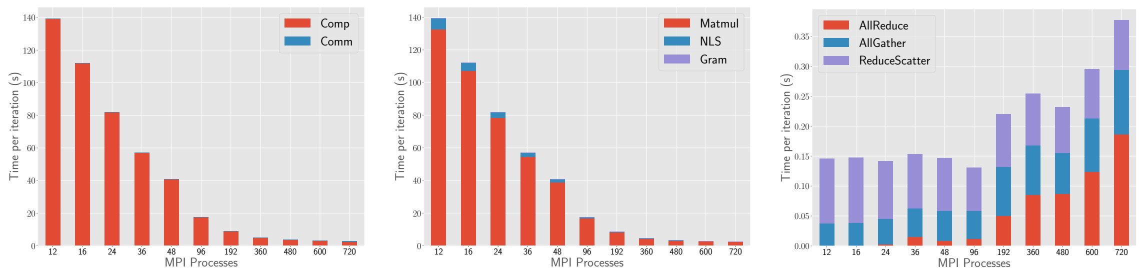


(b) Computation breakdown.



(c) Communication breakdown.

Figure 2.14: Strong-scaling timing breakdown for the dense case with $m = 140{,}000$, $n = 70{,}000$, and $k = 50$. Multiplication with $\mathbf{X}$ is the dominating cost but NLS also appears.



(a) Total breakdown. (b) Computation breakdown. (c) Communication breakdown.

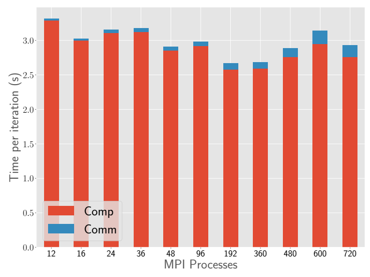
Figure 2.15: Strong-scaling timing breakdown for sparse case with $m = 440{,}000, n = 220{,}000$, and $k = 50$. Multiplication with $\mathbf{X}$ is the dominating cost.

cases when we scale from 36 to 48 processes. Efficiency remains high as we scale to 960 processes with 60% efficiency for the dense case and 75% for sparse. However, we are using *relative* speedup and efficiency in this setting. Therefore, it is not guaranteed that we have the best sequential algorithm.

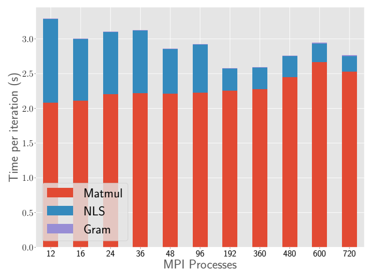
Figures 2.14 and 2.15 show the breakdown of the running times in the strong-scaling experiments. The algorithm is heavily dominated by the computation component. The majority of the runtime is spent on the matrix multiplication with $\mathbf{X}$ as expected (see Figs. 2.14b and 2.15b). The computation dominates more in the sparse case with communication barely registering in the breakdown (see Fig. 2.15a). The computational parts of the algorithm scale with p whereas the communication parts scale less well (Figs. 2.14c and 2.15c). An anomaly is the 4×4 grid in the dense case (Fig. 2.14). It has an unusually large All-Reduce component ($10\times$ the norm) but doesn't affect the overall timings due to the dominance of matrix multiplication costs.

Weak Scaling

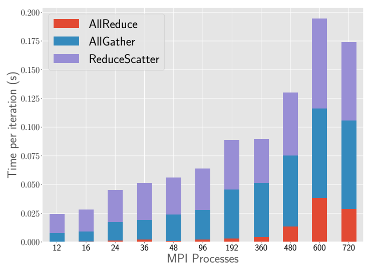
The weak-scaling experiments in Figs. 2.16 and 2.17 were conducted by keeping the memory per processor constant. For the dense case, we start with $m = 96{,}000$ and $n = 45{,}000$ for 12 processors and scale to $m = 720{,}000$ and $n = 360{,}000$ for 720 processors. For the sparse case, these sizes vary from $304{,}000 \times 142{,}500$ to $2{,}280{,}000 \times 1{,}140{,}000$. Since the ratios of $\frac{m}{p_r}$ and $\frac{n}{p_c}$ are kept constant, the local matrix dimension sizes the same. Thus the



(a) Total breakdown.

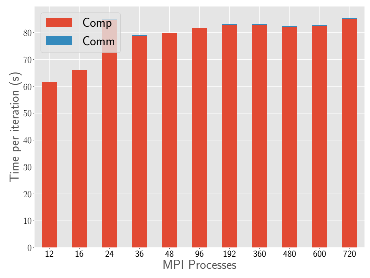


(b) Computation breakdown.

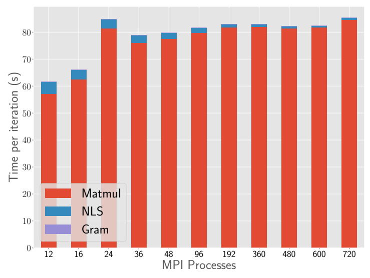


(c) Communication breakdown.

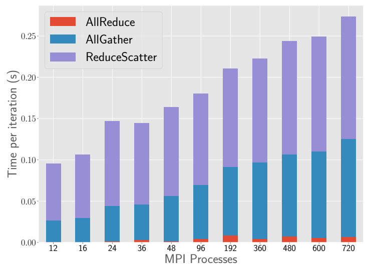
Figure 2.16: Weak-scaling timing breakdowns for the dense case. Here $m/p_r = 24{,}000$ and $n/p_c = 15{,}000$ and $k = 50$. Notice that the NLS times drop dramatically to a constant time per iteration as it scales as $1/p$. $\sqrt{p}$ scaling in communication times is seen.



(a) Total breakdown.



(b) Computation breakdown.



(c) Communication breakdown.

Figure 2.17: Weak-scaling timing breakdowns for the sparse case with $m/p_r = 76{,}000$ and $n/p_c = 47{,}500$ and $k = 50$. NLS times drop quickly like in the dense case. $\sqrt{p}$ scaling in communication times is seen.

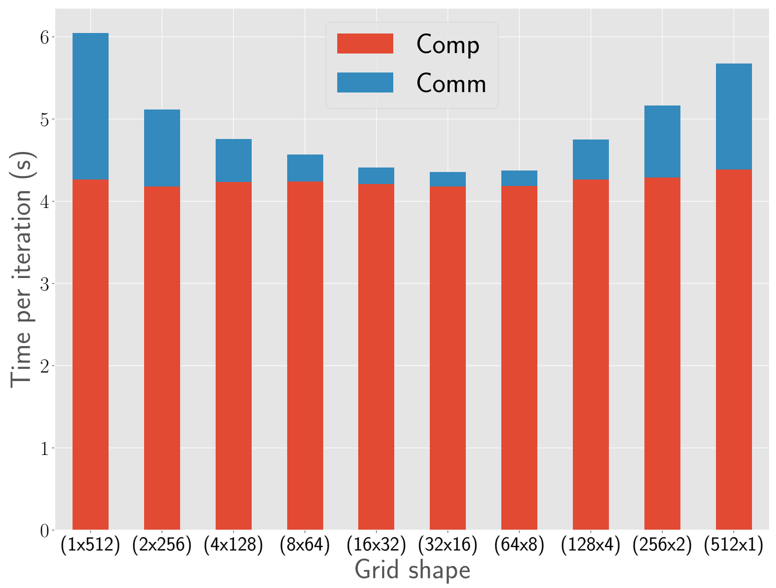
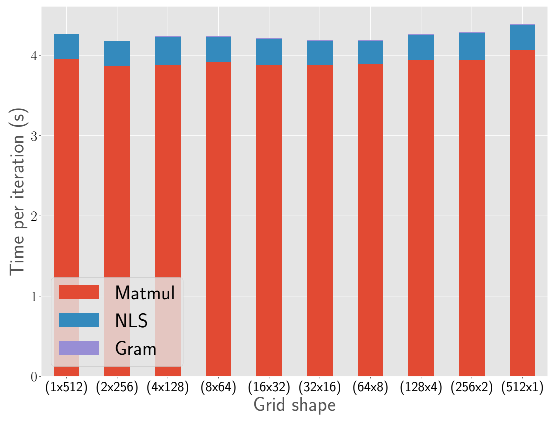


Figure 2.18: Choice of grids for the dense case. Optimal grid selection can lead to up to $1.4\times$ speedup.

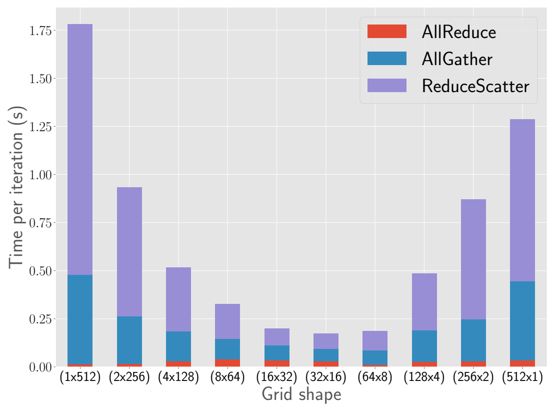
`Matmul` times shouldn't change when we scale as seen in Figs. 2.15b and 2.16b. However, since the NLS time scales as $\frac{m+n}{p}$ it should scale much faster than the matrix multiplication and reach a constant time per iteration. This rapid decrease can clearly be seen in the dense case (Fig. 2.16b). Scaling the matrix dimensions in this manner also increases communication times as we scale up, Figs. 2.16c and 2.17c.

Grid Selection

The final experiment looks at the choice of different grid shapes, $p_r \times p_c$, on Algorithm 3. The theoretically optimal choice, to minimise communication, is to select it based on the aspect ratio of the input matrix (see Section 2.3.3). In our case, we have $\frac{m}{n} = 2$, so we expect the grids with this aspect ratio to have the best performance. Grids with this shape also have the added advantage of having square local portions of the input $\mathbf{X}_{ij}$, which could help in the local multiplication as well. From Fig. 2.19, it is clear that naïvely partitioning $\mathbf{X}$ either block row-wise or column-wise is inferior to the 2D splitting scheme ($1.94\times$ improvement). Looking at the breakdown charts of Fig. 2.18 we see that the computation



(a) Grid choice on computation.



(b) Grid choice on communication.

Figure 2.19: Finer details of grid selection. Majority of the improvement arises from gains in communication costs with computation costs showing little variations across the different grids.

aspects are roughly similar across the grids (Fig. 2.19a), with only a 5% difference between the best (2×256) and worst performing grids (512×1). However, the communication breakdowns tell a completely different story (Fig. 2.19b). The best grid (32×16) is $10\times$ better than the worst grid (1×512). Therefore, all the gains in this experiment come from the communication improvements, highlighting the importance of the choice of grids.

2.4 Summary

This chapter provides an overview of the NMF problem, including its history, geometrical interpretation, and hardness in Section 2.1. Due to NMF's nonconvex nature, the best approximation one can hope to obtain is a local minimum. In Section 2.2, we introduce the popular BCD framework for NMF. In particular, we go over several update algorithms and their common computations. The primary bottleneck for a certain class of NMF algorithms (see Section 2.2.2) is the normal equations formation or gradient computation. These involve computing the matrix products $\mathbf{W}^\mathsf{T}\mathbf{W}$, $\mathbf{H}\mathbf{H}^\mathsf{T}$, $\mathbf{W}^\mathsf{T}\mathbf{X}$, and $\mathbf{H}\mathbf{X}^\mathsf{T}$.

The key to parallelising NMF is a carefully designed matrix multiplication kernel. We give a brief overview of the new communication-optimal class of parallel matrix multiplicaiton algorithms in Section 2.3.2. We introduce PLANC, a software library for NMF, which employs such a communication-avoiding matrix multiplication algorithm (Section 2.3.3). It is also able to exploit the independent nature of the NLS subproblems to solve them in parallel. Finally, we demonstrate these bottlenecks and solutions via a small suite of experiments in Section 2.3.4. In particular, multiplication with $\mathbf{X}$ is shown to be the major bottleneck which is improved by using our communication-avoiding algorithm. Designing this kernel opens up the possibilities of improving other similar CLRA problems which we tackle in the subsequent chapters.

CHAPTER 3
SYMMETRIC CONSTRAINTS

One natural variation on the NMF theme is the switch from the usual *feature-data* representation of a data item to a *data-data* relationship. These inputs are represented either as symmetric similarity matrices or adjacency matrices of undirected graphs. This approach forces symmetric constraints on the factor matrices which we explore in this chapter.

We develop the first distributed-memory parallel implementation of SymNMF, a key data analytics kernel for clustering and dimensionality reduction. Our implementation includes two different algorithms for SymNMF, which give comparable results in terms of time and accuracy. The first algorithm is a parallelisation of an existing sequential approach that uses solvers for nonsymmetric NMF. The second algorithm is a novel approach based on the Gauss-Newton method. It exploits second-order information without incurring large computational and memory costs.

We evaluate the scalability of our algorithms on the Summit system at Oak Ridge National Laboratory, scaling up to 128 nodes (4,096 cores) with 70% efficiency. Additionally, we demonstrate our software on an image segmentation task.

3.1 Introduction

The *symmetric nonnegative matrix factorisation* (SymNMF) problem arises in unsupervised learning and data mining applications [42, 69–72]. In this problem, the input data is naturally represented by a symmetric, nonnegative data matrix $\mathbf{S} \in \mathbb{R}_+^{n\times n}$ where $\mathbf{S} = \mathbf{S}^\mathsf{T}$ and $\mathbf{S} \geq 0$. For instance, in graph clustering $\mathbf{S}$ might be an unweighted adjacency matrix; in image segmentation $\mathbf{S}$ represents pixel-to-pixel positive similarity scores [73]; and in topic modeling $\mathbf{S}$ stores normalised co-occurrence counts [74]. Data analysis tasks such as dimension reduction, clustering, embedding, or imputation of missing values, may be formulated as a CLRA problem on $\mathbf{S} \approx \mathbf{H}^\mathsf{T}\mathbf{H}$. Preserving nonnegativity in $\mathbf{H}$ helps the end-user analyst interpret the results more readily.

Sequential algorithms to compute SymNMF are summarised by Kuang et al. [42]. Following methods for the standard nonsymmetric NMF case [27, 75], it formulates SymNMF as the optimisation problem,

$$\min_{\mathbf{H}\geq 0} \left\|\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2 . \tag{3.1}$$

Like NMF (see Section 2.1.3), Eq. (3.1) is nonlinear and nonconvex, which makes it hard to solve to optimality.

Solution algorithms may be broadly classified into two groups: BCD and direct optimisation. BCD algorithms, ANLS [42] and Cyclic Coordinate Descent (CCD) [69], partition the solution variables into blocks and alternate among solving subproblems with all but one block of a variables fixed, each of which can be solved exactly. ANLS duplicates the solution matrix $\mathbf{H}$, solves the problem as a nonsymmetric problem, and uses regularisation to converge to a unified symmetric solution (discussion in Section 3.3). Direct optimisation techniques such as PGD and Newton-like algorithms [42] iteratively update all the variables at once. First-order direct optimisation uses only gradient information (PGD) and can suffer slow convergence, while second-order methods incorporate Hessian information to improve convergence at the price of higher per-iteration costs. Previously developed second-order methods suffer from a particularly high computational complexity of $O(n^3k^3)$ per iteration, making them infeasible for even moderate data sizes. Regarding solution quality, neither method is clearly superior to the other and both are expensive to compute sequentially even at moderate input sizes [42]. As such, we are motivated to consider both classes of methods and to improve both scalability via parallelism and work-efficiency via algorithmic techniques.

A critical advantage of parallelising the ANLS variant of SymNMF is that one can reuse existing algorithms and software developed for the nonsymmetric NMF case in the previous chapter (Chapter 2). We present our algorithm in Section 3.3. It enjoys the same advantages as the nonsymmetric method, and is the first distributed-memory parallel algorithm for SymNMF.

Our second algorithm is a novel Newton-like algorithm for SymNMF based on the Gauss-Newton method (Section 3.4). It efficiently uses second-order information *implicitly* without incurring prohibitive memory or computational costs. The proposed Gauss-Newton method for SymNMF finds low-rank approximations that are competitive with existing algorithms in quality (see Section 3.5.4). Surprisingly, our parallel Gauss-Newton method using Conjugate Gradients (GNCG) can even be twice as fast as the ANLS variant in practice.

We evaluate these methods on the Summit supercomputer[1] using a variety of datasets (Section 3.5). We are able to scale GNCG up to 128 nodes (4,096 cores) with 70% efficiency. In general, GNCG runs $2\times$ faster than ANLS for most problems where gradient computation is the major bottleneck (see Section 3.5.5). In cases with extremely large k, ANLS becomes competitive again (Section 3.5.6).

Overall, the end results constitute the first distributed-memory parallel methods for the SymNMF problem. The significance is to enable practical use of SymNMF on real-world problems. As an example, we present a case study in which we apply SymNMF to the

[1]The system hosted at Oak Ridge National Laboratory.

segmentation of large satellite images [76]. The pixel embeddings produced by SymNMF are used to detect boundaries and partition images into regions [42, 73].

3.2 Preliminaries

3.2.1 Notation

The matrix $\mathbf{S}$ will refer to a square, nonnegative $n \times n$ data matrix, and we reserve $\mathbf{H}$ to represent the factor matrix of the symmetric approximation. We use $[\mathbf{X}]_+$ to denote projection to the nonnegative orthant. The $\mathrm{vec}(\mathbf{X})$ operator vectorises a matrix by stacking its columns on top of each other. The Kronecker product between two matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{p\times q}$ is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{mp\times nq} .$$

$\mathbf{P}_{p,q}$ is used to denote the commutation matrix which converts the column-major vectorisation of $\mathbf{A} \in \mathbb{R}^{p\times q}$ to its row-major one, that is

$$\mathbf{P}_{p,q}\mathrm{vec}(\mathbf{A}) = \mathrm{vec}\big(\mathbf{A}^\mathsf{T}\big) .$$

As with all permutation matrices $\mathbf{P}_{n,k}$ is orthogonal which means that

$$\mathbf{P}_{p,q}\mathbf{P}_{p,q}^\mathsf{T} = \mathbf{I}_n = \mathbf{P}_{p,q}^\mathsf{T}\mathbf{P}_{p,q} .$$

This leads to the following useful property $\mathbf{P}_{p,q}^\mathsf{T} = \mathbf{P}_{q,p}$, since

$$\mathbf{P}_{p,q}^\mathsf{T}\mathbf{P}_{p,q}\mathrm{vec}(\mathbf{A}) = \mathrm{vec}(\mathbf{A}) = \mathbf{P}_{q,p}\mathrm{vec}\big(\mathbf{A}^\mathsf{T}\big) .$$

The main use of the commutation matrix is to commute the Kronecker product. The following result holds for every $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{r \times q}$,

$$\mathbf{P}_{r,m} \left(\mathbf{A} \otimes \mathbf{B} \right) \mathbf{P}_{n,q} = \mathbf{B} \otimes \mathbf{A} \ .$$

3.2.2 Parallel Nonsymmetric NMF

We build our implementations on top of PLANC [36] which is an extension of MPI-FAUN [35] designed for nonsymmetric NMF. PLANC is written in C++, uses MPI for the interprocessor communication, and uses Armadillo for local matrix operations (interfacing to a BLAS implementation for dense matrix operations). It is designed to solve the problem

$$\min_{\mathbf{W} \geq 0, \mathbf{H} \geq 0} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 \tag{3.2}$$

for dense or sparse, rectangular, nonnegative matrices $\mathbf{X}$ using algorithms that alternate between updating $\mathbf{W}$ for fixed $\mathbf{H}$ and updating $\mathbf{H}$ for fixed $\mathbf{W}$. Section 2.3.3 gives an overview of the different parallel kernels present in PLANC.

3.2.3 Gauss-Newton Method using Conjugate Gradient

The Gauss-Newton (GN) method is a technique for minimising a sum of squares of residual functions. That is, it can be used to solve optimisation problems involving multivariate functions of the form $f(\mathbf{x}) = \frac{1}{2}\sum_l r_l(\mathbf{x})^2$. Here the functions $r_l(\mathbf{x})$ are called the residual functions and are generally nonlinear. GN proceeds to minimise $f(\mathbf{x})$ by starting with an initial guess $\mathbf{x}^{(0)}$ and following the iteration:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \operatorname*{argmin}_{\mathbf{p}} \left\| \mathbf{J}^{(t)}\mathbf{p} - \mathbf{r}^{(t)} \right\|_2^2 . \tag{3.3}$$

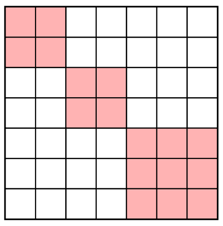
Here $\mathbf{J}^{(t)}$ is the Jacobian matrix defined as $\mathbf{J}_{lq}^{(t)} = \frac{\partial r_l}{\partial \mathbf{x}_q}(\mathbf{x}^{(t)})$ and $\mathbf{r}^{(t)}$ is a vector of the residual function values $r_l(\mathbf{x}^{(t)})$. This iteration is performed until some stopping criteria is met. We solve the linear least-squares problem via the normal equations. The GN method is a second-order optimisation method where the matrix $\mathbf{J}^\mathsf{T}\mathbf{J}$ acts as an approximate Hessian matrix for f [43].

Thus, the main task at each iteration is to find a solution to the linear system:

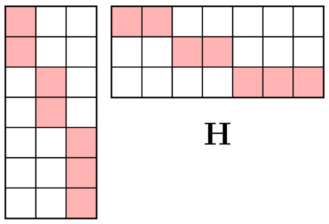
$$\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)\mathbf{p} = \mathbf{J}^{(t)\mathsf{T}}\mathbf{r}^{(t)}. \tag{3.4}$$

For this solve, one can use an iterative method for symmetric positive semi-definite matrices such as the Conjugate Gradient method (CG) [77]. The right-hand-side vector $\mathbf{J}^{(t)\mathsf{T}}\mathbf{r}^{(t)} = \mathbf{g}^{(t)}$ is the gradient evaluated at $\mathbf{x}^{(t)}$. The CG algorithm maintains for each iteration an approximate solution vector, a search or update direction vector, and a residual vector, and it requires the multiplication of the coefficient matrix with an approximate solution vector as well as several vector operations. While the size of the Gramian of the Jacobian is large, CG iterations can be performed efficiently if the matrix-vector multiplication can exploit structure in the Jacobian and avoid the explicit formation of the Gramian. If the system Eq. (3.4) is only solved approximately, this method is known as an *inexact* Gauss-Newton algorithm [78].

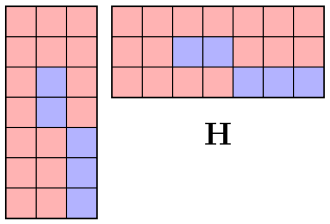
For constrained optimisation problems, the Gauss-Newton step is not guaranteed to maintain the constraint satisfaction. The updated solution can be projected back onto the feasible set. We apply the projected version of this method and show how to implement it efficiently for SymNMF in Section 3.4.1. These methods are known as *two-metric projection methods* because they embody two different scaling matrices; one is $\left(\mathbf{J}^\mathsf{T}\mathbf{J}\right)^{-1}$ used to scale the gradient and the other is the identity matrix used in the Euclidean projection norm [43]. In general, there is no certainty that $\mathbf{p}$ forms a descent direction and thereby guarantee convergence. However as seen in Section 3.5.4, empirically this method behaves reasonably well and reduces the objective similarly to other algorithms.



(a) Input with block diagonal structure.



(b) SymNMF solution.



(c) Spectral Clustering solution.

Figure 3.1: An illustrative example for SymNMF with an input consisting of three diagonal blocks. The red entries indicate larger positive values than the white ones. The blue entries are negative. The SymNMF solution approximates the input additively. Spectral Clustering, on the other hand, performs this task by subtracting the off-diagonal blocks.

3.2.4 Clustering via SymNMF

We provide an intuitive explanation of why SymNMF is expected to extract cluster structures from symmetric nonnegative matrices representing graphs and similarity scores. Just like in NMF (see Section 2.1.4), the nonnegativity constraint $\mathbf{H} \geq 0$ plays an important role in extracting cluster structure. In the simplest case, when $\mathbf{S}$ is a binary matrix with block diagonal structure corresponding to an undirected graph, the key task is to find the nodes which are highly connected. This is the clique finding problem for the block diagonal $\mathbf{S}$. In general, we aim to find subsets or *communities* of nodes that are densely connected with various definitions of densities like cut, ratio cut, normalised cut, amongst others [25].

SymNMF decomposes $\mathbf{S}$ as

$$\mathbf{S} \approx \mathbf{H}^{\mathsf{T}}\mathbf{H} = \sum_{r=1}^{k} \mathbf{H}(r,:)^{\mathsf{T}}\,\mathbf{H}(r,:) = \sum_{r=1}^{k} \hat{\mathbf{h}}_r \hat{\mathbf{h}}_r^{\mathsf{T}}, \tag{3.5}$$

where $\hat{\mathbf{h}}_r$ is the r^{th} row from $\mathbf{H}$ reshaped as a column vector. We can see that $\mathbf{S}$ is approximated as the sum of k symmetric and nonnegative rank-one matrices $\hat{\mathbf{h}}_r \hat{\mathbf{h}}_r^{\mathsf{T}}$. A simple example of an input with three diagonal blocks is shown in Fig. 3.1. The additive nature of SymNMF allows its $\mathbf{H}$ to behave as a cluster membership indicator (see Fig. 3.1b). Each node's group can be found by looking for the maximum entry along its reduced representation (column $\mathbf{h}_i$ for node i). Approximation methods which do not enforce nonnegativity, like Spectral Clustering shown in Fig. 3.1c, do not share this property in general. In this case, each diagonal block need not be approximately separately by an outer product $\hat{\mathbf{h}}_r\hat{\mathbf{h}}_r^\mathsf{T}$ but rather can be combined with multiple other outer products to do so.

SymNMF is closely related to a class of graph clustering methods known as Spectral Clustering [24, 25, 79]. These algorithms rely on the eigendecomposition and are often characterised by the following optimisation problem,

$$\max_{\mathbf{H}\geq 0, \mathbf{H}\mathbf{H}^\mathsf{T}=\mathbf{I}_k} \mathrm{Tr}\left(\mathbf{H}\mathbf{S}\mathbf{H}^\mathsf{T}\right) \ , \tag{3.6}$$

where $\mathbf{H} \in \mathbb{R}^{k\times n}$ and Tr$(\cdot)$ is the trace of a matrix. The rather restrictive constraints ensure that each column of $\mathbf{H}$ contains only a single positive entry and make the problem NP-hard [25]. The objective function in Eq. (3.6) is known to be same as in Eq. (3.1) and we show it here for completeness [80]. Under the constraints of $\mathbf{H} \geq 0, \mathbf{H}\mathbf{H}^\mathsf{T} = \mathbf{I}_k$ we have,

$$\begin{aligned}
&\max \mathrm{Tr}\left(\mathbf{H}\mathbf{S}\mathbf{H}^\mathsf{T}\right)\\
\iff &\min \mathrm{Tr}\left(\mathbf{S}^\mathsf{T}\mathbf{S}\right) - 2\mathrm{Tr}\left(\mathbf{H}\mathbf{S}\mathbf{H}^\mathsf{T}\right) + \mathrm{Tr}\left(\mathbf{I}_k\right)\\
\iff &\min \mathrm{Tr}\left(\left(\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right)^\mathsf{T}\left(\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right)\right)\\
\iff &\min \left\|\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2 \ .
\end{aligned}$$

Relaxing the constraints for Eq. (3.6) is a practical way to make this problem tractable. Spectral clustering methods relax the nonnegativity constraints whereas SymNMF relaxes the orthogonality constraint. Zass and Shashua argue that the nonnegativity constraints are more important than orthogonality for two reasons [81]. The first is that $\mathbf{H} \geq 0$ confers physical meaning to clusters as illustrated in Fig. 3.1. Spectral Clustering also produces physically meaningful $\hat{\mathbf{h}}$ for the case $k = 2$, we can detect cluster membership by checking if column vectors of $\mathbf{H}$ contain a negative entry, but becomes heuristic in nature for $k > 2$. The second is that $\mathbf{H}\mathbf{H}^{\mathsf{T}} = \mathbf{I}_k$ only serves the role to enforce hard clustering of the data items in $\mathbf{S}$, that is each item belongs to only one cluster. Removing them is equivalent to taking a probabilistic or "soft" clustering approach.

A final note on utilising SymNMF for clustering is on the effect of the eigenvalues of $\mathbf{S}$. SymNMF tries to approximate the symmetric and nonnegative $\mathbf{S}$ by $\mathbf{H}^{\mathsf{T}}\mathbf{H}$ which is a rank-k symmetric positive semi-definite matrix. If $\mathbf{S}$ contains fewer than k nonnegative eigenvalues, then SymNMF might not give a good approximation. This condition can be expensive to check but Kuang et al. describe why it is reasonable to assume this is satisfied when we expect $\mathbf{S}$ to contain k clusters [42].

3.2.5 Related Work

SymNMF has been extensively studied and a number of effective sequential algorithms have been proposed for solving Eq. (3.1). Vandaele et al. propose an algorithm using a k block BCD approach in which elements of the matrix $\mathbf{H}$ are iteratively updated row-by-row in a cyclic fashion [69]. This cyclic coordinate descent (CCD) algorithm is not readily parallelisable because of its dependencies among elements. Kuang et al. present three algorithms for computing SymNMF: 1) a method based on PGD, 2) a Newton-type method which utilises second-order information, and 3) a regularised two-block BCD method based on ANLS [42, 80]. The authors focus on the Newton-type and ANLS algorithms as they both tend to outperform the PGD algorithm [42]. They conclude that the ANLS algorithm performs most effectively in terms of quality of solution and run time. We parallelise the ANLS algorithm in this work, as described in Section 3.3. Though we do not develop distributed-memory parallel algorithms for the PGD algorithm [42] or the CCD algorithm [69], we provide convergence experiments from sequential implementations in Section 3.5.4 to compare these algorithms.

The Gauss-Newton method has been used before in the context of low-rank approximation. For example, Gauss-Newton methods are effective for the computation of the CANDECOMP/PARAFAC (CP) tensor decomposition. Vervliet and de Lathauwer detail how GN with CG can be used to efficiently compute the CP decomposition [82]. In the context of high-performance computing, Singh et al. compare the scalability of the Alternating Least Squares (ALS) and GN algorithms for computing a CP decomposition [83]. The authors demonstrate efficient weak-scaling results for both algorithms but less compelling strong-scaling results, particularly for the GN algorithm. The authors attribute this to the fact that the ALS algorithm contains more easily parallelisable computations and is more dominated by computation.

3.3 Nonsymmetric Alternating-Updating NMF with Symmetric Regularisation

3.3.1 SymNMF via Alternating Nonnegative Least Squares

One approach we consider for solving Eq. (3.1) is to compute a nonsymmetric NMF with a regularisation term that drives the two nonsymmetric factors towards each other to encourage convergence to a symmetric solution. As proposed by Kuang et al. [42], we can use the following surrogate optimisation problem:

$$\min_{\mathbf{W}\geq 0, \mathbf{H}\geq 0} \|\mathbf{S} - \mathbf{W}\mathbf{H}\|_F^2 + \gamma \left\|\mathbf{W} - \mathbf{H}^\mathsf{T}\right\|_F^2. \tag{3.7}$$

The symmetry constraint is dropped and in its place we add the regulariser $\gamma \left\|\mathbf{W} - \mathbf{H}^\mathsf{T}\right\|_F^2$, where $\gamma \geq 0$. Note that if $\gamma = 0$ Eq. (3.2) is recovered. This encourages the algorithm to find a solution such that $\mathbf{W} \approx \mathbf{H}^\mathsf{T}$. Zhu et al. show that algorithms dropping the symmetry constraint in this manner with a sufficiently large γ return a solution to Eq. (3.1) [84].

In this approach, we can use existing solvers for nonsymmetric NMF, including those available in PLANC. For alternating-updating algorithms, the subproblem for updating $\mathbf{H}$

is a NLS of the form

$$\min_{\mathbf{H}\geq 0} \left\| \begin{bmatrix} \mathbf{W} \\ \sqrt{\gamma}\, \mathbf{I}_k \end{bmatrix} \mathbf{H} - \begin{bmatrix} \mathbf{S} \\ \sqrt{\gamma}\, \mathbf{W}^\mathsf{T} \end{bmatrix} \right\|_F^2 ,$$

and the subproblem for updating $\mathbf{W}$ is similar. Following Kuang et al., we will refer to this approach as ANLS [42].

The bulk of the computation for any alternating-updating algorithm is the computation of the matrices involved in the gradients: $\mathbf{W}^\mathsf{T}\mathbf{W} + \gamma\mathbf{I}_k$ and $\mathbf{W}^\mathsf{T}\mathbf{S} + \gamma\mathbf{W}^\mathsf{T}$ (for updating $\mathbf{H}$) and $\mathbf{H}\mathbf{H}^\mathsf{T} + \gamma\mathbf{I}_k$ and $\mathbf{H}\mathbf{S} + \gamma\mathbf{H}$ (for updating $\mathbf{W}$), assuming $\mathbf{S}$ is symmetric, which are the matrices appearing in the gradients of the subproblems. More detailed analysis is given in Section 3.3.4.

3.3.2 Parallel Nonsymmetric ANLS Algorithm

We parallelise the ANLS method using the PLANC software framework [35, 36]. For a complete description of the parallel matrix multiplication algorithms used for the nonsymmetric case please refer to Section 2.3.3. Fig. 3.2 illustrates the data distribution of the data and factor matrices for a square matrix and a 3×3 processor grid. The matrices are oriented to emphasise the conformal distributions of $\mathbf{W}$ to processor rows and $\mathbf{H}$ to processor columns, which is designed to support a particular parallel algorithm for matrix multiplication.

3.3.3 Parallel ANLS

We now describe how to adapt the nonsymmetric parallel ANLS algorithm to the symmetric case. We use the same 2D distribution of the data matrix, always with a square $\sqrt{p} \times \sqrt{p}$ processor grid, storing both upper and lower triangles of the matrix explicitly. We also use the same 1D distribution of the factor matrices, so the parallel algorithm for computing Gram matrices does not change.

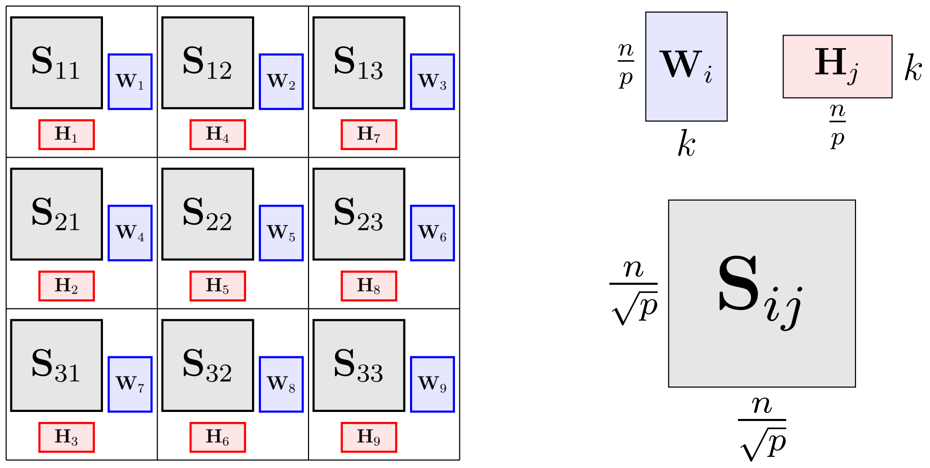


Figure 3.2: Nonsymmetric data distribution for a 3×3 processor grid.

The difference arises in the computation of $\mathbf{W}^\mathsf{T}\mathbf{S} + \gamma\mathbf{W}^\mathsf{T}$ and $\mathbf{HS} + \gamma\mathbf{H}$. In particular, we note that the distributions of $\mathbf{W}$ and $\mathbf{H}$ are not identical. As shown in Fig. 3.2, the second block of $\mathbf{W}$ is owned by processor $(1, 2)$, while the second block of $\mathbf{H}$ is owned by processor $(2, 1)$. The nonsymmetric matrix multiplication algorithm is designed so that $\mathbf{W}^\mathsf{T}\mathbf{S}$ has the same distribution of $\mathbf{H}$. In the symmetric case, to incorporate the regularisation, we must add $\mathbf{W}^\mathsf{T}\mathbf{S}$ to $\gamma\mathbf{W}^\mathsf{T}$, but these matrices are not identically distributed and so their addition requires communication. Because the result will be used to update $\mathbf{H}$, and the distribution of $\mathbf{W}^\mathsf{T}\mathbf{S}$ matches $\mathbf{H}$, we communicate the necessary block of $\mathbf{W}$ to complete the addition. This communication can be performed via pairwise exchanges between symmetric partners (processors (i, j) and (j, i) for $i \neq j$), as depicted in Fig. 3.3. A similar technique is used to communicate $\mathbf{H}$ for the matrix addition with $\mathbf{HS}$ when updating $\mathbf{W}$.

The rest of the nonsymmetric NMF algorithm can be applied directly: multiple algorithms can be used to solve the local nonnegative least squares problem. After convergence, either factor or their average may be used as the symmetric result; to average $\mathbf{W}$ and $\mathbf{H}$ requires using a temporary local copy of the matrix that was communicated for the final update step. We present the ANLS algorithm with symmetric regularisation in Algorithm 4.

Algorithm 4 $[\mathbf{W},\mathbf{H}] = \text{SymANLS}(\mathbf{A},k,\gamma)$

```
Require: $\mathbf{S} \in \mathbb{R}_+^{n\times n}$ is distributed across a $\sqrt{p} \times \sqrt{p}$ grid of processors, $k > 0$ is rank of approximation, $p$ divides $n$
Require: Local matrices: $\mathbf{S}_{ij}$ is $n/\sqrt{p} \times n/\sqrt{p}$, $\hat{\mathbf{W}}_i \in n/\sqrt{p} \times k$ and $\hat{\mathbf{H}}_j \in k \times n/\sqrt{p}$, $\mathbf{W}_r$ is $n/p \times k$ and $\mathbf{H}_c$ is $k \times n/p$
Ensure: $\mathbf{W},\mathbf{H} \approx \text{argmin}_{\tilde{\mathbf{W}},\tilde{\mathbf{H}}\geq 0} \left\|\mathbf{S} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\right\|_F^2 + \gamma \left\|\tilde{\mathbf{W}} - \tilde{\mathbf{H}}^\mathsf{T}\right\|_F^2$
Ensure: $\mathbf{W},\mathbf{H}$ are $n \times k$ row-wise and $k \times n$ column-wise distributed across processors
 1: $\mathbf{p} = (i,j) = p_{ij}$                                  ▷ Processor rank
 2: $r = (i-1)\,p_c + j$                                           ▷ Row-major rank
 3: $c = (j-1)\,p_r + i$                                           ▷ Column-major rank
 4: Initialise $\mathbf{H}_c^{(0)}$
 5: while $t = 1,2,\ldots$ do                       ▷ Till some stopping condition is met.
        % Compute W given H
 6:     $\mathbf{U}_c = \mathbf{H}_c\mathbf{H}_c^\mathsf{T}$
 7:     $\mathbf{H}\mathbf{H}^\mathsf{T}$ = All-Reduce($\mathbf{U}_c$)
 8:     $\hat{\mathbf{H}}_j^{(t)}$ = All-Gather($\mathbf{H}_c^{(t)}$,$(:,j)$)
 9:     $\mathbf{V}_{ij} = \hat{\mathbf{H}}_j^{(t)}\mathbf{S}_{ij}$
10:     $(\mathbf{H}\mathbf{S})_r$ = Reduce-Scatter($\mathbf{V}_{ij}, (i,:)$)
11:     $p_{ij}$ sends $\mathbf{H}_c$ to $p_{ji}$ and receives $\mathbf{H}_r$ from $p_{ji}$
12:     $\mathbf{W}_r^{(t)}$ = update($\mathbf{H}\mathbf{H}^\mathsf{T} + \gamma\mathbf{I}_k, (\mathbf{H}\mathbf{S})_r + \gamma\mathbf{H}_r$)
        % Compute H given W
13:     $\mathbf{X}_r = \mathbf{W}_r^\mathsf{T}\mathbf{W}_r$
14:     $\mathbf{W}^\mathsf{T}\mathbf{W}$ = All-Reduce($\mathbf{X}_r$)
15:     $\hat{\mathbf{W}}_i^{(t)}$ = All-Gather($\mathbf{W}_r^{(t)}$,$(i,:)$)
16:     $\mathbf{Y}_{ij} = \left(\hat{\mathbf{W}}_i^{(t)}\right)^\mathsf{T}\mathbf{S}_{ij}$
17:     $\left(\mathbf{W}^\mathsf{T}\mathbf{S}\right)_c$ = Reduce-Scatter($\mathbf{Y}_{ij}, (:,j)$)
18:     $p_{ij}$ sends $\mathbf{W}_r$ to $p_{ji}$ and receives $\mathbf{W}_c$ from $p_{ji}$
19:     $\mathbf{H}_c^{(t)}$ = update($\mathbf{W}^\mathsf{T}\mathbf{W} + \gamma\mathbf{I}_k, \left(\mathbf{W}^\mathsf{T}\mathbf{S}\right)_c + \gamma\mathbf{W}_c^\mathsf{T}$)
20: end while
```

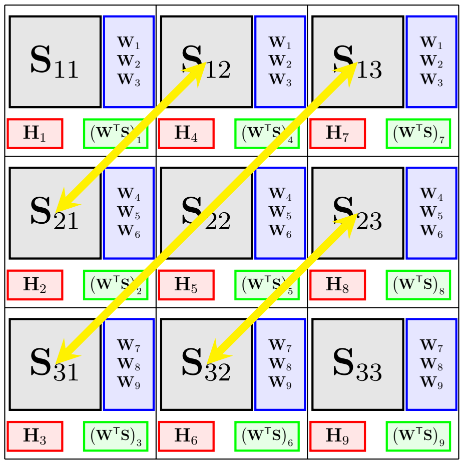


Figure 3.3: Data distribution of symmetric ANLS and communication pattern of $\mathbf{W}$ to compute $\mathbf{W}^\mathsf{T}\mathbf{S} + \gamma\mathbf{W}^\mathsf{T}$ for a 3×3 processor grid. $\mathbf{W}$ needs to be All-Gathered across processor rows. $\mathbf{W}^\mathsf{T}\mathbf{S}$ is Reduce-Scattered across processor columns. The yellow arrows indicate the communication needed to add $\mathbf{W}^\mathsf{T}$ to $\mathbf{W}^\mathsf{T}\mathbf{S}$. The opposite communication patter (swap rows and columns) occurs to compute $\mathbf{HS}$.

3.3.4 Analysis

The computation and communication costs of ANLS for SymNMF are nearly identical to the nonsymmetric case (Section 2.3.3). The dominant computation costs are due to multiplications between the data matrix and each factor matrix, computing Gram matrices of the factor matrices, and evaluating the local update function. Computing the products involving the matrix $\mathbf{S}$ costs $4n^2k/p$ when $\mathbf{S}$ is dense and $4k\text{nnz}(\mathbf{S})/p$ when $\mathbf{S}$ is sparse, the Gram matrix computations cost $O(nk^2/p)$ arithmetic operations, and the matrix additions are $O(nk/p)$. We use the BPP method for the local update, which can cost $O(nk^3s(k)/p)$ operations but in practice takes much fewer.

The size of the data involved in the All-Gather and Reduce-Scatters is $O(nk/\sqrt{p})$, so the costs are $O(nk/\sqrt{p})$ words and $O(\log p)$ messages. The extra cost of the pairwise exchange does not affect the leading order communication costs of the overall algorithm, because the size of the messages is $O(nk/p)$. The communication cost of the Gram all-reduces is $O(\log p)$ messages of size $O(k^2)$. Additionally, the algorithm never communicates the data matrix so these communication costs are the same for both the dense and sparse cases. We compare the costs of GNCG with ANLS (for the dense case) in Table 3.1.

3.4 Gauss-Newton based Distributed Symmetric Nonnegative Matrix Factorisation

3.4.1 Gauss-Newton for SymNMF

Here we derive the Gauss-Newton method for solving the SymNMF objective function in Eq. (3.1) and show how to employ CG efficiently for each GN step. Since we are using the Frobenius norm, our SymNMF objective function is a sum of squares. The GN method is described for general objective functions in Section 3.2.3. In this case, the residual functions are of the form

$$r_{ij}(\mathbf{H}) = s_{ij} - \sum_{\ell=1}^{k} h_{\ell i} h_{\ell j}.$$

Note that we use two indices for the residual functions because they correspond to matrix entries, though they must be vectorised, as $\mathbf{r} = \mathrm{vec}(\mathbf{R})$, before corresponding to rows of the Jacobian. The Jacobian is a $n^2 \times nk$ matrix of partial derivatives of the vectorised residual with respect to the vectorised factor matrix $\mathbf{h} = \mathrm{vec}(\mathbf{H})$ with the following entry wise form.

$$j_{pq}(\mathbf{H}) = \frac{\partial r_p}{\partial h_q} \, .$$

Compactly the residual and Jacobian are defined below.

$$\mathbf{r} = \mathrm{vec}\big(\mathbf{S} - \mathbf{H}^{\mathsf{T}}\mathbf{H}\big) \tag{3.8}$$

$$\mathbf{J} = \frac{\partial \mathbf{r}}{\partial \mathrm{vec}(\mathbf{H})} = \frac{\partial \mathrm{vec}\big(\mathbf{S} - \mathbf{H}^{\mathsf{T}}\mathbf{H}\big)}{\partial \mathrm{vec}(\mathbf{H})} \tag{3.9}$$

We make use of the following identities and the properties of commutation matrices (see Section 3.2) to derive a structured representation of the Jacobian [82].

$$\frac{\partial \text{vec}(\mathbf{WH})}{\partial \text{vec}(\mathbf{W})} = \mathbf{H}^\mathsf{T} \otimes \mathbf{I}_m \quad \frac{\partial \text{vec}(\mathbf{WH})}{\partial \text{vec}(\mathbf{H})} = \mathbf{I}_n \otimes \mathbf{W} \quad \frac{\partial \text{vec}(\mathbf{WH})}{\partial \text{vec}\big(\mathbf{W}^\mathsf{T}\big)} = \big(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_m\big)\, \mathbf{P}_{k,m} \tag{3.10}$$

We can now obtain $\mathbf{J}$ as follow,

$$\begin{aligned}
\mathbf{J} &= \frac{\partial \text{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big)}{\partial \text{vec}(\mathbf{H})} \\
&= -\frac{\partial \text{vec}\big(\mathbf{H}^\mathsf{T}\mathbf{H}\big)}{\partial \text{vec}(\mathbf{H})} \\
&= -\left(\frac{\partial \text{vec}(\mathbf{AH})}{\partial \text{vec}(\mathbf{H})} + \frac{\partial \text{vec}\big(\mathbf{H}^\mathsf{T}\mathbf{B}\big)}{\partial \text{vec}(\mathbf{H})}\right) \qquad \text{(product rule for derivatives.)} \\
\mathbf{J} &= -\big(\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}\big) - \big(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\big)\, \mathbf{P}_{k,n}
\end{aligned}$$

For the product rule step, we fix one of $\mathbf{H}$ factors in the matrix product $\mathbf{H}^\mathsf{T}\mathbf{H}$ as $\mathbf{AH}$ and $\mathbf{H}^\mathsf{T}\mathbf{B}$ for the first and second term respectively.

Gradient computation

In order to take a GN step, we must solve a linear system with coefficient matrix $\mathbf{J}^\mathsf{T}\mathbf{J}$ and right-hand-side $\mathbf{J}^\mathsf{T}\mathbf{r}$. The vectorised residual of our problem is $\mathbf{r} = \text{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big)$. The right-hand-side vector $\mathbf{g} = 2\mathbf{J}^\mathsf{T}\mathbf{r}$ (gradient, see Section 3.2.3) can be efficiently formed via

$$\begin{aligned}
\mathbf{g} &= -2((\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}) + (\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n)\mathbf{P}_{k,n})^\mathsf{T}\text{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big) \\
&= -2(\mathbf{I}_n \otimes \mathbf{H})\text{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big) - \mathbf{P}_{n,k}\,(\mathbf{H} \otimes \mathbf{I}_n)\,\text{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big) \\
&= -2\text{vec}\big(\mathbf{HS} - \mathbf{HH}^\mathsf{T}\mathbf{H}\big) - \mathbf{P}_{n,k}\text{vec}\Big(\big(\mathbf{HS} - \mathbf{HH}^\mathsf{T}\mathbf{H}\big)^\mathsf{T}\Big) \\
&= 4\text{vec}\big(\mathbf{HH}^\mathsf{T}\mathbf{H} - \mathbf{HS}\big)\,.
\end{aligned}$$

The second last equality is due to the identity $(\mathbf{B}^\mathsf{T} \otimes \mathbf{A})\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{AXB})$. An extra two appears in our $\mathbf{g}$ calculation, since our objective function is, $\left\|\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2 = \mathbf{r}^\mathsf{T}\mathbf{r}$, and not $\frac{1}{2}\mathbf{r}^\mathsf{T}\mathbf{r}$ as in Section 3.2.3. The approximate Hessian becomes $2\mathbf{J}^\mathsf{T}\mathbf{J}$ in this case as well. Since the constants cancel out when we solve for $\mathbf{J}^\mathsf{T}\mathbf{J}\mathbf{p} = \mathbf{J}^\mathsf{T}\mathbf{r}$, we can drop the extra factor of two and work with only $\mathbf{J}^\mathsf{T}\mathbf{r}$ and $\mathbf{J}^\mathsf{T}\mathbf{J}$ from here on.

Applying Gramian of Jacobian

In order to solve the linear system for the GN step using CG, we need to apply the coefficient matrix $\mathbf{J}^\mathsf{T}\mathbf{J}$ to a vector.

$$\begin{aligned}
\mathbf{J}^\mathsf{T}\mathbf{J} &= \left(\left(\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\right)\mathbf{P}_{k,n}\right)^\mathsf{T} \left(\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\right)\mathbf{P}_{k,n} && (3.11)\\
&= \left(\left(\mathbf{I}_n \otimes \mathbf{H}\right) + \mathbf{P}_{n,k}\left(\mathbf{H} \otimes \mathbf{I}_n\right)\right)\left(\left(\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\right)\mathbf{P}_{k,n}\right) && (3.12)\\
&= \left(\mathbf{I}_n \otimes \mathbf{H}\mathbf{H}^\mathsf{T}\right) + \mathbf{P}_{n,k}\left(\mathbf{H} \otimes \mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{H}\right)\mathbf{P}_{k,n} + \mathbf{P}_{n,k}\left(\mathbf{H}\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\right)\mathbf{P}_{k,n} && (3.13)\\
&= 2\left(\left(\mathbf{I}_n \otimes \mathbf{H}\mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{H}\right)\mathbf{P}_{k,n}\right) && (3.14)
\end{aligned}$$

Thus, to apply the Gramian to a vector $\mathbf{x}$, which we will consider to be a vectorisation of an $k \times n$ matrix $\mathbf{X}$, we use

$$\begin{aligned}
\mathbf{J}^\mathsf{T}\mathbf{J}\mathbf{x} &= 2\left(\left(\mathbf{I}_n \otimes \mathbf{H}\mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{H}\right)\mathbf{P}_{k,n}\right)\mathbf{x}\\
&= 2\left(\mathbf{I}_n \otimes \mathbf{H}\mathbf{H}^\mathsf{T}\right)\mathrm{vec}(\mathbf{X}) + 2\left(\mathbf{H}^\mathsf{T} \otimes \mathbf{H}\right)\mathbf{P}_{k,n}\mathrm{vec}(\mathbf{X})\\
&= 2\mathrm{vec}\left(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{X}\right) + 2\left(\mathbf{H}^\mathsf{T} \otimes \mathbf{H}\right)\mathrm{vec}\left(\mathbf{X}^\mathsf{T}\right)\\
&= 2\left(\mathrm{vec}\left(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{X}\right) + \mathrm{vec}\left(\mathbf{H}\mathbf{X}^\mathsf{T}\mathbf{H}\right)\right),
\end{aligned}$$

which can be computed using four dense matrix multiplications.

3.4.2 Parallel GNCG

We present the parallel algorithm for GNCG in Algorithm 5. Nearly all of the pseudocode is devoted to the "vector" operations of CG, which in our case corresponds to matrix additions and matrix inner products because we maintain all of the CG vectors as matrices with the same distribution as the factor matrix $\mathbf{H}$. The comments in the pseudocode show the standard vector notation of CG. We encapsulate the expensive operations that are unique to GN for SymNMF in function calls to Compute-Gradient (Algorithm 6, described in Section 3.4.2) and Apply-Gramian (Algorithm 7, described in Section 3.4.2).

Because of the nesting of iterative algorithms, there is overloaded terminology and a clash of standard notation. We use the matrix $\mathbf{H}$ to represent the factor matrix, which is the solution vector of the Gauss-Newton iteration. We use the matrix $\mathbf{X}$ to represent the step direction for the Gauss-Newton iteration, which is also the solution vector of the linear system solved approximately by CG. The matrices $\mathbf{P}$ and $\mathbf{R}$ follow the standard notation of CG and correspond to the step direction and residual of the linear system. Matrix $\mathbf{Y}$ is a temporary variable needed within CG, the output of the single matrix-vector product (computed by Apply-Gramian in our case).

The right-hand-side of the linear system, $\mathbf{b}$ in standard notation, is the gradient of the GN step, which is computed and stored in the residual matrix $\mathbf{R}$ in Line 7. The GN step is taken in Line 22, initially in the direction of $\mathbf{X}$ and then projected onto the set of nonnegative matrices.

Gradient computation

For each GN step, we compute the gradient, which is the right-hand-side vector of the linear system solved approximately by CG. Algorithm 6 shows the parallel algorithm for evaluating the gradient expression derived in Section 3.4.1. The algorithm consists of three matrix multiplications: $\mathbf{HS}$, $\mathbf{HH}^\mathsf{T}$, and $(\mathbf{HH}^\mathsf{T})\mathbf{H}$. The first two multiplications appear in the ANLS algorithm (Algorithm 4). To compute $\mathbf{HH}^\mathsf{T}$, with $\mathbf{H}$ column-distributed,

Algorithm 5 $[\mathbf{W}, \mathbf{H}] = \mathrm{SymGNCG}(\mathbf{S}, k, s_{\max})$

```
Require: $\mathbf{S} \in \mathbb{R}_+^{n \times n}$ is distributed across a $\sqrt{p} \times \sqrt{p}$ grid of processors, $k > 0$ is rank of approximation, $p$ divides $n$
Require: Local matrices: $\mathbf{H}_c, \mathbf{X}_c, \mathbf{P}_c, \mathbf{R}_C, \mathbf{Y}_c$ are $k \times n/p$
Ensure: $\mathbf{H} \approx \mathrm{argmin}_{\tilde{\mathbf{H}} \geq 0} \left\| \mathbf{S} - \tilde{\mathbf{H}}^{\mathsf{T}} \tilde{\mathbf{H}} \right\|_F^2$
Ensure: $\mathbf{H}$ is $k \times n$ columnwise distributed across processors
 1: $\mathbf{p} = (i, j) = p_{ij}$                                   ▷ Processor rank
 2: $r = (i-1)\, p_c + j$                                            ▷ Row-major rank
 3: $c = (j-1)\, p_r + i$                                            ▷ Column-major rank
 4: Initialise $\mathbf{H}_c^{(0)}$
 5: while $t = 1, 2, \ldots$ do                                      ▷ Till some stopping condition is met.
 6:     $\mathbf{X}_c = \mathbf{0}$                                  ▷ Initialise $\mathbf{x}_0 = \mathbf{0}$
 7:     $\mathbf{R}_c$ = Compute-Gradient($\mathbf{S}, \mathbf{H}^{(t)}$)   ▷ $\mathbf{r} = \mathbf{b} - \mathbf{J}^{\mathsf{T}}\mathbf{J}\mathbf{x}_0$
 8:     $\mathbf{P}_c = \mathbf{R}_c$                                ▷ $\mathbf{p} = \mathbf{r}$
 9:     $\epsilon_c^{\text{old}} = \langle \mathbf{R}_c, \mathbf{R}_c \rangle$
10:     $\epsilon^{\text{old}}$ = All-Reduce($\epsilon_c^{\text{old}}$)
11:     for $s = 1$ to $s_{\max}$ do
12:         $\mathbf{Y}_c$ = Apply-Gramian($\mathbf{H}^{(t)}, \mathbf{P}$)     ▷ $\mathbf{y} = \mathbf{J}^{\mathsf{T}}\mathbf{J}\mathbf{p}$
13:         $\alpha_c = \epsilon^{\text{old}} / \langle \mathbf{P}_c, \mathbf{Y}_c \rangle$
14:         $\alpha$ = All-Reduce($\alpha_c$)
15:         $\mathbf{X}_c = \mathbf{X}_c + \alpha \mathbf{P}_c$                ▷ $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
16:         $\mathbf{R}_c = \mathbf{R}_c - \alpha \mathbf{Y}_c$                ▷ $\mathbf{r} = \mathbf{r} - \alpha \mathbf{y}$
17:         $\epsilon_c = \langle \mathbf{R}_c, \mathbf{R}_c \rangle$
18:         $\epsilon$ = All-Reduce($\epsilon_c$)
19:         $\mathbf{P}_c = \mathbf{R}_c + (\epsilon / \epsilon^{\text{old}}) \mathbf{P}_c$   ▷ $\mathbf{p} = \mathbf{r} + \beta \mathbf{p}$
20:         $\epsilon^{\text{old}} = \epsilon$
21:     end for
22:     $\mathbf{H}_{ij} = [\mathbf{H}_c - \mathbf{X}_c]_+$                ▷ projected GN step
23: end while
```

each processor performs a local (symmetric) multiplication followed by an All-Reduce collective. Afterwards, the third multiplication between $\mathbf{H}$ and $\mathbf{H}\mathbf{H}^\mathsf{T}$ can be performed locally. Because the 1D distribution of $\mathbf{H}$ conforms to the 2D distribution of $\mathbf{S}$, the first multiplication involves an All-Gather collective, local multiplication, and a Reduce-Scatter collective. We note that the output matrix $\mathbf{HS}$ is 1D distributed, but in a different order than $\mathbf{H}$, as shown in Fig. 3.4. In order to perform the matrix subtraction, we must redistribute $\mathbf{HS}$ to match $\mathbf{H}$, which can be done via pairwise exchanges between symmetric partners (Line 7).

```
Algorithm 6 $\mathbf{G}$ = Compute-Gradient$(\mathbf{A}, \mathbf{H})$
Require: S $\in \mathbb{R}^{n\times n}$ is distributed across a $\sqrt{p} \times \sqrt{p}$ grid of processors, $\mathbf{H} \in \mathbb{R}^{k\times n}$ is
    column-wise distributed
Ensure: $\mathbf{G} = -2(\mathbf{HS} - \mathbf{HH}^\mathsf{T}\mathbf{H})$ distributed row-wise
 1: $\mathbf{p} = (i,j) = p_{ij}$                                  ▷ Processor rank
 2: $r = (i-1)\,p_c + j$                                           ▷ Row-major rank
 3: $c = (j-1)\,p_r + i$                                           ▷ Column-major rank
    % Compute HS
 4: $\mathbf{H}_j$ = All-Gather($\mathbf{H}_c$,$(:,j)$)
 5: $\mathbf{V}_{ij} = \mathbf{H}_j\mathbf{S}_{ij}$
 6: $(\mathbf{HS})_c$ = Reduce-Scatter($\mathbf{V}_{ij}, (i,:)$)
 7: $p_{ij}$ sends $(\mathbf{HS})_c$ to $p_{ji}$ and receives $(\mathbf{HS})_r$ from $p_{ji}$
    % Compute $\mathbf{HH}^\mathsf{T}$
 8: $\mathbf{U}_c = \mathbf{H}_c\mathbf{H}_c^\mathsf{T}$
 9: $\mathbf{HH}^\mathsf{T}$ = All-Reduce($\mathbf{U}_c$)
    /* Compute $-2(\mathbf{HS} - (\mathbf{HH}^\mathsf{T})\mathbf{H})$ */
10: $\mathbf{G}_c = -2((\mathbf{HS})_c - (\mathbf{HH}^\mathsf{T})\mathbf{H}_c)$
```

Applying Gramian of Jacobian

Algorithm 7 shows the parallel algorithm for applying the Gramian of the Jacobian to a vector (reshaped into a matrix for our case), which is required of every CG iteration. Using identical column-wise distributions of $\mathbf{H}$, the linear-system solution vector $\mathbf{X}$, and the output vector $\mathbf{Y}$, we can perform the four matrix multiplications derived in Section 3.4.1 efficiently in parallel, using only two collective communication operations. Computing the $k \times k$ matrices $\mathbf{HH}^\mathsf{T}$ and $\mathbf{HX}^\mathsf{T}$ requires reducing results across all processors. By using

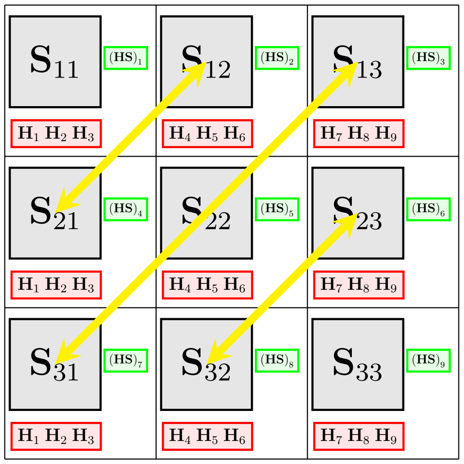


Figure 3.4: Data distribution of GNCG and communication pattern of $\mathbf{H}$ to compute $\mathbf{HS} - (\mathbf{HH}^\mathsf{T})\mathbf{H}$ for a 3×3 processor grid. $\mathbf{HS}$ involves the usual All-Gather across processor columns followed by Reduce-Scatter across the rows. Symmetric swaps, shown by yellow arrows, are needed to match the distributions of $\mathbf{H}$ and $\mathbf{HS}$.

All-Reduce, we can obtain those matrices on all processors in order to multiply the local blocks of $\mathbf{X}$ and $\mathbf{H}$ with those $k \times k$ matrices and perform the addition with no further communication.

Because the Gramian of $\mathbf{H}$ does not change over CG iterations (and it's also used in the computation of the gradient), it does not need to be recomputed for each CG iteration. We employ this optimisation in our implementation but leave it out of the pseudocode for simpilicity. This avoids a computation step (Line 4) and a communication step (Line 5).

Algorithm 7 $\mathbf{Y} = \text{Apply-Gramian}(\mathbf{H}, \mathbf{X})$

```
Require: $\mathbf{H}, \mathbf{X} \in \mathbb{R}^{k \times n}$ are distributed column-wise (identically) across processors
Ensure: $\mathbf{Y} = 2(\mathbf{HH}^\mathsf{T}\mathbf{X} + \mathbf{HX}^\mathsf{T}\mathbf{H})$ distributed column-wise
1: $\mathbf{p} = (i, j) = p_{ij}$                ▷ Processor rank
2: $r = (i-1)\,p_c + j$                          ▷ Row-major rank
3: $c = (j-1)\,p_r + i$                          ▷ Column-major rank
4: $\mathbf{U}_c = \mathbf{H}_c\mathbf{H}_c^\mathsf{T}$
5: $\mathbf{U}$ = All-Reduce($\mathbf{U}_c$)
6: $\mathbf{V}_c = \mathbf{H}_c\mathbf{X}_c^\mathsf{T}$
7: $\mathbf{V}$ = All-Reduce($\mathbf{V}_c$)
8: $\mathbf{Y}_c = 2\mathbf{UX}_c + 2\mathbf{VH}_c$
```

Table 3.1: Per-iteration per-processor costs for dense case.

Algorithm	flops	words	messages
ANLS	$\frac{4n^2k}{p} + O\left(\frac{nk^2}{p}\right)$	$O\left(\frac{nk}{\sqrt{p}} + k^2\right)$	$O\left(\log p\right)$
GNCG	$\frac{2n^2k}{p} + O\left(\frac{s_{\text{max}}nk^2}{p}\right)$	$O\left(\frac{nk}{\sqrt{p}} + s_{\text{max}}k^2\right)$	$O\left(s_{\text{max}} \log p\right)$

3.4.3 Analysis

We now analyse the cost of single GN iteration, which involves s_{max} CG iterations. Aside from the CG iterations, the most expensive operation is the call to Compute-Gradient (Line 7), shown in Algorithm 6. As analysed in the case of ANLS (Section 3.3.4), the cost of computing $\mathbf{HS}$ is $2n^2k/p$ flops (if the matrix is dense), $O(nk/\sqrt{p})$ words, and $O(\log p)$ messages. If the matrix is sparse, the flop cost is $2\text{nnz}(S)k/p$, assuming perfect load balance of the nonzeros. The cost of the extra step of redistribution of $\mathbf{HS}$ is dominated by the costs of the multiplication. As before, computing $\mathbf{HH}^\mathsf{T}$ requires $O(nk^2/p)$ flops, $O(k^2)$ words, and $O(\log p)$ messages, and the final multiplication requires another $O(nk^2)$ flops but no more communication.

The cost of each CG iteration is dominated by the call to Apply-Gramian (Line 12), shown in Algorithm 7. All four local matrix multiplications involve $O(nk^2/p)$ flops, and the two collectives cost $O(k^2)$ words and $O(\log p)$ messages.

Thus, the cost of each GN iteration, assuming a fixed number s_{max} of CG iterations and a dense matrix, is $2n^2k/p + O\left(s_{\text{max}}nk^2/p\right)$ flops, $O\left(nk/\sqrt{p} + s_{\text{max}}k^2\right)$ words, and $O\left(s_{\text{max}} \log p\right)$ messages. We compare the costs of GNCG with ANLS (for the dense case) in Table 3.1.

3.5 Experiments

3.5.1 Experimental Setup

All our experiments were conducted on Summit, a supercomputer at the Oak Ridge Leadership Computing Facility [85]. It is an IBM system comprising 4,600 compute nodes. Each Summit node contains 2 IBM® POWER9® processors on separate sockets. Sockets are connected via dual NVLINK® capable of transferring at 25 GB/s between each other. Nodes are connected to an InfiniBand network providing a node injection bandwidth of 23 GB/s. Each node contains 512 GB of DDR4 memory. Additionally, Summit nodes have 6 NVIDIA® Volta™ V100 accelerators but they are not used by our implementation [2].

PLANC uses the Armadillo library [68] for matrix operations. Armadillo stores dense matrices in column major order and sparse matrices in the Compressed Sparse Column (CSC) format. We link Armadillo (version 9.900) with OpenBLAS (version 0.3.9) and IBM Spectrum MPI (version 10.3.1.2-20200121) for dense BLAS and LAPACK operations and compile using the GNU C++ compiler version 6.4.0.

All the scaling experiments are conducted with flat MPI scaling, that is each core is assigned to a different MPI process. We found the flat setting to run faster and use it as the basis for our scaling experiments.

Beyond reported speedups, we also examined the absolute performance of our implementation by assessing the performance of Armadillo on a single Summit node. In particular, we ran matrix multiply kernels in a manner similar to the flat MPI setting by launching multiple matrix multiply kernels, each bound to a core, on one node. For dense GEMM on large square matrices, Armadillo achieved 63% of the peak flops. However, if one the matrices has a small dimension, as in our experiments, Armadillo instead achieved 43% of peak. This is a reasonable fraction of the peak performance and is close to the 75% achieved in most systems [86].

[2]https://github.com/ramkikannan/planc

In the sparse setting, we compared Armadillo with Eigen [87] for a dense-matrix-times-sparse-matrix kernel. Both libraries performed similarly and we use Armadillo in our experiments. This kernel is expected to be bound by memory bandwidth, especially if the dense matrix is of low rank as in SymNMF. We computed a conservative lower bound on the bandwidth achieved by this kernel and compared that to the sustainable bandwidth reported by the Stream triad benchmark [88]. By "conservative" we mean that we consider only compulsory load traffic, which is the sum of the bytes need to store the input and output matrices, and divide this value by the kernel runtime [89, 90]. This value was found to be 24% of the peak Stream triad bandwidth which is comparable to 10-35% of peak performance cited in earlier studies [91, 92] [3].

Our implementation is constrained to run on square processor grids. Summit nodes have sockets with 21 cores on each socket and the scheduler requires tasks to be divided uniformly across sockets. This forces us to limit the number of processor grid configurations to either use 16 or 18 processors per socket as we scale across nodes. Our experiments scale up to 128 nodes (256 sockets) with MPI processor grid sizes of $1 \times 1, 2 \times 2, 3 \times 3, 4 \times 4, 6 \times 6, 8 \times 8, 12 \times 12, 24 \times 24, 32 \times 32, 48 \times 48$ and 64×64. Here, 16 processors is the largest configuration that can fit in a single socket and 36 processors is the largest that can fit in a node.

3.5.2 Datasets

Pixel Similarity Data [93]

The Pixel Similarity matrix is generated using the Berkeley Segmentation Engine [73]. Each image is flattened to a vector of pixels and a similarity matrix is generated between pixels. The similarity value can be computed based on various factors including brightness, colour and textural cues. We compute similarities only between spatially near-by pixels. This neighborhood is defined by a disk of radius 20 pixels around every pixel [42]. We used

[3]Hong et al. studied this kernel in the context of GPUs [91].

Table 3.2: Pixel Similarity Data.

Image	Image Size	Matrix Size (n)	nonzeros
lighthouse_61_9	1,525×1,419	2,163,975	32,406,651
amusement_park_186_6	2,865×2,535	7,262,775	108,844,443
shipyard_11_1	5,584×4,304	24,033,536	360,325,074

three satellite images from the Functional Map of the World (fMoW) dataset to generate these matrices [76]. We randomly permute the matrices for load balancing. Some salient features of the images are described in Table 3.2.

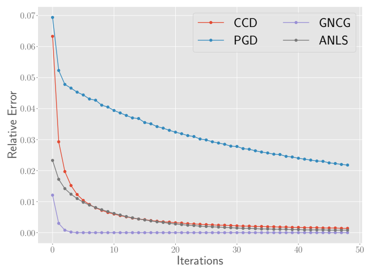
Synthetic

Our synthetic datasets are constructed in two ways depending on whether the input is dense or sparse. For the dense case we generate $\mathbf{S} = \mathbf{H}^\mathsf{T}\mathbf{H}$ where $\mathbf{H}$ is a random low-rank and nonnegative matrix. This is an exact SymNMF model, and we can confirm that the residual error of our algorithm with a random start converges to zero. For benchmarking we run a fixed number of iterations of the SymNMF algorithms rather than till convergence. For sparse inputs, we specify a fixed density of 0.005.

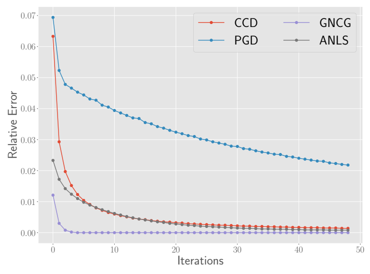
3.5.3 Performance Breakdown

We breakdown the running time of our algorithms into the following categories.

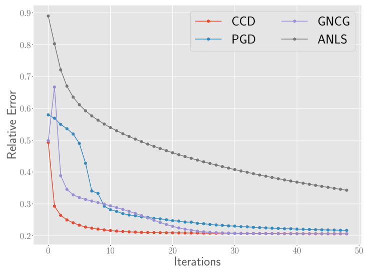
Matrix Multiply: This is the application of the data matrix to the factor matrices for computing $\mathbf{HS}$ (or $\mathbf{W}^\mathsf{T}\mathbf{S}$). These products are needed for the RHS in the nonnegative least squares subproblems in the ANLS version and the gradient in the GNCG. This is further broken into the computation and communication phases. The communication phases are the All-Gather, Reduce-Scatter, and the Sendrecv of the factor matrices. Only the local matrix multiplication call is considered for the computation phase and the Reduce-Scatter computation is counted towards the communication phase. There is only a single Matrix



(a) Random symmetric matrix with $n = 1{,}000$.



(b) SPSD matrix with $n = 1{,}000$ and $k = 50$.



(c) Soybean dataset with $n = 200$ and $k = 6$.

Figure 3.5: Convergence comparison of the sequential SymNMF algorithms.The GNCG relative error is comparable to other SymNMF variants.

Multiply phase per outer iteration in GNCG versus two per outer iteration for the ANLS version.

Solve: This is the rest of the work needed to complete an inner iteration. Primarily, this is constructing the Gramian matrices $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}\mathbf{H}^\mathsf{T}$ and performing the nonnegative least squares solves in the ANLS variant. For GNCG we include the calculations involved in the CG section of the algorithm which include the matrix multiplies involved in Apply-Gram. There are All-Reduce communications involved in this phase but only involve smaller $k \times k$ matrices.

Other: This includes smaller computations like norm checks and applying regularisations which are not explicitly timed in the above phases.

3.5.4 Convergence

We first test the sequential performance of our proposed Gauss-Newton algorithm against other SymNMF variants in Fig. 3.5. We run SymNMF (ANLS) [42], PGD [42] and Cyclic Coordinate Descent (CCD) [69] on three different inputs: random symmetric matrices, low-rank symmetric positive semi-definite (SPSD) matrices, and the Soybean dataset [94]. The SPSD case is an exact low-rank input and all algorithms converge quickly. For the other inputs, which need not have exact low rank, all algorithms perform similarly. Fig. 3.5

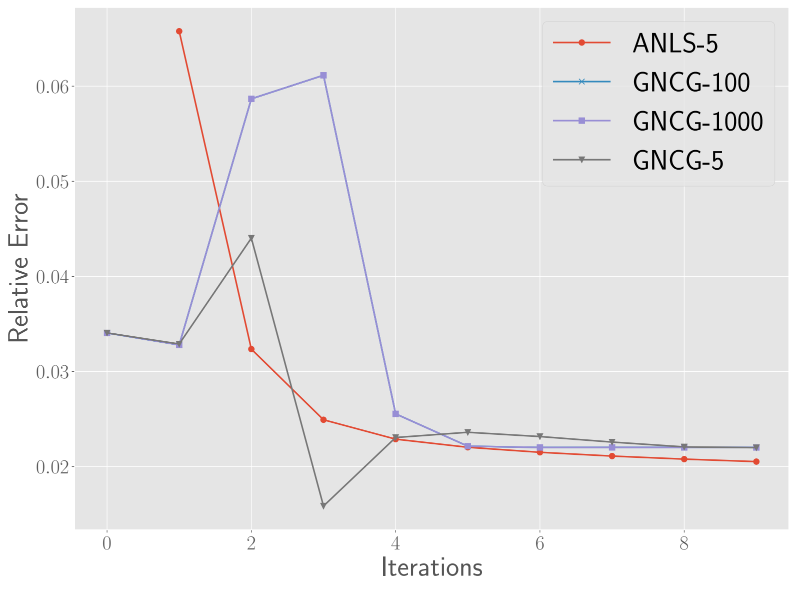


Figure 3.6: Synthetic with $n = 442{,}368$ and $k = 100$ with different number of inner CG iterations. Inner CG iterations does not seem to affect the final relative error.

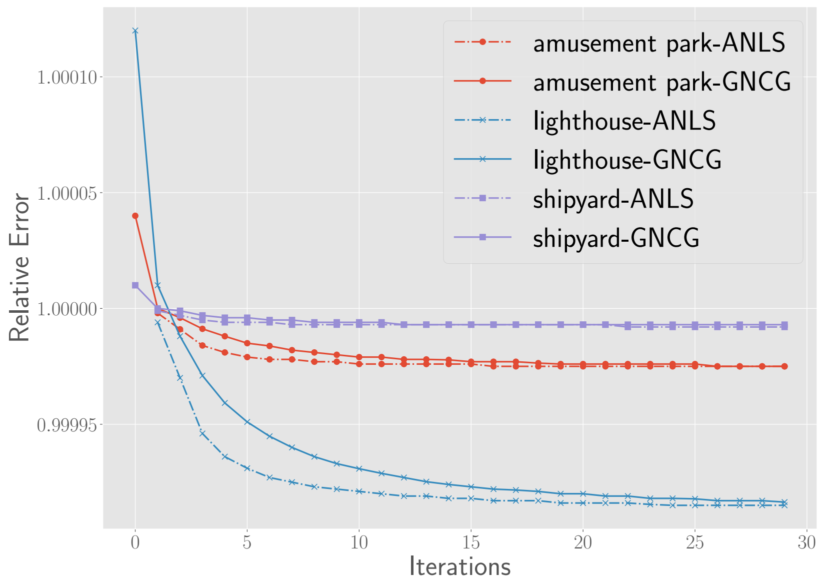


Figure 3.7: Convergence on the Pixel Similarity matrices. Both ANLS and GNCG perform similarly in terms of final relative error.

shows that the Gauss-Newton method is competitive with the other algorithms. The Soybean dataset is taken from the UC Irvine Machine Learning Repository[4]. We apply various standard preprocessing steps, e.g., removing small clusters, resulting in a 200×200 similarity matrix.

Figure 3.6 and Fig. 3.7 shows the convergence of the parallel SymNMF algorithms on large dense synthetic low-rank matrices and the Pixel Similarity data. Since the synthetic matrices are exactly low rank, we expect to see good convergence and small relative errors $\left\|\mathbf{S} - \mathbf{H}^{\mathsf{T}}\mathbf{H}\right\|_F^2 / \|\mathbf{S}\|_F^2$. We test GNCG with different number of inner CG iterations allowed per outer iteration. We specify three different settings where we allow 5, k, or 1,000 inner CG iterations per outer iteration.

Fig. 3.6 shows good convergence across all the algorithm settings on synthetic matrices with $n = 442{,}368$ and $k = 100$. The relative error is calculated as the average of five different initialisations. The different algorithms solve the same problem from the same starting point. The number of CG iterations does not significantly affect the final relative error. The difference between 100 and 1,000 CG iterations is imperceptible in the figure. However, it does affect the execution time: the 5 CG iteration is the fastest of the different settings, though the difference is slight when running time is dominated by Matrix Multiply (performed once per GN iteration). Since this does not greatly affect the relative error, we set $s_{\text{max}} = 5$ for the rest of the experiments.

Fig. 3.7 shows the convergence for 30 iterations on the Pixel Similarity matrices with $k = 16$. This is the rank used to generate embeddings from images in prior work [42, 73]. The relative error is large but decreasing over time. In this case, we seek only to discover embeddings in this task rather than to factor the matrix exactly. Once again, we can see similar performance with both the ANLS and GNCG variants.

[4]https://archive.ics.uci.edu/ml/index.php

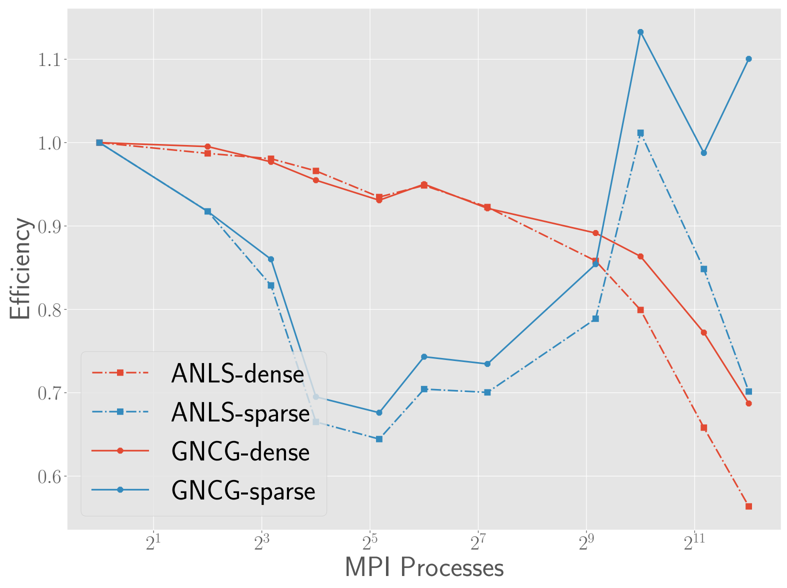
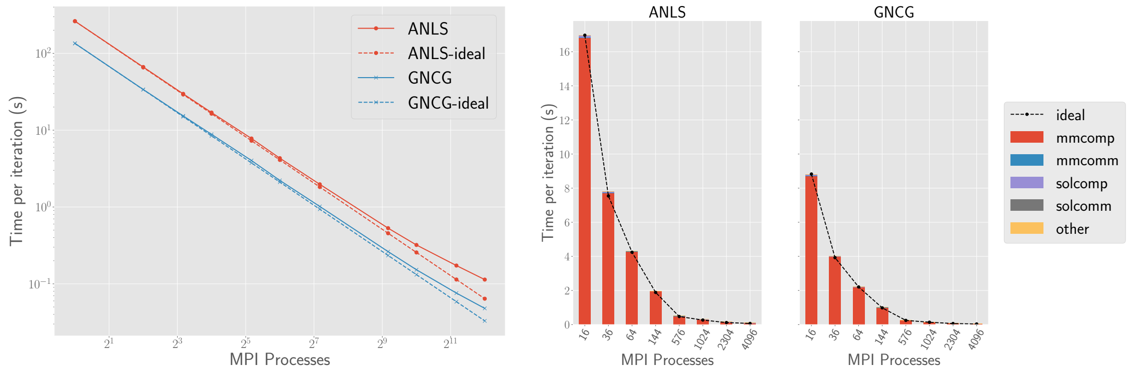


Figure 3.8: Strong scaling efficiency upto 128 nodes of Summit. Initial efficiency drop is noticed once we scale out a socket (16 processes) and cache effects are observed for the sparse case after 576 processes.

3.5.5 Scaling Studies

Strong Scaling

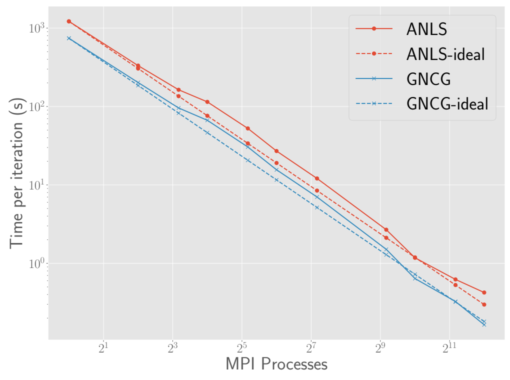
We present the strong-scaling performance for both dense and sparse inputs in Figs. 3.9 and 3.10. Our matrix sizes are chosen to fill up a single socket's memory on Summit. We



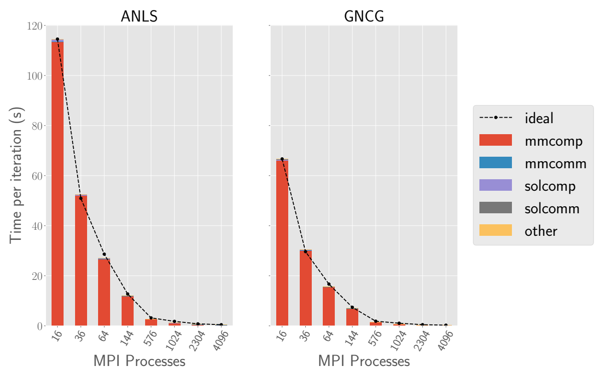
(a) Total time per iteration.

(b) Breakdown.

Figure 3.9: Dense strong scaling with $n = 156{,}401$ and $k = 48$.



(a) Total time per iteration.



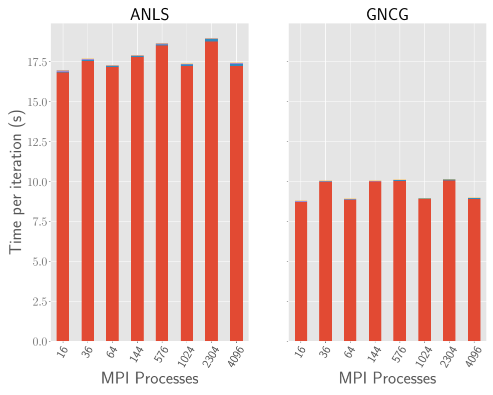
(b) Breakdown.

Figure 3.10: Sparse strong scaling with $n = 884{,}736$, density = 0.005 and $k = 48$.

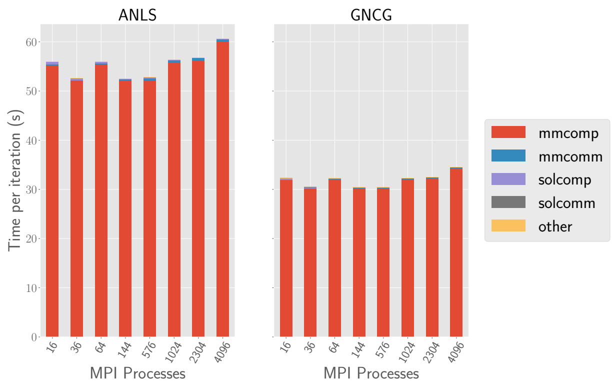
use $n = 156{,}401$ for the dense case and $n = 884{,}736$ for the sparse case with density 0.005. We use a low rank of 48 which is a reasonable size for embeddings.

Figures 3.9a and 3.10a shows the average time per outer iteration of the algorithms as the number of processors is scaled up. In general, the performance scales gracefully up to 4,096 cores, but we can see a noticeable bump when we first span a node (i.e. 36 processes). Figures 3.9b and 3.10b clearly show that the Matrix Multiply time is the dominant cost for both cases. GNCG takes advantage of this fact and is approximately $2\times$ faster than the ANLS variant.

Figure 3.8 shows the scaling efficiency of both cases. The dense case is able to scale gracefully up to 4,096 processes with 55% efficiency for ANLS and 70% efficiency for GNCG. The sparse case behaves more erratically and is able to scale at roughly 70% efficiency till 576 processes. It displays slight superlinear tendency at the larger grid dimensions which we attribute to caching effects as the problem size decays. The problem size of the input matrix is $27{,}648 \times 27{,}648$ for 1,024 processors and is only expected to occupy about 58 MB in memory per process. The SymNMF problem is bandwidth bound for these input dimensions. This is seen clearly in the sparse case as efficiency drops within a socket but stabilises as more sockets are added. This is because bandwidth doesn't increase when we scale within a socket as more cores are used.



(a) Dense $n = 156{,}401 \times \sqrt{\text{nodes}}$.



(b) Sparse $n = 625{,}603 \times \sqrt{\text{nodes}}$ and density=0.005.

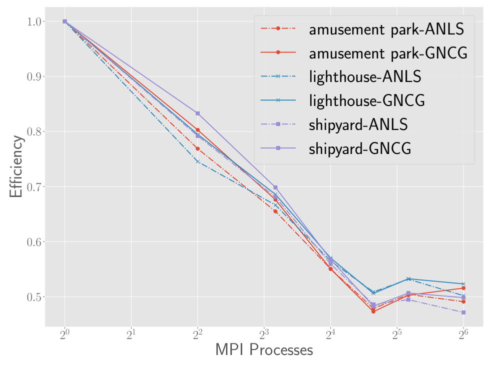
Figure 3.11: Weak scaling with $k = 48$. Efficient configurations for the dense and sparse are different.

Weak Scaling

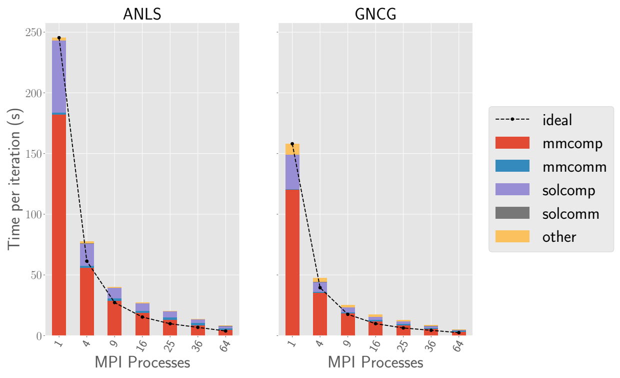
Figure 3.11 demonstrates the weak-scaling performance of our algorithms up to 4,096 processes. The memory per node is kept constant and scales from an initial size of $n_0 = 156{,}401$ for dense inputs and $n_0 = 625{,}603$ for sparse inputs. Matrix dimensions are increased proportionally to the square root of the number of nodes as we scale up. This keeps the local S matrix dimensions constant per processor. Since we expect the computation to be bottlenecked by the matrix multiplication call, we expect to observe flat runtimes. Figure 3.11 confirms this prediction and we see roughly the same performance on all processor grids. We can see two distinct sets of bars in the graphs with one set running slightly faster than the other. In the dense case the faster runs correspond to MPI grids of size 16, 64, 1,024, and 4,096. All these configurations have 16 MPI processes per socket whereas the others have 18 per socket. Interestingly, the opposite effect is seen in the sparse case with the 18 core per socket configurations running slightly faster.

Scaling on Pixel Similarity Matrices

Next, we consider scaling performance on the Pixel Similarity matrices in Fig. 3.12. All the experiments were run with $k = 16$. Fig. 3.12a shows that all the matrices scale similarly.



(a) Efficiency.



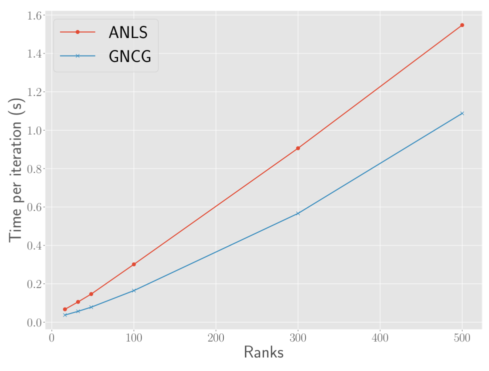
(b) Shipyard.

Figure 3.12: Strong scaling on the Pixel Similarity matrices with $k = 16$.

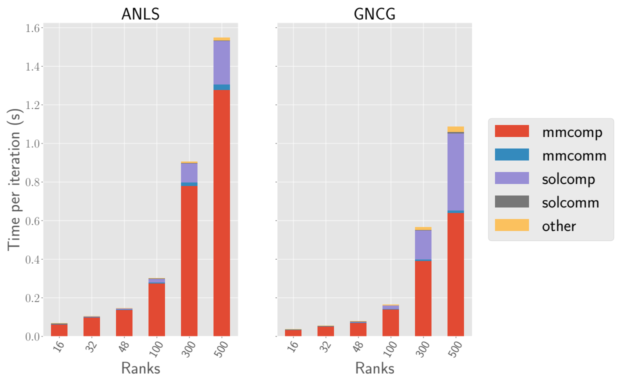
The algorithms steadily lose efficiency till they scale out of a socket and then stabilise at 50% efficiency. This is because bandwidth does not scale with cores within a socket and our algorithm is bandwidth bound. Bandwidth is only increased when scaling out to multiple sockets. Since the algorithm seems to perform similarly on all three inputs, we only show the breakdown for the shipyard image. Unlike the previous scaling runs we can see other components of the algorithm apart from just the matrix multiply. One can see that the matrix multiply is scaling in $O(p)$, but it is not as clear for the solver times. GNCG is still the faster algorithm due to performing fewer matrix multiplies.

3.5.6 Low-rank Sweep

The k-sweep experiment describes the variation in running time when larger low-rank parameters are chosen. For larger matrix dimensions we are completely dominated by the matrix multiply time and these variations do not affect the overall run time. Therefore, we choose a relatively small matrix to conduct this experiment. Fig. 3.13a shows how SymNMF runtimes vary for a dense synthetic matrix of size $14{,}000 \times 14{,}000$. We see a linear increase in runtime as k increases. Fig. 3.13b shows the breakdown of this linear increase. The proportion of solver time increases more rapidly for GNCG than ANLS. This indicates that for cases with extremely large k, it might be better to use ANLS than GNCG. The runtimes for a sparse input with the same memory footprint is similar.

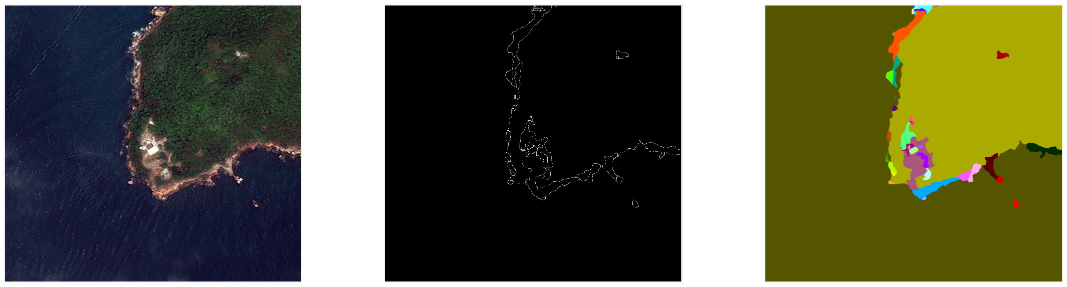


(a) Runtime.



(b) Breakdown.

Figure 3.13: K-Sweep with a dense synthetic matrix with $n = 14{,}000$ on a single node with a 4×4 processor grid.



(a) Original Image (b) Boundary Map (c) Segmented Image

Figure 3.14: Boundary detection and image segmentation using features generated by SymNMF.

3.5.7 Image Segmentation

We recreate the image segmentation experiments described in earlier works [42, 73] albeit on much larger images. The task is to cluster the pixels of an image into a nonoverlapping set of closed regions. Once these regions are discovered we can determine "boundary" pixels which segment the image. The Berkeley Segmentation Engine (BSE) [73] is one of the classical segmentation algorithms used for this task. It represents the pixels in the image as nodes in a graph and defines the segmentation task as a graph partitioning problem. We refer the reader to earlier works for the details on generating such a graph [42, 73]. Spectral clustering [73] is used as the graph partitioning algorithm. In this method, an eigendecomposition is used to generate embeddings for each pixel. We replace those embeddings with the ones produced by the SymNMF algorithm and leave the rest of the pipeline intact. Fig. 3.14 displays the boundary and regions discovered for the lighthouse image.

3.6 Summary

The experiments in the preceding sections show that the proposed SymNMF algorithms perform well both in terms of scaling and low-rank approximations. In comparing ANLS and GNCG, our results show both a BCD and a second-order method can be parallelised efficiently. We did not observe large deviations in convergence between the two methods, so the relative efficiency depends mostly on the per-iteration costs. As seen from our scaling experiments, GNCG runs about twice as fast when the matrix multiply time is dominant, which we expect for large n and small k. However, when the other parts of the algorithm come into play (small n and relatively large k), as is the case for the pixel similarity matrices, ANLS is more competitive, depending on the number of CG iterations used by GNCG.

We also mention some of the limitations of the current work which we hope to address in the future. The first limitation is the use of square processor grids of the form $\sqrt{p} \times \sqrt{p}$, ignoring the symmetry in the input data. Generalising the approach to triangular grids could avoid the redundant storage of $\mathbf{S}$ and possibly enable better load balancing and finding effective communication patterns [95]. Another approach to reducing the constraints on mapping a square grid to the architecture is to develop a hybrid implementation, assigning only one MPI process to each node (or socket) and employing shared-memory parallel subroutines locally.

Though outside the scope of this work, we see many opportunities for further performance optimisation for different classes of sparse matrices from applications outside image segmentation, such as text data and other undirected relationship graphs. For example, in the extremely sparse case we should switch from collective communication to a point-to-point communication scheme [96]. Because the algorithms iteratively apply the matrix, it may also be worth partitioning the matrix data more carefully using graph or hypergraph partitioners to achieve both load balance and low communication costs. As discussed earlier and shown in Fig. 3.12, for large sparse matrices, the matrix multiply times decrease in proportion to the solver times. In light of this, optimisations in the nonnegative least squares solver such as those discussed in prior work could become important to the runtime of the ANLS method [97].

CHAPTER 4
MULTIMODAL INPUTS

We consider the case of clustering when there are multiple sources of input for a data item. In particular, we are working with JointNMF, a hybrid method for mining information from data that contains both feature-data relations and data-data connection structure [72]. JointNMF optimises an integrated objective function that is able to handle these complex datasets without the need for any additional clustering or complicated preprocessing.

We develop the first distributed-memory parallel algorithms for JointNMF. Apart from parallelising the original JointNMF method, we introduce two new methods based on gradient descent and the Gauss-Newton method. As in the SymNMF case, the Gauss-Newton algorithm is a second-order method which is carefully designed to have reasonable computational and memory costs.

We evaluate the scalability of our algorithms on the Phoenix system at Georgia Tech, scaling up to 40 nodes (960 cores) with 40% efficiency.

4.1 Introduction

Numerous datasets, like social networks, document corpora, gene regulatory networks, image datasets, amongst others, can be represented naturally in the form of "attributed graphs" [39, 72]. For example, consider a corpus of documents represented as an attributed graph. Each document forms a node of the graph with citations between patents forming the edges of the graph. The goal is to cluster them using both information sources simultaneously. The text within each document can be used to form the features for each node. JointNMF is a CLRA method and a natural extension of NMF and SymNMF to handle this type of data [72].

JointNMF formulates the optimisation problem as follows,

$$\min_{\mathbf{W}\geq 0,\mathbf{H}\geq 0} \|\mathbf{X}-\mathbf{W}\mathbf{H}\|_F^2 + \alpha \left\|\mathbf{S}-\mathbf{H}^{\mathsf{T}}\mathbf{H}\right\|_F^2, \tag{4.1}$$

where $\mathbf{W} \in \mathbb{R}_{+}^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_{+}^{k \times n}$ are low-rank matrices. The hyperparameter α can be adjusted to emphasise which objective, either the NMF or SymNMF one, is of more importance to the analyst. Fusing the two information sources at the objective level allows us to utilise the CLRA machinery developed for NMF and SymNMF as well as make the results interpretable without any additional preprocessing or clustering steps. Sequential algorithms for Eq. (4.1) are discussed by Du et al. [72]. In particular, they develop an ANLS type algorithm for Eq. (4.1) similar to the one developed for SymNMF (see Section 3.3).

As with SymNMF, we develop both BCD and direct methods for parallel JointNMF. We first parallelise the three-block BCD method developed by Du et al. based on ANLS. This exercise is used to highlight the different computational challenges that arise when there are two large input matrices involved in the matrix multiplications. We explore the different processor grid layouts, optimised for multiplication with $\mathbf{X}$, with $\mathbf{S}$, or both, needed to handle this case. We next develop two direct methods, a PGD algorithm and an inexact Gauss-Newton method. Unlike the Gauss-Newton method developed for SymNMF, (see Section 3.4) this algorithm has stronger guarantees for convergence while still enjoying the low computational and memory requirements as the previous algorithm.

Our serial experiments highlight the superior performance of the ANLS and Gauss-Newton methods when compared to PGD. Finally, we evaluate these algorithms on the Phoenix supercomputer [67], and scale to 40 nodes (960 cores) at 40% efficiency.

4.2 Preliminaries

4.2.1 Notation

The only new notation used in this chapter is the logical indexing used for accessing elements of matrices and vectors. For example, let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathcal{I} = \{(i, j) : 0 \leq \mathbf{A}(i, j) \leq 1\}$. Then $\mathbf{A}(\mathcal{I})$ or $\mathbf{A}_{\mathcal{I}}$ will return all the elements of $\mathbf{A}$ which are between 0 and 1.

4.2.2 Extending to Multiple Inputs

The most common form of JointNMF is the optimisation problem involving a single feature and connection matrix, $\mathbf{X}$ and $\mathbf{S}$, as shown in Eq. (4.1). This formulation can be extended to more than two inputs in a natural way.

$$\min_{\mathbf{W}_1\geq 0,\ldots,\mathbf{W}_p\geq 0,\mathbf{H}\geq 0} \sum_{i=1}^{p} \gamma_i \left\|\mathbf{X} - \mathbf{W}_i\mathbf{H}\right\|_F^2 + \sum_{j=1}^{q} \alpha_j \left\|\mathbf{S}_j - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2 \tag{4.2}$$

Note that the $\mathbf{H}$ is common to all terms in the objective. Using this joint formula, we are able to incorporate all the input sources at the objective level and obtain a single *embedding* matrix. Clustering and other data mining tasks can now work off this single embedding.

We shall now show the equivalence between Eq. (4.2) and Eq. (4.1). Looking at the features term, we can combine the p different terms as follows.

$$\begin{aligned}
\sum_{i=1}^{p} \gamma_i \left\|\mathbf{X} - \mathbf{W}_i\mathbf{H}\right\|_F^2 &= \left\| \begin{bmatrix} \sqrt{\gamma_1}\mathbf{X}_1 - \sqrt{\gamma_1}\mathbf{W}_1\mathbf{H} \\ \vdots \\ \sqrt{\gamma_p}\mathbf{X}_p - \sqrt{\gamma_p}\mathbf{W}_p\mathbf{H} \end{bmatrix} \right\|_F^2 \\
&= \left\| \begin{bmatrix} \sqrt{\gamma_1}\mathbf{X}_1 \\ \vdots \\ \sqrt{\gamma_p}\mathbf{X}_p \end{bmatrix} - \begin{bmatrix} \sqrt{\gamma_1}\mathbf{W}_1 \\ \vdots \\ \sqrt{\gamma_p}\mathbf{W}_p \end{bmatrix} \mathbf{H} \right\|_F^2 \\
&= \left\| \hat{\mathbf{X}} - \hat{\mathbf{W}}\mathbf{H} \right\|_F^2
\end{aligned}$$

Thus, replacing $\mathbf{X}$ with the block matrix $\hat{\mathbf{X}}$ leaves the optimisation problem unchanged.

Utilising the same trick for the connection matrices is not as straightforward since the term $\mathbf{H}^\mathsf{T}\mathbf{H}$ cannot be blocked in a similar manner. We shall show this by induction by working with two matrices, $\mathbf{S}_1$ and $\mathbf{S}_2$ first. Let $\mathbf{Y} = \mathbf{H}^\mathsf{T}\mathbf{H}$, $\hat{\alpha} = \alpha_1 + \alpha_2$, and $\hat{\mathbf{S}} = \frac{\alpha_1\mathbf{S}_1 + \alpha_2\mathbf{S}_2}{\alpha_1+\alpha_2}$. Now let us look at the difference $\hat{\alpha}\left\|\hat{\mathbf{S}} - \mathbf{Y}\right\|_F^2 - \sum_{j=1}^{2} \alpha_j \left\|\mathbf{S}_j - \mathbf{Y}\right\|_F^2$. Expanding out the

(i, j) entry we have,

$$
\begin{aligned}
r_{ij} &= (\alpha_1 + \alpha_2) \left(\hat{\mathbf{S}}\,(i,j) - \mathbf{Y}\,(i,j) \right)^2 - \alpha_1 \left(\mathbf{S}_1\,(i,j) - \mathbf{Y}\,(i,j) \right)^2 - \alpha_2 \left(\mathbf{S}_2\,(i,j) - \mathbf{Y}\,(i,j) \right)^2 \\
&= (\alpha_1 + \alpha_2)\, \hat{\mathbf{S}}\,(i,j)^2 - \alpha_1 \mathbf{S}_1\,(i,j)^2 - \alpha_2 \mathbf{S}_2\,(i,j)^2 + (\alpha_1 + \alpha_2 - \alpha_1 - \alpha_2)\, \mathbf{Y}\,(i,j)^2 \\
&\qquad - 2 \left((\alpha_1 + \alpha_2)\, \hat{\mathbf{S}}\,(i,j) - \alpha_1 \mathbf{S}_1\,(i,j) - \alpha_2 \mathbf{S}_2\,(i,j) \right) \mathbf{Y}\,(i,j) \\
&= \frac{\alpha_1^2 \mathbf{S}_1\,(i,j)^2 + \alpha_2^2 \mathbf{S}_2\,(i,j)^2 + 2\alpha_1\alpha_2 \mathbf{S}_1\,(i,j)\, \mathbf{S}_2\,(i,j)}{\alpha_1 + \alpha_2} - \alpha_1 \mathbf{S}_1\,(i,j)^2 - \alpha_2 \mathbf{S}_2\,(i,j)^2 \\
&= -\alpha_1 \alpha_2 \left(\mathbf{S}_1\,(i,j) - \mathbf{S}_2\,(i,j) \right)^2 \ .
\end{aligned}
$$

The difference only depends on $\mathbf{S}_1$ and $\mathbf{S}_2$ and not $\mathbf{Y} = \mathbf{H}^{\mathsf{T}}\mathbf{H}$. Therefore, solving with $\hat{\mathbf{S}}$ in Eq. (4.1) will result in the same solution as using $\mathbf{S}_1$ and $\mathbf{S}_2$ in Eq. (4.2). The objective values will differ by a constant, which does not affect the solution $\mathbf{H}$. By induction, we can replace the q inputs $(\alpha_i, \mathbf{S}_i)$ with a single input $\left(\sum_{j=1}^{q} \alpha_j, \frac{\sum_{j=1}^{q} \alpha_j \mathbf{S}_j}{\sum_{j=1}^{q} \alpha_j} \right)$.

Therefore, the original formulation, Eq. (4.1), is versatile enough to handle multiple information sources via some simple preprocessing. We concentrate only on this case for parallelisation.

4.2.3 Convergent Two-Metric Projection Methods

In Section 3.2.3, we described a two-metric projection method for a generic problem and mentioned its lack of convergence guarantees. However, when the constraints are relatively simple box-constraints of the form $l \leq x \leq u$, there exists a class of scaling matrices that achieve convergence [43]. Nonnegativity constraints fall under this category, and we briefly describe how to modify the scaling matrix $\mathbf{D} = \mathbf{J}^{\mathsf{T}}\mathbf{J}$ to achieve convergence.

Recall that the setup to our problem is exactly like Section 3.2.3. We wish to minimise multivariate functions of the form $f(\mathbf{x}) = \frac{1}{2} \sum_i r_i\,(\mathbf{x})^2$ subject to $\mathbf{x} \geq 0$. We update an initial guess $\mathbf{x}^{(0)}$ along the step direction $\mathbf{p}$, as $\mathbf{x}^{(t+1)} = \left[\mathbf{x}^{(t)} - \lambda \mathbf{p} \right]_+$, such that

$$
\left(\mathbf{J}^{(t)\mathsf{T}} \mathbf{J}^{(t)} \right) \mathbf{p} = \mathbf{J}^{(t)} \mathbf{r}^{(t)},
$$

where $\mathbf{J}^{(t)}$ and $\mathbf{r}^{(t)}$ are the Jacobian and residual at iteration t and $\lambda > 0$ is the step size. In the unconstrained case, $\mathbf{p}$ takes the form of a scaling of the gradient at iteration t,

$$\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)^{-1}\mathbf{J}^{(t)}\mathbf{r}^{(t)} = \mathbf{D}^{(t)}\mathbf{g}^{(t)} .$$

To make this procedure converge, we need to make $\mathbf{D}^{(t)}$ diagonal with respect to variables whose constraints are *active* [98]. Let us identify these active variables as $\mathcal{A} = \left\{i : 0 \leq x_i \leq \epsilon, \frac{\partial f}{\partial x_i} > 0\right\}$ where ϵ is a small constant. The complement of $\mathcal{A}$ is the set of free variables $\mathcal{F} = \{1, \ldots, n\} \setminus \mathcal{A}$. Without loss of generality, let us assume $\mathbf{x}$ is permuted like $\begin{bmatrix} \mathbf{x}_{\mathcal{F}} \\ \mathbf{x}_{\mathcal{A}} \end{bmatrix}$. Then the step direction can be found by solving

$$\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)\begin{bmatrix} \mathbf{p}_{\mathcal{F}} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_{\mathcal{F}}^{(t)} \\ \mathbf{0} \end{bmatrix} \tag{4.3}$$

$$\mathbf{p}_{\mathcal{A}} = \mathbf{g}_{\mathcal{A}} . \tag{4.4}$$

The gradient scaling matrix $\mathbf{D}^{(t)}$ takes the following form

$$\mathbf{D}^{(t)} = \begin{bmatrix} \mathbf{D}_{\mathcal{F}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{\mathcal{A}} \end{bmatrix} .$$

Second-order information is captured for the free variables whereas the active variables are kept fixed at the constraints. Bertsekas shows that this class of scaling matrices result in $\mathbf{p}$ being a descent direction, and with an appropriately chosen step size λ, such an iterative procedure converges to a stationary point [98].

4.2.4 Related Work

The fusion of multiple information sources is a common topic in CLRA literature. Various formulations of fusing multiple feature and connection matrices with different constraints have been proposed in the clustering [72, 99–101] and anomaly detection [39, 102, 103] contexts. In both settings the fusion of information sources has been shown to be more effective than working on individual portions of the data. Most of the clustering methods are for unsupervised graph mining, and Whang et al. [101] have extended this approach to hypergraphs and to the semi-supervised case. For more information on the different formulations, we refer the reader elsewhere [104].

The PGD and projected Gauss-Newton methods are staples of modern optimisation [43, 78] and have been used for CLRA [48, 82, 105]. In this chapter, we are primarily focused on the momentum variant of PGD [106, 107] and the box-constrained version of Gauss-Newton [98]. While these methods have been developed for general optimisation, to the best of our knowledge our approach is the first known treatment of these methods for JointNMF in the distributed-memory setting.

4.3 Parallel JointNMF

We give a brief description of the different methods implemented in PLANC for JointNMF. They have a similar vein to the algorithms described in Sections 2.3.3, 3.3 and 3.4 and so, we highlight only the major differences here.

4.3.1 JointNMF via ANLS

Du et al. proposed a way to solve Eq. (4.1) by dropping the symmetric constraint and using a penalty term [72]. They propose the surrogate optimisation,

$$\min_{\mathbf{W}\geq 0, \mathbf{H}\geq 0, \hat{\mathbf{H}}\geq 0} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 + \alpha \left\|\mathbf{S} - \hat{\mathbf{H}}^\mathsf{T}\mathbf{H}\right\|_F^2 + \beta \left\|\hat{\mathbf{H}} - \mathbf{H}\right\|_F^2 , \tag{4.5}$$

where $\hat{\mathbf{H}} \in \mathbb{R}_+^{k \times n}$ and $\beta \geq 0$ is the regularisation parameter. This formulation is motivated by a similar construction for the SymNMF problem (see Section 3.3). This reformulation can be solved using a three-block BCD scheme, updating $\mathbf{W}, \hat{\mathbf{H}}$, and $\mathbf{H}$ in turn. The following NLS subproblems are iteratively solved.

$$\min_{\mathbf{W} \geq 0} \left\| \mathbf{H}^\mathsf{T} \mathbf{W}^\mathsf{T} - \mathbf{X}^\mathsf{T} \right\|_F^2 \tag{4.6}$$

$$\min_{\hat{\mathbf{H}} \geq 0} \left\| \begin{bmatrix} \sqrt{\alpha}\mathbf{H}^\mathsf{T} \\ \sqrt{\beta}\mathbf{I}_k \end{bmatrix} \hat{\mathbf{H}} - \begin{bmatrix} \sqrt{\alpha}\mathbf{S} \\ \sqrt{\beta}\mathbf{H} \end{bmatrix} \right\|_F^2 \tag{4.7}$$

$$\min_{\mathbf{H} \geq 0} \left\| \begin{bmatrix} \mathbf{W} \\ \sqrt{\alpha}\hat{\mathbf{H}}^\mathsf{T} \\ \sqrt{\beta}\mathbf{I}_k \end{bmatrix} \mathbf{H} - \begin{bmatrix} \mathbf{X} \\ \sqrt{\alpha}\mathbf{S} \\ \sqrt{\beta}\hat{\mathbf{H}} \end{bmatrix} \right\|_F^2 \tag{4.8}$$

The major computations for this ANLS version of JointNMF are the matrix calculations needed for computing the gradients. These are the Gram calculations ($\mathbf{H}\mathbf{H}^\mathsf{T}$, $\alpha\mathbf{H}\mathbf{H}^\mathsf{T} + \beta\mathbf{I}_k$, and $\mathbf{W}^\mathsf{T}\mathbf{W} + \alpha\hat{\mathbf{H}}\hat{\mathbf{H}}^\mathsf{T} + \beta\mathbf{I}_k$) and the large ones involving the input matrices ($\mathbf{H}\mathbf{X}^\mathsf{T}$, $\alpha\mathbf{H}\mathbf{S} + \beta\mathbf{H}$, $\mathbf{W}^\mathsf{T}\mathbf{X} + \alpha\hat{\mathbf{H}}\mathbf{S} + \beta\hat{\mathbf{H}}$). Parallelising these kernels is the major focus of this chapter.

4.3.2 JointNMF via PGD

PGD is a straightforward way to tackle Eq. (4.1). The gradients with respect to the factor matrices (see Section 4.3.3) are

$$\nabla_{\mathbf{W}} f = 2\left(\mathbf{W}\mathbf{H}\mathbf{H}^\mathsf{T} - \mathbf{X}\mathbf{H}^\mathsf{T}\right) \tag{4.9}$$

$$\nabla_{\mathbf{H}} f = 2\left(\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H} - \mathbf{W}^\mathsf{T}\mathbf{X}\right) + 4\alpha\left(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{H} - \mathbf{H}\mathbf{S}\right) . \tag{4.10}$$

Let $\mathbf{P}_{\mathbf{W}}^{(t)}$ and $\mathbf{P}_{\mathbf{H}}^{(t)}$ be the steps taken by the algorithm at iteration t for the factors $\mathbf{W}$ and $\mathbf{H}$, respectively. Then the updates via PGD take the following form.

$$\mathbf{P}_{\mathbf{W}}^{(t)} = \gamma \mathbf{P}_{\mathbf{W}}^{(t-1)} + \frac{\nabla_{\mathbf{W}} f}{\|\nabla_{\mathbf{W}} f\|_F} \tag{4.11}$$

$$\mathbf{W}^{(t)} = \mathbf{W}^{(t-1)} - \lambda \mathbf{P}_{\mathbf{W}}^{(t)} \tag{4.12}$$

$$\mathbf{P}_{\mathbf{H}}^{(t)} = \gamma \mathbf{P}_{\mathbf{H}}^{(t-1)} + \frac{\nabla_{\mathbf{H}} f}{\|\nabla_{\mathbf{H}} f\|_F} \tag{4.13}$$

$$\mathbf{H}^{(t)} = \mathbf{H}^{(t-1)} - \lambda \mathbf{P}_{\mathbf{H}}^{(t)} \tag{4.14}$$

Here, $\gamma \geq 0$ is the momentum parameter, which helps accelerate gradient descent in the relevant direction and dampens oscillations [106, 107]. The parameter $\lambda \geq 0$ is the step size, which is found via experiment. We use a variant of backtracking line search to step in a direction which reduces the objective [43, 105]. Notice that computational bottlenecks are a subset of matrix multiplications as in the ANLS case, namely, $\mathbf{W}^\mathsf{T}\mathbf{W}, \mathbf{H}\mathbf{H}^\mathsf{T}, \mathbf{W}^\mathsf{T}\mathbf{X}, \mathbf{X}\mathbf{H}^\mathsf{T}$, and $\mathbf{HS}$.

4.3.3 JointNMF via PGNCG

We describe the Projected Gauss-Newton method using Conjugate Gradients (PGNCG) method for the JointNMF objective function in Eq. (4.1). This section follows closely that of Section 3.4, so we sketch the various derivations. Let us define the vectorised residuals of the different parts of the objective as

$$\mathbf{r} = \begin{bmatrix} \mathbf{r}_\mathbf{X} \\ \mathbf{r}_\mathbf{S} \end{bmatrix} = \begin{bmatrix} \mathrm{vec}(\mathbf{X} - \mathbf{WH}) \\ \sqrt{\alpha}\mathrm{vec}\big(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\big) \end{bmatrix} \in \mathbb{R}^{mn+n^2} . \tag{4.15}$$

The Jacobian for Eq. (4.1) is a 2×2 block matrix of the form

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{r}_\mathbf{X}}{\partial \mathrm{vec}(\mathbf{W})} & \frac{\partial \mathbf{r}_\mathbf{X}}{\partial \mathrm{vec}(\mathbf{H})} \\ \frac{\partial \mathbf{r}_\mathbf{S}}{\partial \mathrm{vec}(\mathbf{W})} & \frac{\partial \mathbf{r}_\mathbf{S}}{\partial \mathrm{vec}(\mathbf{H})} \end{bmatrix} = - \begin{bmatrix} \mathbf{H}^\mathsf{T} \otimes \mathbf{I}_m & \mathbf{I}_n \otimes \mathbf{W} \\ \mathbf{0} & \sqrt{\alpha}\left(\left(\mathbf{I}_n \otimes \mathbf{H}^\mathsf{T}\right) + \left(\mathbf{H}^\mathsf{T} \otimes \mathbf{I}_n\right)\mathbf{P}_{k,n}\right) \end{bmatrix}.$$

We use the identities, Eq. (3.10), to derive an expression for $\mathbf{J} \in \mathbb{R}^{\left(mn+n^2\right)\times(mk+nk)}$. First, let us verify that $2\mathbf{J}^\mathsf{T}\mathbf{r}$ gives the gradient (see Section 3.4.1).

$$\begin{aligned}
2\mathbf{J}^\mathsf{T}\mathbf{r} &= -2\begin{bmatrix} \mathbf{H}^\mathsf{T}\otimes\mathbf{I}_m & \mathbf{I}_n\otimes\mathbf{W} \\ \mathbf{0} & \sqrt{\alpha}\left(\left(\mathbf{I}_n\otimes\mathbf{H}^\mathsf{T}\right)+\left(\mathbf{H}^\mathsf{T}\otimes\mathbf{I}_n\right)\mathbf{P}_{k,n}\right)\end{bmatrix}^\mathsf{T}\begin{bmatrix}\mathrm{vec}(\mathbf{X}-\mathbf{W}\mathbf{H}) \\ \sqrt{\alpha}\mathrm{vec}\left(\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right)\end{bmatrix} \\
&= -2\begin{bmatrix} \mathbf{H}\otimes\mathbf{I}_m & \mathbf{0} \\ \mathbf{I}_n\otimes\mathbf{W}^\mathsf{T} & \sqrt{\alpha}\left(\mathbf{P}_{n,k}\left(\mathbf{I}_n\otimes\mathbf{H}\right)+\left(\mathbf{H}\otimes\mathbf{I}_n\right)\right)\end{bmatrix}\begin{bmatrix}\mathrm{vec}(\mathbf{X}-\mathbf{W}\mathbf{H}) \\ \sqrt{\alpha}\mathrm{vec}\left(\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right)\end{bmatrix} \\
&= -2\begin{bmatrix} \left(\mathbf{H}\otimes\mathbf{I}_m\right)\mathrm{vec}(\mathbf{X}-\mathbf{W}\mathbf{H}) \\ \left(\mathbf{I}_n\otimes\mathbf{W}^\mathsf{T}\right)\mathrm{vec}(\mathbf{X}-\mathbf{W}\mathbf{H})+\alpha\left(\left(\mathbf{P}_{n,k}\left(\mathbf{I}_n\otimes\mathbf{H}\right)+\left(\mathbf{H}\otimes\mathbf{I}_n\right)\right)\mathrm{vec}\left(\mathbf{S}-\mathbf{H}^\mathsf{T}\mathbf{H}\right)\right)\end{bmatrix} \\
&= 2\begin{bmatrix} \mathrm{vec}\left(\mathbf{W}\mathbf{H}\mathbf{H}^\mathsf{T}-\mathbf{X}\mathbf{H}^\mathsf{T}\right) \\ \mathrm{vec}\left(\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H}-\mathbf{W}^\mathsf{T}\mathbf{X}\right)+2\alpha\mathrm{vec}\left(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{H}-\mathbf{H}\mathbf{S}\right)\end{bmatrix} \\
&= \begin{bmatrix}\mathrm{vec}(\nabla_\mathbf{W} f) \\ \mathrm{vec}(\nabla_\mathbf{H} f)\end{bmatrix} = \mathbf{g}
\end{aligned}$$

Similarly, applying $\mathbf{J}^\mathsf{T}\mathbf{J}$ to a vector $\mathbf{x} = \begin{bmatrix}\mathrm{vec}(\mathbf{X_W}) \\ \mathrm{vec}(\mathbf{X_W})\end{bmatrix}$ results in

$$\mathbf{y} = \begin{bmatrix}\mathrm{vec}(\mathbf{Y_W}) \\ \mathrm{vec}(\mathbf{Y_H})\end{bmatrix} \tag{4.16}$$

$$= \mathbf{J}^\mathsf{T}\mathbf{J}\mathbf{x} \tag{4.17}$$

$$= \mathbf{J}^\mathsf{T}\mathbf{J}\begin{bmatrix}\mathrm{vec}(\mathbf{X_W}) \\ \mathrm{vec}(\mathbf{X_H})\end{bmatrix} \tag{4.18}$$

$$= \begin{bmatrix}\mathrm{vec}\left(\mathbf{X_W}\mathbf{H}\mathbf{H}^\mathsf{T}+\mathbf{W}\mathbf{X_H}\mathbf{H}^\mathsf{T}\right) \\ \mathrm{vec}\left(\mathbf{W}^\mathsf{T}\mathbf{X_W}\mathbf{H}+\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H}\right)+2\alpha\mathrm{vec}\left(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{X_H}+\mathbf{H}^\mathsf{T}\mathbf{X_H}\mathbf{H}\right).\end{bmatrix} \tag{4.19}$$

Computing the gradient requires multiplication with $\mathbf{X}$ and $\mathbf{S}$ whereas applying the Gramian of the Jacobian involves only $m \times k$, $n \times k$, and $k \times k$ matrices.

Now we need to perform the PGNCG step, which is different from the algorithm used in SymNMF. Just like the SymNMF case, we drop the extra constant and work with $\mathbf{J}^{\mathsf{T}}\mathbf{r}$ and $\mathbf{J}^{\mathsf{T}}\mathbf{J}$ for the PGNCG step (see Section 3.4.1). First, identifying the active-set can be done independently at every processor via checking their local portions of the factor matrices and gradients, as follows.

$$\mathcal{A}_{\mathbf{W}} = \{(i,j) : 0 \leq \mathbf{W}(i,j) \leq \epsilon, \nabla_{\mathbf{W}} f(i,j) > 0\} \tag{4.20}$$

$$\mathcal{A}_{\mathbf{H}} = \{(i,j) : 0 \leq \mathbf{H}(i,j) \leq \epsilon, \nabla_{\mathbf{H}} f(i,j) > 0\} \tag{4.21}$$

The complements of the active-sets are the free variables $\mathcal{F}_{\mathbf{W}}$ and $\mathcal{F}_{\mathbf{H}}$. Instead of using the exact active-set with $\mathbf{H}(i,j) = 0$, we use a small fixed scalar ϵ to prevent zigzagging of the solution [43]. To compute the correct step direction, before starting the CG iterations we mask the gradients as $\nabla_{\mathbf{W}} f(\mathcal{A}_{\mathbf{W}}) = 0$ and $\nabla_{\mathbf{H}} f(\mathcal{A}_{\mathbf{H}}) = 0$. Every time we apply $\mathbf{J}^{\mathsf{T}}\mathbf{J}$, we similarly mask the outputs $\mathbf{Y}_{\mathbf{W}}(\mathcal{A}_{\mathbf{W}}) = 0$ and $\mathbf{Y}_{\mathbf{H}}(\mathcal{A}_{\mathbf{H}}) = 0$. Finally, the step directions of the free variables are set to the gradient [82, 98, 105].

4.3.4 Parallel Matrix Multiplication Options

While the individual algorithms might seem complicated, the bottleneck computations are still the matrix multiplications with the input matrices $\mathbf{X}$ and $\mathbf{S}$. The different products to be calculated are $\mathbf{W}^{\mathsf{T}}\mathbf{X}$, $\mathbf{H}\mathbf{X}^{\mathsf{T}}$, $\mathbf{HS}$, and in case of the ANLS variant $\hat{\mathbf{H}}^{\mathsf{T}}\mathbf{S}$. From our discussion in Section 2.3.2, we know that for a 2D communication-optimal algorithm we need the processor grid to have the same aspect ratio as the input matrix, i.e, $\frac{p_r}{p_c} \approx \frac{m}{n}$ when multiplying with $\mathbf{X} \in \mathbb{R}_{+}^{m \times n}$. However, to be communication-optimal with respect to $\mathbf{S}$ we need $\frac{p_r}{p_c} \approx 1$. To understand these trade-offs, we shall systematically evaluate the grid and data layout choices. Throughout this section we shall work with the ANLS variant, but these arguments can easily be modified for the PGD and PGNCG case.

4.3.5 Grid Choices

The communication costs for the four multiplications are given by Eq. (4.25) assuming we are working with a $p_r \times p_c$ grid of p processors[1].

$$T_{\text{comm}}\left(\mathbf{W}^{\mathsf{T}}\mathbf{X}\right) = 2\alpha \log p + \beta\left(\frac{mk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right) \tag{4.22}$$

$$T_{\text{comm}}\left(\mathbf{H}\mathbf{X}^{\mathsf{T}}\right) = 2\alpha \log p + \beta\left(\frac{mk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right) \tag{4.23}$$

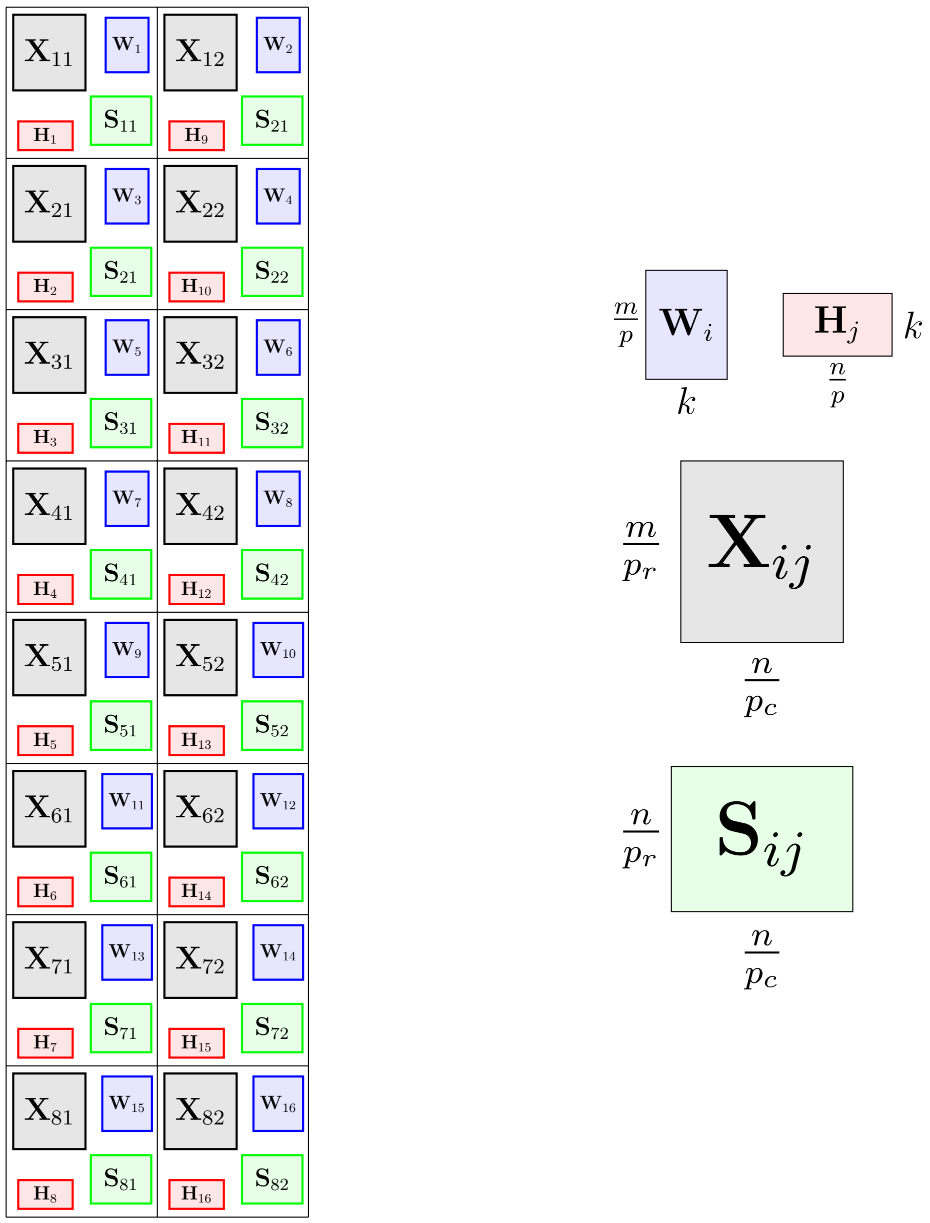
$$T_{\text{comm}}\left(\hat{\mathbf{H}}^{\mathsf{T}}\mathbf{S}\right) = 2\alpha \log p + \beta\left(\frac{nk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right) \tag{4.24}$$

$$T_{\text{comm}}\left(\mathbf{H}\mathbf{S}\right) = 2\alpha \log p + \beta\left(\frac{nk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right) \tag{4.25}$$

The total words communicated is $\frac{(2m+2n)k}{p}(p_c - 1) + \frac{4nk}{p}(p_r - 1)$, which is the same as a 2D multiplication with a large matrix of size $(2m + 2n) \times 4n$. Thus, a processor grid with aspect ratio $\frac{p_r}{p_c} \approx \frac{2m+2n}{4n} = \frac{m+n}{2n}$ will be communication efficient. Similarly, for the PGD and PGNCG algorithms we get $\frac{p_r}{p_c} \approx \frac{2m+n}{3n}$. This data distribution is shown in Fig. 4.1.

An alternate technique would be to logically partition the p processors into two grids: $p = p_r \times p_c = q_r \times q_c$. The aspect ratio of one grid could approximate that of $\mathbf{X}$ whereas the other would be closer to that of $\mathbf{S}$, as shown in Fig. 4.2. Thus, we could be theoretically communication-optimal for all matrix multiplications. However, the factor $\mathbf{H}$ must be duplicated on both grids since it needs to be multiplied by both $\mathbf{X}$ and $\mathbf{S}$. Care must be taken to ensure that these copies of $\mathbf{H}$ are kept in sync as the algorithms progress.

[1]We can eliminate an extra All-Gather term from either $\mathbf{HS}$ or $\mathbf{HX}^{\mathsf{T}}$ by carefully selecting the update order to ensure that $\mathbf{H}$ is updated last in every inner iteration. We ignore this optimisation in our analysis but it is straightforward to include. The optimal aspect ratio changes to $\frac{2m+n}{4n}$ in this case.



X_{11} W_1 H_1 S_{11} X_{12} W_2 H_9 S_{21}
X_{21} W_3 H_2 S_{21} X_{22} W_4 H_{10} S_{22}
X_{31} W_5 H_3 S_{31} X_{32} W_6 H_{11} S_{32}
X_{41} W_7 H_4 S_{41} X_{42} W_8 H_{12} S_{42}
X_{51} W_9 H_5 S_{51} X_{52} W_{10} H_{13} S_{52}
X_{61} W_{11} H_6 S_{61} X_{62} W_{12} H_{14} S_{62}
X_{71} W_{13} H_7 S_{71} X_{72} W_{14} H_{15} S_{72}
X_{81} W_{15} H_8 S_{81} X_{82} W_{16} H_{16} S_{82}
$\frac{m}{p}$ W_i k
H_j k $\frac{n}{p}$
$\frac{m}{p_r}$ X_{ij} $\frac{n}{p_c}$
$\frac{n}{p_r}$ S_{ij} $\frac{n}{p_c}$

Figure 4.1: PLANC data distribution of the JointNMF matrices on a single 8×2 processor grid. The two large input matrices $\mathbf{X} \in \mathbb{R}_+^{m \times n}$ and $\mathbf{S} \in \mathbb{R}_+^{n \times n}$ are 2D partitioned across the processor grid whereas as the factor matrices are 1D partitioned.

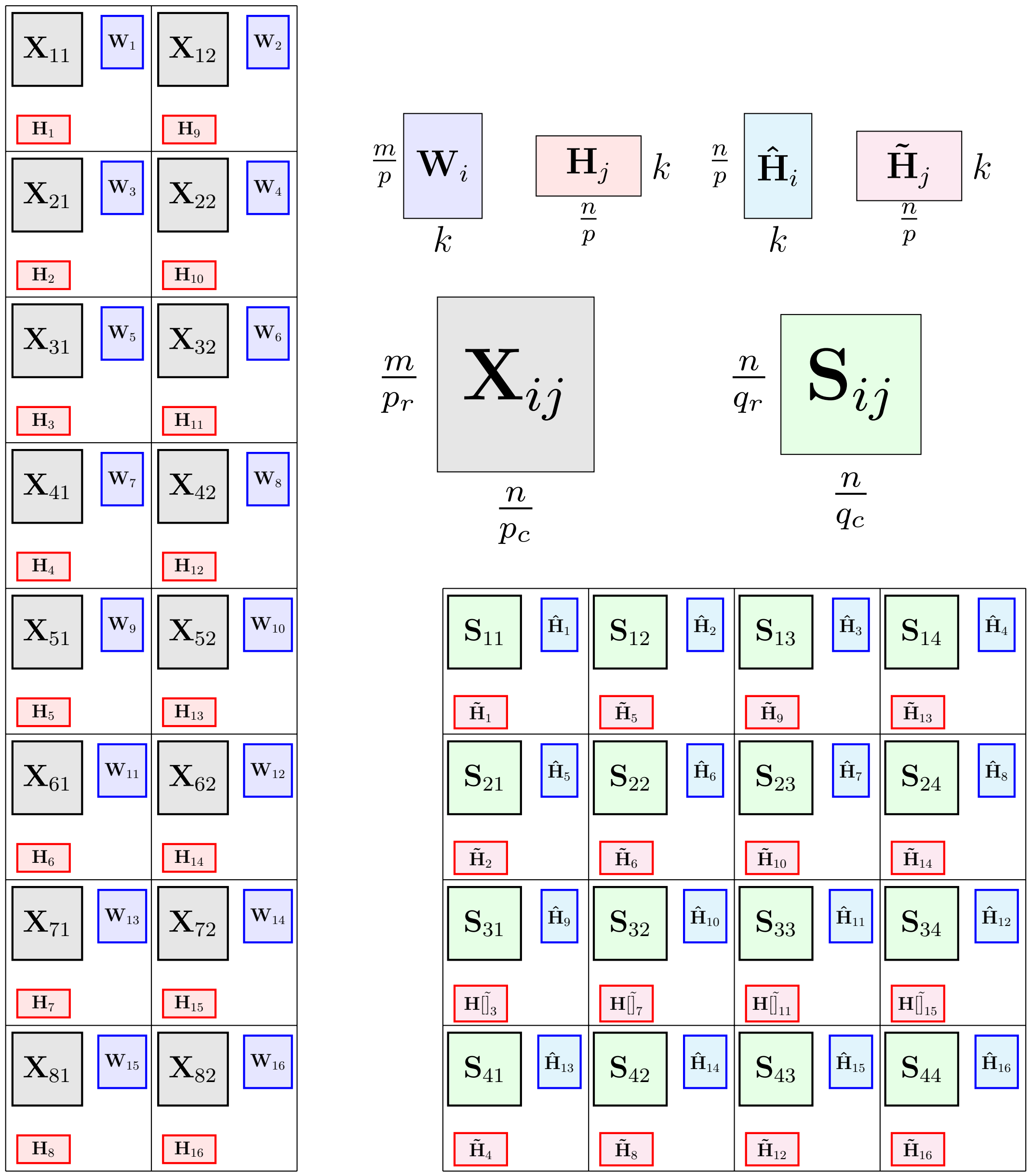

$\mathbf{X}_{11}$ $\mathbf{W}_1$ $\mathbf{H}_1$
$\mathbf{X}_{12}$ $\mathbf{W}_2$ $\mathbf{H}_9$
$\mathbf{X}_{21}$ $\mathbf{W}_3$ $\mathbf{H}_2$
$\mathbf{X}_{22}$ $\mathbf{W}_4$ $\mathbf{H}_{10}$
$\mathbf{X}_{31}$ $\mathbf{W}_5$ $\mathbf{H}_3$
$\mathbf{X}_{32}$ $\mathbf{W}_6$ $\mathbf{H}_{11}$
$\mathbf{X}_{41}$ $\mathbf{W}_7$ $\mathbf{H}_4$
$\mathbf{X}_{42}$ $\mathbf{W}_8$ $\mathbf{H}_{12}$
$\mathbf{X}_{51}$ $\mathbf{W}_9$ $\mathbf{H}_5$
$\mathbf{X}_{52}$ $\mathbf{W}_{10}$ $\mathbf{H}_{13}$
$\mathbf{X}_{61}$ $\mathbf{W}_{11}$ $\mathbf{H}_6$
$\mathbf{X}_{62}$ $\mathbf{W}_{12}$ $\mathbf{H}_{14}$
$\mathbf{X}_{71}$ $\mathbf{W}_{13}$ $\mathbf{H}_7$
$\mathbf{X}_{72}$ $\mathbf{W}_{14}$ $\mathbf{H}_{15}$
$\mathbf{X}_{81}$ $\mathbf{W}_{15}$ $\mathbf{H}_8$
$\mathbf{X}_{82}$ $\mathbf{W}_{16}$ $\mathbf{H}_{16}$
$\frac{m}{p}$ $\mathbf{W}_i$ k
$\mathbf{H}_j$ k $\frac{n}{p}$
$\frac{n}{p}$ $\hat{\mathbf{H}}_i$ k
$\tilde{\mathbf{H}}_j$ k $\frac{n}{p}$
$\frac{m}{p_r}$ $\mathbf{X}_{ij}$ $\frac{n}{p_c}$
$\frac{n}{q_r}$ $\mathbf{S}_{ij}$ $\frac{n}{q_c}$
$\mathbf{S}_{11}$ $\hat{\mathbf{H}}_1$ $\tilde{\mathbf{H}}_1$
$\mathbf{S}_{12}$ $\hat{\mathbf{H}}_2$ $\tilde{\mathbf{H}}_5$
$\mathbf{S}_{13}$ $\hat{\mathbf{H}}_3$ $\tilde{\mathbf{H}}_9$
$\mathbf{S}_{14}$ $\hat{\mathbf{H}}_4$ $\tilde{\mathbf{H}}_{13}$
$\mathbf{S}_{21}$ $\hat{\mathbf{H}}_5$ $\tilde{\mathbf{H}}_2$
$\mathbf{S}_{22}$ $\hat{\mathbf{H}}_6$ $\tilde{\mathbf{H}}_6$
$\mathbf{S}_{23}$ $\hat{\mathbf{H}}_7$ $\tilde{\mathbf{H}}_{10}$
$\mathbf{S}_{24}$ $\hat{\mathbf{H}}_8$ $\tilde{\mathbf{H}}_{14}$
$\mathbf{S}_{31}$ $\hat{\mathbf{H}}_9$ $\tilde{\mathbf{H}}_3$
$\mathbf{S}_{32}$ $\hat{\mathbf{H}}_{10}$ $\tilde{\mathbf{H}}_7$
$\mathbf{S}_{33}$ $\hat{\mathbf{H}}_{11}$ $\tilde{\mathbf{H}}_{11}$
$\mathbf{S}_{34}$ $\hat{\mathbf{H}}_{12}$ $\tilde{\mathbf{H}}_{15}$
$\mathbf{S}_{41}$ $\hat{\mathbf{H}}_{13}$ $\tilde{\mathbf{H}}_4$
$\mathbf{S}_{42}$ $\hat{\mathbf{H}}_{14}$ $\tilde{\mathbf{H}}_8$
$\mathbf{S}_{43}$ $\hat{\mathbf{H}}_{15}$ $\tilde{\mathbf{H}}_{12}$
$\mathbf{S}_{44}$ $\hat{\mathbf{H}}_{16}$ $\tilde{\mathbf{H}}_{16}$

Figure 4.2: PLANC data distribution of the JointNMF matrices with 16 processors arranged in two logical grids. The 8×2 processor grid is used for multiplying with $\mathbf{X}$ and the square 4×4 grid for $\mathbf{S}$. $\tilde{\mathbf{H}}$ is the replicated factor matrix on the second. Care must be taken to synchronise the factor matrices, $\mathbf{H}$ and $\tilde{\mathbf{H}}$, between the two grids.

4.3.6 Swap Communication

Both grid choices introduce extra communication in the algorithms enumerated below.

1. The distribution of a symmetric matrix $\mathbf{S}$ in a rectangular processor grid causes the one-to-one property of pairwise swaps (see Section 3.3) to no longer hold when needing symmetric regularisation.

2. Computing the gradient of $\mathbf{H}$ requires additions with column-order 1D distributed $\mathbf{W}^{\mathsf{T}}\mathbf{X}$ and row-order 1D distributed $\mathbf{HS}$.

3. Synchronising $\mathbf{H}$ between the grids in the double grid case needs extra communication. Since the $\mathbf{H}$ resides on two different logical grids with differing layouts, swap communications are needed to keep them in sync after the update steps.

We shall prove that these "swap" communications are minimal and involve at most three different processors. That is, a single processor will need to send information to at most two other processors and similarly receive messages from at most two others. The sending and receiving processors could be different from each other. We assume that communicating matrices are 1D distributed across the grids in a contiguous load-balanced way, as shown in Figs. 4.1 and 4.2.

Let us work with the case of synchronising $\mathbf{H}$ between two grids of p processors. In this case, we are communicating a $n \times k$ matrix between a $p_r \times p_c$ grid (X grid) and a $q_r \times q_c$ one (S grid). In the general case when $p \nmid n$, each processor in the X grid could have $\left\lfloor \frac{\lfloor \frac{n}{p_c} \rfloor}{p_r} \right\rfloor$, $\left\lfloor \frac{\lceil \frac{n}{p_c} \rceil}{p_r} \right\rfloor$, $\left\lceil \frac{\lfloor \frac{n}{p_c} \rfloor}{p_r} \right\rceil$, or $\left\lceil \frac{\lceil \frac{n}{p_c} \rceil}{p_r} \right\rceil$ columns of $\mathbf{H}$ as its local copy. We make use of the following nested division property of floor and ceiling functions for $a > 0$ and arbitrary x and b:

$$\left\lfloor \frac{\lfloor \frac{x}{b} \rfloor}{a} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \quad \text{and} \quad \left\lceil \frac{\lceil \frac{x}{b} \rceil}{a} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil .$$

Since $p_r, p_c > 0$ and $p_r p_c = p$ we have that the minimum number of columns of $\mathbf{H}$ owned by a processor is $\left\lfloor \frac{n}{p_r p_c} \right\rfloor = \left\lfloor \frac{n}{p} \right\rfloor$ and the maximum is $\left\lceil \frac{n}{p} \right\rceil$. These two values can differ at most by 1. The same argument can be used to show that in the S grid, each processor owns $\left\lceil \frac{n}{p} \right\rceil$ or $\left\lfloor \frac{n}{p} \right\rfloor$ columns of the replicated factor. Now let us assume that a processor has to send its factor matrix to three processors in the second grid. In the minimal case, the sending processor sends one column to the first receiving, $\left\lfloor \frac{n}{p} \right\rfloor$ to the middle processor, and one column to the last. Therefore, the number of columns it possesses is

$$\left\lfloor \frac{n}{p} \right\rfloor + 2 \geq 1 + \left\lceil \frac{n}{p} \right\rceil ,$$

which is a contradiction. Therefore, a processor needs to send only to two other processors. Similar arguments can be derived for the receiving case and for redistributing a row-major distributed matrix to a column-major one. Thus, the extra communication introduced by JointNMF can be limited to just a couple of swap messages.

4.4 Experiments

4.4.1 Experimental Setup

The serial convergence experiments were carried out on a server with two Intel® Xeon® E5-2680 v3 CPUs and 377 GB memory. All of our experiments at scale were carried out on Phoenix, a supercomputer hosted at Georgia Tech [67]. Each of the 852 compute nodes of Phoenix, in the basic node class, contains two 2.7 GHz Intel® Xeon® 6226 12-core processors, one per socket. Every node has a main memory of 192 GB (DDR4-2,933 MHz) and are interconnected via HDR100 Infiniband interconnect.

PLANC uses Armadillo for matrix operations [68]. Armadillo stores dense matrices in column-major order and sparse matrices in the Compressed Sparse Column (CSC) format. We link Armadillo (version 11.2.3) with OpenBLAS (0.3.13) for dense BLAS and LAPACK operations and compile using GNU C++ version 8.3.0. All our scaling operations are conducted in the flat MPI setting.

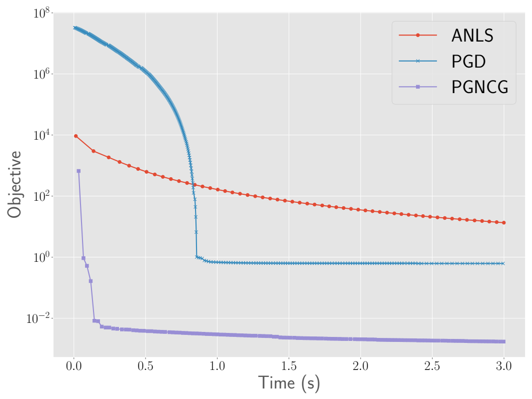
Table 4.1: Convergence studies on the different JointNMF algorithms. The relative objective, time taken, number of function evaluations, and number of knee points are shown. The best performing method is highlighted in bold.

Input	Algorithm	Relative Objective	Time	Function Calls	Knee Point
Dense	ANLS	**0.0002**	35.25 s	**1,000.0**	9.0
	PGD	0.5117	59.90 s	28,659.0	115.8
	PGNCG	0.0009	**23.66 s**	2,004.8	**2.0**
Sparse	ANLS	**0.8589**	55.11 s	**1,000.0**	33.6
	PGD	0.9587	50.74 s	11,947.6	78.0
	PGNCG	0.8597	**35.06 s**	1,895.2	**11.2**
Y04	ANLS	0.9028	135.23 s	**100.0**	**5.0**
	PGD	0.9948	727.35 s	1,716.0	278.2
	PGNCG	**0.8869**	**108.3 s**	198.2	18.6

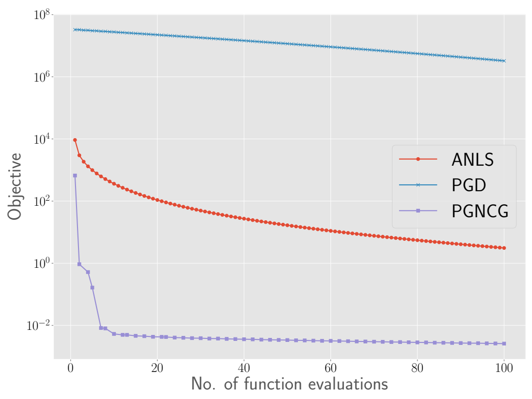
The scaling experiments for this study were conducted on dense synthetic inputs. These matrices are created as nonnegative low-rank matrices by multiplying a randomly generated $\mathbf{W}$ and $\mathbf{H}$. All our timings are averaged over 5 different runs of 10 iterations each. We use $\gamma = 0.9$ for PGD as suggested by Ruder [107]. We use the default ANLS hyperparameters of $\alpha = \|\mathbf{X}\|_F^2 / \|\mathbf{S}\|_F^2$ and $\beta = \alpha \max(\mathbf{S})$ as mentioned by Du et al [72]. For PGNCG we found that setting the inner CG iterations to 20 gave us the best results in terms of minimising the residual.

4.4.2 Convergence

We test the serial performance of our proposed JointNMF algorithms, PGD and PGNCG, on three different datasets and compare against the ANLS version of Du et al. [72]. The dense synthetic input consists of true low rank inputs $\mathbf{X} = \mathbf{WH}$ and $\mathbf{S} = \mathbf{H}^\mathsf{T}\mathbf{H}$ which are perturbed slightly by 1% Gaussian random noise. The dimensions for the dense case were $m = 1{,}000, n = 600$, and $k = 30$. The sparse synthetic case is a uniform random matrix for $\mathbf{X}$ and normal random matrix for $\mathbf{S}$ using Matlab's `sprandsym` function. The negative values in $\mathbf{S}$ have their signs flipped to become nonnegative. Here the dimensions were $m = 1{,}000, n = 600$, and we choose $k = 30$. Both matrices have approximate densities



(a) Convergence versus time.



(b) Convergence versus function evaluations.

Figure 4.3: JointNMF convergence shown for the first 3 seconds and 100 function evaluations of the dense synthetic run. PGNCG displays the fastest drop in objective. ANLS steadily decreases, eventually catching up and surpassing PGNCG around 25 seconds (out of the graph). PGD while initially decreasing the objective reasonably well gets stuck at a poor local minimum. The knee points for ANLS and PGNCG are seen at points 9 and 2 respectively. PGD gets the knee point only at function call 115.

of 0.1. Finally, we test convergence of a real-world patent dataset [72]. Each patent is a document encoded via tf-idf and the citations between the patents are used to generate the connection matrices. We use the Y04 collection of patents which has $\mathbf{X} \in \mathbb{R}_+^{8,142 \times 3,242}$ and $\mathbf{S} \in \mathbb{R}_+^{3,242 \times 3,242}$ with $k = 76$. The densities for $\mathbf{X}$ and $\mathbf{S}$ are 0.0141 and 0.0041, respectively.

We run each algorithm five times with different random seeds. All the methods are initialised with the same starting guesses $\mathbf{W}^{(0)}$ and $\mathbf{H}^{(0)}$. The average results over the runs are shown in Table 4.1. Apart from relative objective and running time, we also capture the number of times the objective is evaluated as well as the "knee" or "elbow" point of the objective versus number of function evaluations curve. Since PGD and PGNCG employ a line search, they can perform more function evaluations than the ANLS method for the same number of outer iterations. ANLS performs one function evaluation per outer iteration. We define the knee as the point where the function is best approximated by a pair of lines[2].

[2]https://www.mathworks.com/matlabcentral/fileexchange/35094-knee-point

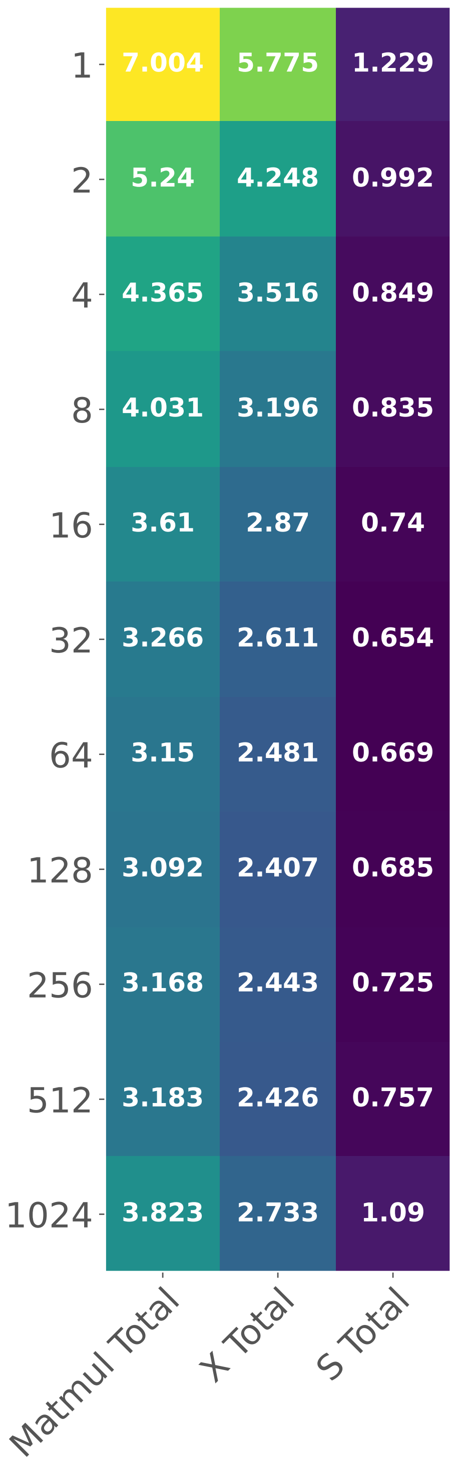
Table 4.1 shows the final relative objective achieved by the three different algorithms. It is evident that PGD converges a lot slower than the other two methods. Taking a look at the rate of convergence for the dense case in Fig. 4.3, we can see that while PGD is able to perform each update quickly it's not able to decrease the objective sufficiently. This slow rate of convergence results in a large number of function calls and later knee points for PGD than ANLS and PGNCG. Perhaps a more aggressive line search method might alleviate this slow convergence of PGD. This observation adds more weight to the claim of using more accurate update methods than simple gradient descent.

It is more difficult to distinguish between the PGNCG and ANLS methods. They show similar performance in terms of relative objective and location of the knee point. A surprising finding is that PGNCG runs approximately 24-57% faster in spite of performing roughly twice as many function evaluations as ANLS. There are two possible explanations for this behaviour. ANLS performs an extra large matrix multiplication per function evaluation ($\hat{\mathbf{H}}\mathbf{S}$) when compared to PGNCG, which could be expensive. The second is that the inexact CG iterations of PGNCG might be running faster than the exact NLS solve employed by the ANLS method. We shall benchmark these regions in the scaling studies since time per function evaluation is the key performance characteristic in the parallel setting.

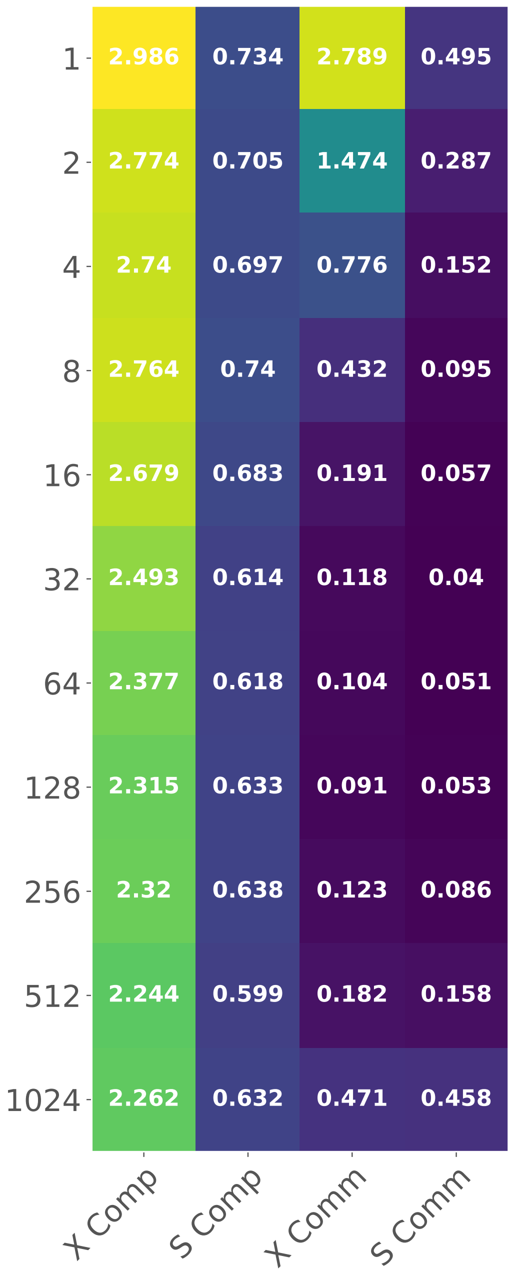
4.4.3 Grid Choice for Parallel Matrix Multiplication

As we have seen in the previous chapters (Sections 2.3.4 and 3.5), matrix multiplication consumes the majority of the time for these methods. Therefore, the choice of grid layout is a crucial one for JointNMF. First, we sweep all possible combinations of grids for a 1,024 MPI processes and benchmark the different matrix multiplication times for the ANLS method with a single grid configuration (Fig. 4.4). The matrix multiplication times are normalised by the number of function evaluations.

We used a dense synthetic input with $m = 1{,}228{,}800$, $n = 307{,}200$, and $k = 50$. Since the aspect ratio for $\mathbf{X}$ is 4 we expect the 64×16 grid to be optimal for multiplications with



(a) Total time heatmap.



(b) Heatmap breakdown for $\mathbf{X}$ and $\mathbf{S}$.

Figure 4.4: Matrix multiplication times (in seconds) for different grid combinations for JointNMF with m = 1,228,800,n = 307,200, and k = 50. The experiment was run on 43 nodes (1,024 cores) and the y-axis shows the p_r value for the grid chosen (with $p_c = \frac{1024}{p_r}$). 128×8 is the best overall grid and the fastest for $\mathbf{X}$ whereas, 32×32 is best for $\mathbf{S}$.

Table 4.2: Running JointNMF with either single or double grid configurations. The best performance is highlighted in bold.

Label	X grid	S grid	Time	Speedup
Empirical best single grid and best $\mathbf{X}$	128×8	128×8	**3.2772 s**	**1.0769**
Theoretical best $\mathbf{X}$	64×16	64×16	3.4684 s	1.0176
Empirical and theoretical best $\mathbf{S}$	32×32	32×32	3.5293 s	1.0000
Empirical best double	128×8	32×32	**3.3612 s**	**1.0500**
Theoretical best double	64×16	32×32	3.5145 s	1.0042

$\mathbf{X}$ and a 32×32 for $\mathbf{S}$. From Fig. 4.4b, we can see the computation times for the different configurations do not vary a lot — 33% for $\mathbf{X}$ and 22% for $\mathbf{S}$ — but the communication times can fluctuate by up to $30\times$ for $\mathbf{X}$ and $12\times$ for $\mathbf{S}$. This is expected since the grid choices primarily affect the communication costs of the matrix multiplications. The best single grid for $\mathbf{X}$ was the 128×8 setting and for $\mathbf{S}$ it was 32×32, coinciding well with theory. The theoretical optimal grid for $\mathbf{X}$ also performs reasonably, being only 3% off the best grid. The single-best grid for both matrix multiplication, by adding the $\mathbf{X}$ and $\mathbf{S}$ times, is the 128×8 case.

In theory, by adding the best possible times from Fig. 4.4a, we should be able to beat the single-grid performance by utilising the double-grid approach. For our experiments, the times from the best possible configurations for $\mathbf{X}$ and $\mathbf{S}$ adds up to 3.061 s. This time is only a paltry 1% improvement from the best single grid (3.092 s). We run the same experiments with following grid settings:

1. The empirically found best single grid overall (128×8).
2. The theoretically best single grid for $\mathbf{X}$ (64×16).
3. The theoretically and empirically found best single grid for $\mathbf{S}$ (32×32).
4. The empirically found best double grid. A 128×8 grid for the $\mathbf{X}$ multiplications and a 32×32 grid for the ones with $\mathbf{S}$.
5. The theoretical best double grid with shape 64×16 for $\mathbf{X}$ and 32×32 grid for $\mathbf{S}$.

The matrix multiplication times per function evaluation for these grids are shown in Table 4.2. The two empirically chosen grids (best single and best double) are the best performing settings. The double-grid configuration is not able to beat the best single-grid setting. However, the times are very similar (all within 8% of each other). It might be simpler to use the settings that are easier to calculate beforehand than performing empirical evaluations.

4.4.4 Strong Scaling

We close out our experiments with some strong-scaling studies as shown in Figs. 4.5 and 4.6. We run on dense synthetic matrices with dimensions $m = 184{,}320$, $n = 46{,}080$, and $k = 50$ from 1 node (12 cores) to 40 nodes (960 cores) of the Phoenix cluster. We use three different grid settings:

1. Best empirical single-grid (ES): We mimic the best single-grid from the preceding section. We approximate the grid aspect ratio of 8 as we scale up. The different grid shapes in this setting are 12×1, 12×2, 18×1, 16×3, 24×4, 32×6, 40×6, 60×6, 60×8, 60×10, 72×10, 84×10, 80×12.

2. Best empirical double-grid (ED): We follow the best double-grid from the preceding case. We approximate the grid aspect ratio of 8 for the X grid and use as close to square grids for S. The grid shapes for X grids are the same as the ES setting. The S grids have the following shapes: 4×3, 6×4, 6×6, 8×6, 12×8, 16×12, 16×15, 20×18, 24×20, 25×24, 30×24, 30×28, 32×30.

3. Best theoretical double-grid (TD): This grid is the theoretically best setting, in which each grid attempts to be as close to the input matrices aspect ratios. The X grids try to obtain the ratio 4 and are 6×2, 8×3, 12×3, 12×4, 16×6, 24×8, 30×8, 36×10, 40×12, 50×12, 48×15, 60×14, 60×16. The S grids are chosen to be as square as possible and have the same shapes as the S grids for ED.

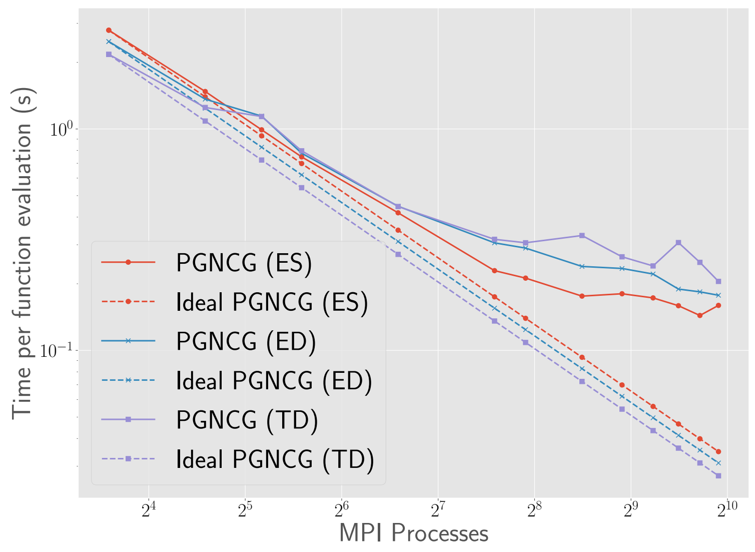
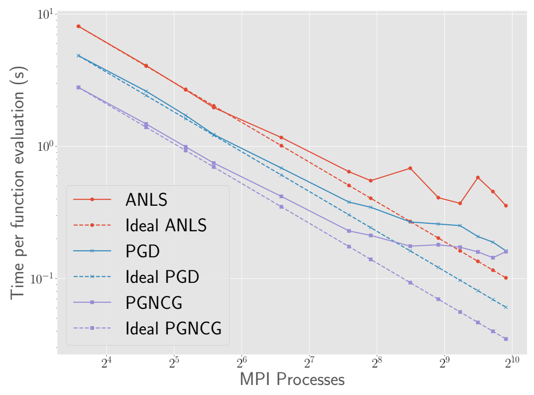


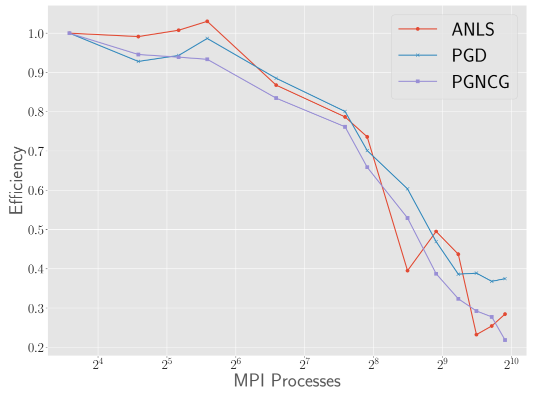
Figure 4.5: Time per function evaluation for the PGNCG algorithm under all three grid settings. The input is a dense synthetic matrix with $m = 184{,}320$, $n = 46{,}080$, and $k = 50$.

Fig. 4.5 shows the effect of the different grid settings on the PGNCG method. They all scale elegantly till 10 nodes (240 cores) before losing efficiency. The ES settings has the best time per function evaluation overall, but the difference is small between the grids at 960 processes: 0.1597 s for ES, 0.1774 s for ED, and 0.2050 s for TD.

The comparison between the different algorithms for the ES grid is shown in Fig. 4.6. Similar to the previous case, we see good scaling till 240 MPI processes (10 nodes) before performance degrades. Interestingly, PGNCG is roughly $2.9\times$ faster than ANLS at the single-socket level and maintains that advantage till 960 processes at $2.2\times$. We also break down the computation for both of these methods in Fig. 4.7. Both the NLS and matrix multiplication components are larger for ANLS, due to extra multiplication performed by the ANLS method and the relatively cheaper update algorithm in PGNCG. Both these components scale linearly with p and we should expect the difference to become smaller at larger scales. Similar results are seen for the ED and TD grid settings.

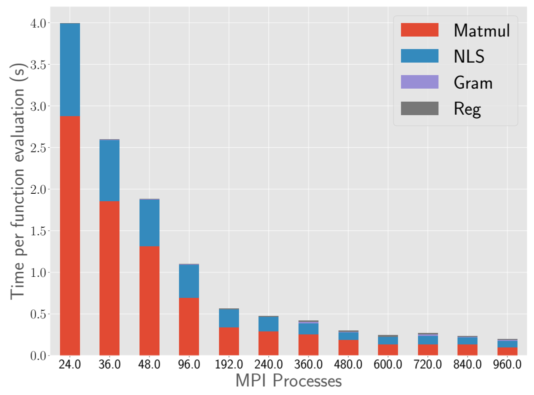


(a) Time per function evaluation.

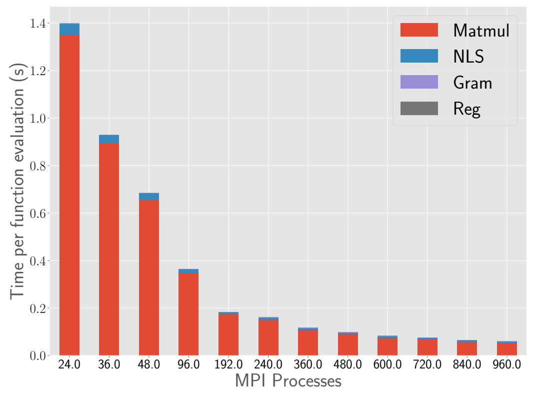


(b) Efficiency.

Figure 4.6: Strong scaling for dense synthetic matrices with $m = 184{,}320$, $n = 46{,}080$, and $k = 50$ run in the single grid setting (ES). The grids are selected with $p_r/p_c \approx 8$.



(a) ANLS breakdown.



(b) PGNCG breakdown.

Figure 4.7: Strong scaling computation breakdown for dense synthetic matrices with $m = 184{,}320$, $n = 46{,}080$, and $k = 50$ run in the single grid setting (ES). The grids are selected with $p_r/p_c \approx 8$. The extra matrix multiplication and NLS times account for the extra time per function evaluation taken by ANLS.

4.5 Summary

We extend our CLRA toolkit to handle multimodal inputs by including JointNMF in the suite of tools available in PLANC. With this inclusion, we are able to tackle two of the most common forms of data encountered: feature-data relationships and data-data relationships. We show that our current formulation with two inputs can easily be modified to handle arbitrary inputs in either mode.

Two primary challenges were tackled during the development of parallel algorithms for JointNMF: handling matrices with different aspect ratios and extending SymNMF to general rectangular grids in a communication-efficient manner. We resolved both these issues in the same manner by incurring only a limited number of point-to-point messages. We demonstrated the different grid choices theoretically and conducted experiments to verify them. We developed new second-order algorithms for JointNMF using a similar strategy as in the SymNMF case. In contrast to the earlier GNCG algorithm our current PGNCG version comes with guarantees of convergence. Both PGD and PGNCG run faster than ANLS with respect to the number of function evaluations. However, as seen in the preceding sections PGNCG does not have as stark an advantage over ANLS as in the SymNMF case since it performs more function evaluations. All three methods perform similarly in terms of scaling.

CHAPTER 5
EXTENDING TO MULTILINEAR APPROXIMATIONS

The PLANC software package is also capable of handling *tensors*, which are higher order extensions of matrices. Tensors have recently become popular in data mining community due to their natural ability to capture multi-way data dependencies. These datasets are naturally indexed by three or more indices. An example from the previous chapters would be an attributed graph varying over time. Every edge in this object is indexed by a source, destination, and time stamp. Tensors have many applications in clustering, dimensionality reduction, latent factor models, and many others.

The *Matricised-Tensor Times Khatri-Rao Product* (MTTKRP) kernel is the key computation for tensor CLRA algorithms similar to the matrix multiplication in the matrix case. PLANC implements a distributed-memory parallel MTTKRP which allows us to scale to massive datasets. In this chapter we describe our parallel algorithms, software package, and report efficiency and scalability results for both synthetic and real-world data.

5.1 Introduction

The focus of this chapter is the approximation of nonnegative matrices, and its generalization nonnegative tensors, into nonnegative factors. We are already familiar with NMF from previous chapters but we restate it here for completeness.

$$\min_{\mathbf{W}\geq 0, \mathbf{H}\geq 0} \left\| \mathbf{X} - \sum_{r=1}^{k} \mathbf{W}(:,r)\mathbf{H}(:,r)^{\mathsf{T}} \right\|_F^2 \tag{5.1}$$

Here $\mathbf{X} \in \mathbb{R}_+^{m\times n}$ is a nonnegative input matrix and k is a fixed constant, which is the *rank* of the approximation desired. This definition can be generalised to multidimensional arrays, also known as tensors, and the related problem known as Nonnegative Tensor Factorisation

(NTF) or Nonnegative CANDECOMP/PARAFAC (NCP)[1]. Formally, NCP can be defined as

$$\min_{\mathbf{H}^{(i)} \geq 0} \left\| \mathcal{X} - \sum_{r=1}^{k} \mathbf{H}^{(1)}(:,r) \circ \cdots \circ \mathbf{H}^{(N)}(:,r) \right\|_F^2 \tag{5.2}$$

for a fixed rank k, where $\mathbf{H}^{(1)}(:,i) \circ \cdots \circ \mathbf{H}^{(N)}(:,i)$ is the outer product of the i^{th} vector from all the N factors. This outer product yields a rank-one tensor and $\sum_{r=1}^{k} \mathbf{H}^{(1)}(:,r) \circ \cdots \circ \mathbf{H}^{(N)}(:,r)$ results in a sum of k rank-one tensors that approximate the N^{th} order nonnegative input tensor $\mathcal{X}$.

In developing an efficient parallel algorithm for computing NCP of a dense tensor, the key is to parallelise the bottleneck computation known as MTTKRP [108]. A different result is needed for each mode of the tensor, and the MTTKRP for mode n for $1 \leq n \leq N$, is defined as

$$\mathbf{M}^{(n)} = \mathbf{X}_{(n)} \left(\mathbf{H}^{(N)} \odot \cdots \odot \mathbf{H}^{(n+1)} \odot \mathbf{H}^{(n-1)} \odot \cdots \odot \mathbf{H}^{(1)} \right) \tag{5.3}$$

where $\mathbf{X}_{(n)}$ is a matricisation or flattening of the tensor with respect to mode n and $\odot$ is the Khatri-Rao product or columnwise-wise Kronecker product [108, 109]. For the matrix case this corresponds to the computations $\mathbf{X}^{\mathsf{T}}\mathbf{W}$ and $\mathbf{XH}$, which have highly optimised single node implementations [16, 18] and can be parallelised efficiently [17, 110]. The kernels for NCP can be cast as matrix computations, but the complicated layout of tensors in memory prevents the straightforward use of BLAS and LAPACK and parallel matrix algorithms.

PLANC[2] has been developed in stages with the initial dense NMF algorithm [35], followed by sparse NMF [96], extension to tensors [111, 112], addition of GPU acceleration and new NLS solvers [36, 37], to hierarchical decompositions [113], and finally to multi-

[1]We use the terms NCP and NTF interchangeably in this chapter.

[2]The software is available at https://github.com/ramkikannan/planc

modal inputs [38]. For the rest of this chapter, we shall present the parallelisation strategies and experimental results for the more general NCP algorithms.

5.2 Preliminaries

5.2.1 Notation

Tensors will be denoted using Euler script (e.g., $\mathcal{T}$), matrices will be denoted with uppercase boldface (e.g., $\mathbf{M}$), vectors will be denoted with lowercase boldface (e.g., $\mathbf{v}$), and scalars will not be boldface (e.g., s). We use Matlab style notation to index into tensors, matrices, and vectors, and we use 1-indexing. For example, $\mathbf{M}(:, c)$ gives the c^{th} column of the matrix $\mathbf{M}$.

We use $\circ$ to denote the outer product of two or more vectors. The Hadamard product is the element-wise matrix product and will be denoted using $*$. The Khatri-Rao Product (KRP) will be denoted with $\odot$. Given matrices $\mathbf{A}$ and $\mathbf{B}$ that are $I_A \times R$ and $I_B \times R$, the KRP $\mathbf{K} = \mathbf{A} \odot \mathbf{B}$ is $I_A I_B \times R$. It can be thought of as a row-wise Hadamard product, where $\mathbf{K}(i + I_A(j-1), :) = \mathbf{A}(i, :) * \mathbf{B}(j, :)$, or a column-wise Kronecker product, where $\mathbf{K}(:, c) = \mathbf{A}(:, c) \otimes \mathbf{B}(:, c)$.

The CP decomposition of a tensor is a LRA of a tensor, where the approximation is a sum of rank-one tensors and each rank-one tensor is the outer product of vectors. We use the notation,

$$\mathcal{X} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \sum_{r=1}^{k} \mathbf{H}^{(1)}(:, r) \circ \cdots \circ \mathbf{H}^{(N)}(:, r) \ ,$$

to represent a rank-k LRA model, where $\mathbf{H}^{(n)}$ is called a factor matrix and collects the mode-n vectors of the rank-one tensors as columns. The columns of the factor matrices are often normalised, with weights collected into a vector $\boldsymbol{\lambda}$ of length k; in this case we use the notation $[\![\boldsymbol{\lambda}; \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$. Unlike the matrix case, all factor matrices have low-rank (k)

number of columns. This convention is a change from the NMF case, where $\mathbf{H} \in \mathbb{R}_+^{k \times n}$, as we seek to find embeddings for all modes and not just for the data item mode.

An NCP constrains the factor matrices to have nonnegative values. In this work, we are interested in NCP models that are good approximations to $\mathcal{X}$ in the least-squares sense. That is, we seek

$$\min_{\mathbf{H}^{(i)} \geq 0} \left\| \mathcal{X} - [\![\boldsymbol{\lambda}; \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] \right\|, \tag{5.4}$$

where the tensor norm is a generalisation of the matrix Frobenius or vector 2-norm, the square root of the sum of squares of the entries. Here $\mathcal{X}$ is a tensor with dimensions $I_1 \times ... \times I_N$. We denote the product of its dimensions by I, that is $I = \prod I_n$.

The n^{th} mode matricised tensor denoted by $\mathbf{X}_{(n)}$ is a $I_n \times I/I_n$ matrix formed by organising the n^{th} mode fibers of $\mathcal{X}$ into the columns of a matrix. The MTTKRP will be central to this work and takes the form $\mathbf{M}^{(n)} = \mathbf{X}_{(n)}\mathbf{K}^{(n)}$, where $\mathbf{K}^{(n)} = \mathbf{H}^{(N)} \odot \cdots \odot \mathbf{H}^{(n+1)} \odot \mathbf{H}^{(n-1)} \odot \cdots \odot \mathbf{H}^{(1)}$.

5.2.2 Nonnegative CP and Alternating-Updating Methods

The CP decomposition is a low-rank approximation of a multi-dimensional array, or tensor, which generalises matrix approximations like the truncated singular value decomposition. As in Fig. 5.1, CP decomposition approximates the given input matrix as sum of k rank-one tensors.

Algorithm 8 shows the pseudocode for an alternating-updating algorithm applied to NCP [27]. Line 11, Line 12, and Line 14 compute matrices involved in the gradients of the subproblem objective functions, and Line 13 uses those matrices to update the current factor matrix. The NLS-Update in Line 13 can be implemented in different ways (see Section 2.2.2). In a faithful BCD algorithm, the subproblems are solved exactly; in this case, the subproblem is a nonnegative linear least-squares problem, which is convex.

However, as discussed in Section 2.2.2 for the matrix case, there are other reasonable alternatives to updating the factor matrix without solving the N-block coordinate descent

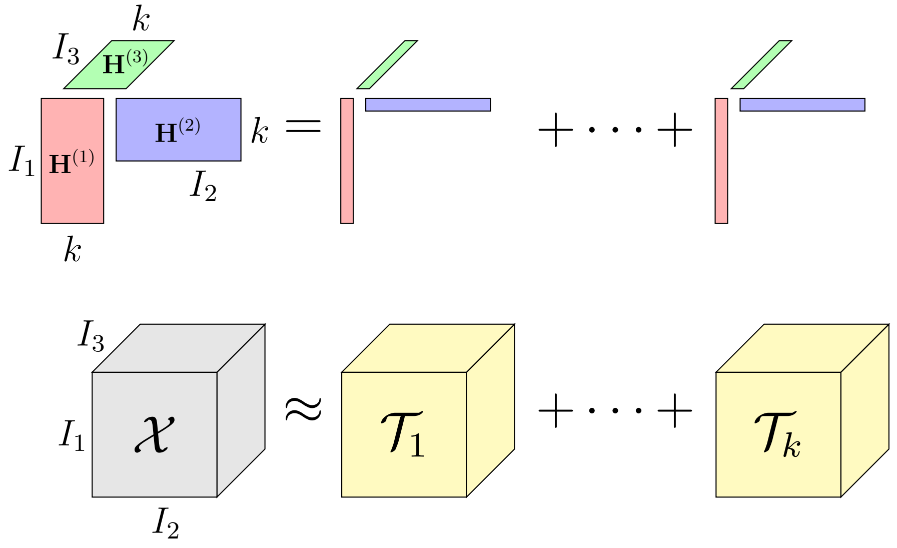


Figure 5.1: Visualisation of the CP decomposition of the input tensor $\mathfrak{X}$. The input is approximated as a sum of k rank-one tensors $\sum_{r=1}^{k} \mathfrak{T}_r$. Each rank-one tensor is an outer product of vectors which are gathered into the columns of the factor matrices $\mathbf{H}^{(1)}, \mathbf{H}^{(2)}$, and $\mathbf{H}^{(3)}$.

Algorithm 8 $[\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \text{NCP}(\mathfrak{X}, k)$

```
Require: $\mathfrak{X}$ is $I_1 \times \cdots \times I_N$ tensor, $k$ is approximation rank
 1: % Initialise data
 2: for $n = 2$ to $N$ do
 3:     Initialise $\mathbf{H}^{(n)}$
 4:     $\mathbf{G}^{(n)} = \mathbf{H}^{(n)\mathsf{T}}\mathbf{H}^{(n)}$
 5: end for
 6: % Compute NCP approximation
 7: while stopping criteria not satisfied do
 8:     % Perform outer iteration
 9:     for $n = 1$ to $N$ do
10:         % Compute new factor matrix in nth mode
11:         $\mathbf{M}^{(n)} = \text{MTTKRP}(\mathfrak{X}, \{\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(n-1)}, \mathbf{H}^{(n+1)}, \ldots, \mathbf{H}^{(N)}\}, n)$
12:         $\mathbf{S}^{(n)} = \mathbf{G}^{(1)} * \cdots * \mathbf{G}^{(n-1)} * \mathbf{G}^{(n+1)} * \cdots * \mathbf{G}^{(N)}$
13:         $\mathbf{H}^{(n)} = \text{NLS-Update}(\mathbf{S}^{(n)}, \mathbf{M}^{(n)})$
14:         $\mathbf{G}^{(n)} = \mathbf{H}^{(n)\mathsf{T}}\mathbf{H}^{(n)}$
15:     end for
16: end while
Ensure: $\mathfrak{X} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$
```

subproblem exactly. The parallel algorithm presented in this chapter is generally agnostic to the approach used to solve the nonnegative least-squares subproblems, as all these methods are bottlenecked by the subroutine they have in common, the MTTKRP.

5.3 Related Work

The formulation of NCP with least-squares error and algorithms for computing it go back as far as the 1990s [114, 115], developed in part as a generalisation of nonnegative matrix factorisation algorithms [14] to tensors. Sidiropoulos et al. provide a more detailed and complete survey that includes basic tensor factorisation models with and without constraints, broad coverage of algorithms, and recent driving applications [116]. The mathematical tensor operations discussed and the notation used in this chapter follow Kolda and Bader's survey [109].

Recently, there has been growing interest in scaling tensor operations to bigger data and more processors in both the data mining/machine learning and the high-performance computing communities. For sparse tensors, there have been parallelisation efforts to compute CP decompositions on shared-memory platforms [117, 118], distributed-memory platforms [119–121], and GPUs [122–124], and these approaches can be generalised to constrained problems [125].

Liavas et al. [126] extend a parallel algorithm designed for sparse tensors [120] to the 3D dense case. They use the "medium-grained" dense tensor distribution and row-wise factor matrix distribution, which is exactly the same as our distribution strategy (see Section 5.4.1), and they use a Nesterov-based algorithm to enforce the nonnegativity constraints. A similar data distribution and parallel algorithm for computing a single dense MTTKRP computation is proposed by Ballard, Knight, and Rouse [127]. Another approach to parallelising NCP decomposition of dense tensors is presented by Phan and Cichocki, but they use a dynamic tensor factorisation, which performs different, more independent computations across processors [128]. Moon et al. address the data locality optimisations needed during the NLS phase of the algorithm for both shared memory and GPU systems [123]. Ma and Solomonik [129] and Singh et al. [83] compute unconstrained CP decompositions using the Cyclops Tensor Framework [130] as a backend for parallel dense tensor contractions. The former uses a pairwise perturbation technique to approximate MTTKRP computations within Alternating Least-Squares, and the latter applies a direct optimisation technique based on Gauss-Newton.

The idea of using dimension trees (discussed in Section 5.4.2) to avoid recomputation within MTTKRPs across modes is introduced by Phan et al. for computing the CP decomposition of dense tensors [131]. General reuse patterns and mode splitting were present in earlier works on variants of the Tucker Decomposition [132, 133]. It has also been used for sparse CP [118, 121] and sparse Tucker [119].

An alternate approach to speeding up CP computations is by reducing the tensor size either via sampling or compression. A large body of work exists for randomised tensor methods [134–137] which are recently being extended to the constrained problem [138, 139]. The approach to reduce the tensor size is to first compress the tensor using a different decomposition, like Tucker, and then compute CP on this reduced array. This method has been discussed in further detail by Bro and De Jong [140] and Thomasi and Bro [141], but it becomes more difficult to impose nonnegative constraints on the overall model. A separate approach is to compute the (constrained) CP decomposition of the entire approximation, rather than only the core tensor, exploiting the structure of the Tucker model to perform the optimisation algorithm more efficiently [142].

PLANC presents the first distributed-memory implementation of NCP for dense tensors of arbitrary orders. It utilises the data layouts of Ballard et al., which is shown to be communication-optimal for MTTKRP [127, 143]. Dimension trees are also employed to reuse computations across multiple MTTKRPs [131].

5.4 Algorithms

5.4.1 Parallel NCP Algorithm

Algorithm Overview

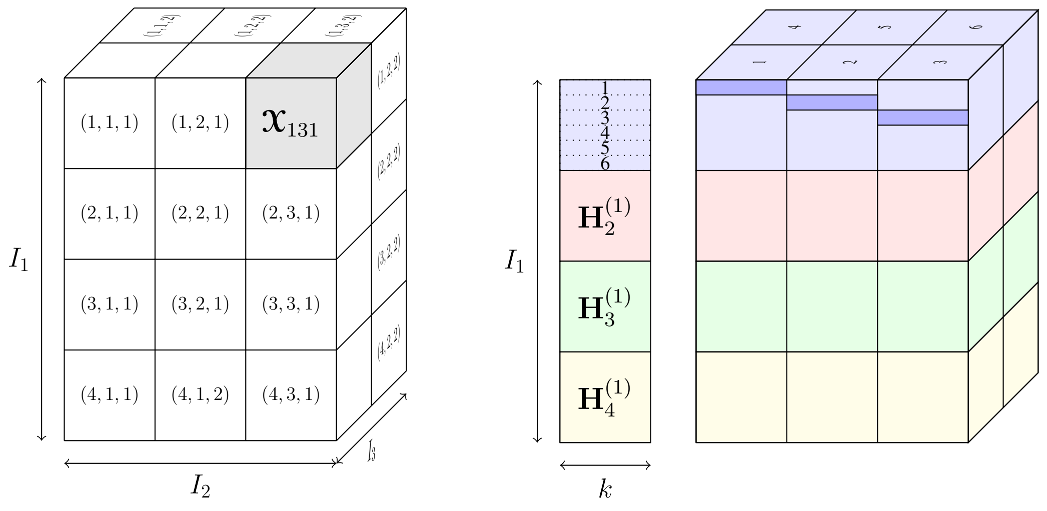
The basic sequential algorithm is given in Algorithm 8, and the parallel version is given in Algorithm 9. In Algorithm 9, we will refer to both the inner iteration, in which one factor matrix is updated (Line 10 to Line 20), and the outer iteration, in which all factor matrices are updated (Line 8 to Line 21). In the parallel algorithm, the processors are organised into a logical multidimensional grid (tensor) with as many modes as the data tensor. The communication patterns used in the algorithm are MPI collectives: All-Reduce, Reduce-Scatter, and All-Gather. The processor communicators (across which the collectives are performed) include the set of all processors and the sets of processors within the same processor slice. Processors within a mode-n slice all have the same n^{th} coordinate. Each processor is part of N different PROC-SLICE communicators, which we denote by PROC-SLICE(n, p_n), where n refers to the mode and p_n refers to the n^{th} processor coordinate (i.e., the p_n-th processor slice in mode n).

The method of enforcing the nonnegativity constraints of the linear least-squares solve or update generally affects only local computation because each row of a factor matrix can be updated independently. In our algorithm, each processor solves the linear problem or computes the update for its subset of rows (see Line 15). The most expensive (and most complicated) part of the parallel algorithm is the computation of the MTTKRP, which corresponds to Line 12, Line 13, and Line 19.

The details that are omitted from this presentation of the algorithm include the normalisation of each factor matrix after it is computed and the computation of the residual error at the end of an outer iteration. These two computations do involve both local computation and communication, but their costs are negligible.

Algorithm 9 $[\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \text{Par-NCP}(\boldsymbol{\mathcal{X}}, R)$

```
Require: $\boldsymbol{\mathcal{X}}$ is an $I_1 \times \cdots \times I_N$ tensor distributed across a $P_1 \times \cdots \times P_N$ grid of $P$ processors,
    so that $\boldsymbol{\mathcal{X}}_{\mathbf{p}}$ is $(I_1/P_1) \times \cdots \times (I_N/P_N)$ and is owned by processor $\mathbf{p} = (p_1, \ldots, p_N)$,
    $k$ is rank of approximation
 1: for $n = 2$ to $N$ do
 2:     Initialise $\mathbf{H}_{\mathbf{p}}^{(n)}$ of dimensions $(I_n/P) \times k$
 3:     $\overline{\mathbf{G}}$ = Local-SYRK($\mathbf{H}_{\mathbf{p}}^{(n)}$)
 4:     $\mathbf{G}^{(n)}$ = All-Reduce($\overline{\mathbf{G}}$, ALL-PROCS)
 5:     $\mathbf{H}_{p_n}^{(n)}$ = All-Gather($\mathbf{H}_{\mathbf{p}}^{(n)}$, PROC-SLICE$(n, p_n)$)
 6: end for
 7: % Compute NCP approximation
 8: while not converged do
 9:     % Perform outer iteration
10:     for $n = 1$ to $N$ do
11:         % Compute new factor matrix in $n$th mode
12:         $\overline{\mathbf{M}}$ = Local-MTTKRP($\boldsymbol{\mathcal{X}}_{\mathbf{p}}$, $\{\mathbf{H}_{p_1}^{(1)}, \ldots, \mathbf{H}_{p_{n-1}}^{(n-1)}, \mathbf{H}_{p_{n+1}}^{(n+1)}, \ldots, \mathbf{H}_{p_N}^{(N)}\}$, $n$)
13:         $\mathbf{M}_{\mathbf{p}}^{(n)}$ = Reduce-Scatter($\overline{\mathbf{M}}$, PROC-SLICE$(n, p_n)$)
14:         $\mathbf{S}^{(n)} = \mathbf{G}^{(1)} * \cdots * \mathbf{G}^{(n-1)} * \mathbf{G}^{(n+1)} * \cdots * \mathbf{G}^{(N)}$
15:         $\mathbf{H}_{\mathbf{p}}^{(n)}$ = NLS-Update($\mathbf{S}^{(n)}$, $\mathbf{M}_{\mathbf{p}}^{(n)}$)
16:                                                         ▷ Organise data for later modes
17:         $\overline{\mathbf{G}} = {\mathbf{H}_{\mathbf{p}}^{(n)}}^{\mathsf{T}} \mathbf{H}_{\mathbf{p}}^{(n)}$
18:         $\mathbf{G}^{(n)}$ = All-Reduce($\overline{\mathbf{G}}$, ALL-PROCS)
19:         $\mathbf{H}_{p_n}^{(n)}$ = All-Gather($\mathbf{H}_{\mathbf{p}}^{(n)}$, PROC-SLICE$(n, p_n)$)
20:     end for
21: end while
Ensure: $\boldsymbol{\mathcal{X}} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$
Ensure: Local matrices: $\mathbf{H}_{\mathbf{p}}^{(n)}$ is $(I_n/P) \times k$ and owned by processor $\mathbf{p} = (p_1, \ldots, p_N)$,
    for $1 \leq n \leq N$, $\boldsymbol{\lambda}$ stored redundantly on every processor
```



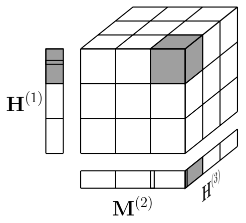
(a) Tensor distribution. (b) Factor matrix distribution.

Figure 5.2: PLANC data distribution for an $I_1 \times I_2 \times I_3$ tensor $\mathcal{X}$ on 24 processors arranged as a $4 \times 3 \times 2$ grid. The local tensor $\mathcal{X}_{131}$ for processor $(1,3,1)$ with dimensions $\frac{I_1}{4} \times \frac{I_2}{3} \times \frac{I_3}{2}$ is highlighted. $\mathbf{H}^{(1)}$ is distributed as block rows with each block being replicated along each mode-1 slice. The local parts of $\mathbf{H}_1^{(1)}$ is shown as dark blue rectangles along PROC-SLICE$(1,1)$ with labels $1,\ldots,6$ in the natural mode-descending order.

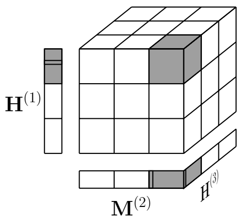
Data Distribution

Given a logical processor grid of processors $P_1 \times \cdots \times P_N$, we distribute the size $I_1 \times \cdots \times I_N$ tensor $\mathcal{X}$ in a block or Cartesian partition. Each processor owns a local tensor of dimensions $(I_1/P_1) \times \cdots \times (I_N/P_N)$, and only one copy of the tensor is stored. Locally, the tensor is stored linearly, with entries ordered in a natural mode-descending way that generalises column-major layout of matrices. Given a processor $\mathbf{p} = (p_1, \ldots, p_N)$, we denote its local tensor $\mathcal{X}_{\mathbf{p}}$.

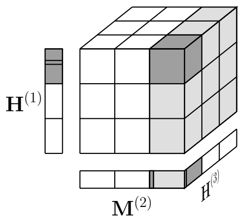
Each factor matrix is distributed across processors in a block row partition, so that each processor owns a subset of the rows. We use the notation $\mathbf{H}_{\mathbf{p}}^{(n)}$, which has dimensions $I_n/P \times k$, to denote the local part of the nth factor matrix stored on processor $\mathbf{p}$. However, we also make use of a redundant distribution of the factor matrices across processors, because all processors in a mode-n processor slice need access to the same entries of $\mathbf{H}^{(n)}$ to perform their computations. The notation $\mathbf{H}_{p_n}^{(n)}$ denotes the $I_n/P_n \times k$ submatrix of $\mathbf{H}^{(n)}$



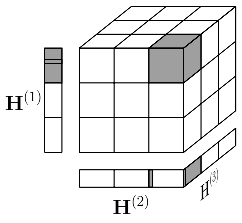
(a) Start n^{th} iteration with redundant subset of rows of each input matrix.



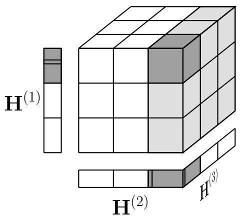
(b) Compute local MTTKRP for contribution to output matrix $\mathbf{M}^{(2)}$.



(c) Reduce-Scatter to compute and distribute rows of $\mathbf{M}^{(2)}$.



(d) Compute NLS update to obtain $\mathbf{H}^{(2)}_{\mathbf{p}}$ from $\mathbf{M}^{(2)}_{\mathbf{p}}$ (and $\mathbf{S}^{(2)}$).



(e) All-Gather to collect rows of $\mathbf{H}^{(2)}$ needed for later inner iterations.

Figure 5.3: Illustration of 2nd inner iteration of Par-NCP algorithm for 3-way tensor on a $3 \times 3 \times 3$ processor grid, showing data distribution, communication, and computation across steps. Highlighted areas correspond to processor $(1, 3, 1)$ and its processor slice with which it communicates. The column normalisation and computation of $\mathbf{G}^{(2)}$, which involve communication across all processors, is not shown here.

that is redundantly stored on all processors whose n^{th} coordinate is p_n (there are P/P_n such processors). Figure 5.2 shows a 3-dimensional example.

Other matrices involved in the algorithm include $\mathbf{M}^{(n)}_{\mathbf{p}}$, which is the result of the MTTKRP computation and has the same distribution scheme as $\mathbf{H}^{(n)}_{\mathbf{p}}$, and $\mathbf{G}^{(n)}$, which is the $k \times k$ Gram matrix of the factor matrix $\mathbf{H}^{(n)}$ and is stored redundantly on all processors.

Inner Iteration

The inner iteration is displayed graphically in Fig. 5.3 for a 3-way example and an update of the 2nd factor matrix. The main idea is that at the start of the n^{th} inner iteration (Fig. 5.3a), all of the data is in place for each processor to perform a local MTTKRP computation, which can be computed using a dimension tree as described in Section 5.4.2. This means that all processors in a slice redundantly own the same rows of the corresponding factor matrix (for all modes except n). After the local MTTKRP is computed (Fig. 5.3b), each processor has computed a contribution to a subset of the rows of the global MTTKRP $\mathbf{M}^{(n)}$, but its contribution must be summed up with the contributions of all other processors in its mode-n slice (denoted $\text{PROC-SLICE}(n, p_n)$ in Line 13 of Algorithm 9). This summation is performed with a Reduce-Scatter collective across the mode-n processor slice that achieves a row-wise partition of the result (in Fig. 5.3c, the light gray shading corresponds to the rows of $\mathbf{M}^{(2)}$ to which processor $(1, 3, 1)$ contributes and the dark gray shading corresponds to the rows it receives as output). The output distribution of the Reduce-Scatter is designed so that afterwards, the update of the factor matrix in that mode can be performed row-wise in parallel. $\mathbf{S}^{(n)}$ can be computed locally since the Gram matrices, $\mathbf{G}^{(n)}$, are stored redundantly on all processors. Along with $\mathbf{S}^{(n)}$ each processor updates its own rows of the factor matrix given its rows of the MTTKRP result (Fig. 5.3d). The remainder of the inner iteration is preparing and distributing the new factor matrix data for future inner iterations, which includes an All-Gather of the newly computed factor matrix $\mathbf{H}^{(n)}$ across mode-n processor slices (Fig. 5.3e) and recomputing $\mathbf{G}^{(n)} = \mathbf{H}^{(n)\mathsf{T}}\mathbf{H}^{(n)}$.

Analysis

We will analyse the cost of a single outer iteration. While the number of outer iterations is sensitive to the NLS method used, the outer-iteration time is generally the same across methods. We summarise the analysis in Table 5.1, which provides the dominant computation, communication, and memory costs of a single outer-iteration: computing $\mathbf{S}^{(n)}$ and $\mathbf{M}^{(n)}$ for each n that is common to all NLS algorithms.

Computation: The local computation occurs at Lines 12, 14, 15 and 17. The cost of Line 14 is $O(Nk^2)$, the cost of Line 15 is $O(k^3 I_n/P)$, which is a loose upper bound for most methods (see Section 2.2.2), and the cost of Line 17 is $O(k(I_n/P)^2)$. The sum

Table 5.1: Costs per outer iteration (while loop of Algorithm 9) and per processor in terms of computation (flops), communication (words moved), and memory (words) required to compute $\mathbf{S}^{(n)}$ and $\mathbf{M}^{(n)}$ for each n, assuming the local MTTKRP uses a dimension tree [127]. These costs do not include the computation (and possibly communication) costs of the particular NLS algorithm.

Computation	**Communication**	**Temporary Memory**
$O\left(\frac{k}{P}\prod_n I_n + \frac{k^2}{P}\sum_n I_n\right)$	$O\left(k\sum_n \frac{I_n}{P_n}\right)$	$O\left(k\left(\prod_n \frac{I_n}{P_n}\right)^{1/2} + k\sum_n \frac{I_n}{P_n}\right)$

of these three costs across all inner iterations is $O\left(k^2N^2 + (k^3/P)\sum I_n + (k/P^2)\sum I_n^2\right)$, which is dominated by the cost of the MTTKRP. We compute the cost to perform the MTTKRPs using dimension trees amortised over all inner iterations. The dimension tree optimization is explained in detail in Section 5.4.2. The outer iteration cost is dominated by the two partial MTTKRP computations (from the root of the tree), which together are $O((k/P)\prod I_n) = O(Ik/P)$ and dominate the costs of the multi-TTVs. This cost involves the product of all the tensor dimensions, which is why it dominates, and scales linearly with P.

Communication: The communication within the inner iteration occurs at Lines 13, 18 and 19. Line 18 involves $O(k^2)$ data and a collective across all processors. Lines 13 and 19 involve $O(I_nk/P_n)$ data across a subset of P/P_n processors. Thus, the All-Reduce dominates the latency cost and the Reduce-Scatter/All-Gather dominate the bandwidth cost. Using efficient algorithms for the collectives, the total outer-iteration communication cost is $O(k\sum I_n/P_n)$ words and $O(N\log P)$ messages.

If P is large enough, then the bandwidth cost can achieve a value of $O(NkI^{1/N}/P^{1/N})$ by making the local tensors as cubical (all local tensor dimensions are roughly the same), which is communication-optimal [127]. If P is not large enough (or if the tensor dimensions are too skewed) to obtain perfectly cubical tensors, then choosing the processor grid so that local tensors are as cubical as possible is also communication-optimal [143] (in this case some of the processor grid dimensions will be 1). We note that the cost in the case of large P scales with $P^{1/N}$, which is far from linear scaling. However, it is proportional to the geometric mean of the tensor dimensions (on the order of one tensor dimension), which is much less than the computation cost dependence on the product of all dimensions. We report the cost for an arbitrary processor grid in Table 5.1, because the simplified cost expression is not always achievable. In any case of input tensor and number of processors, optimising the processor grid within our framework is sufficient to obtain the communication lower bounds for MTTKRP to within a constant factor [127, 143].

Memory: The algorithm requires extra local memory to run. Aside from the memory required to store the local tensor of $O(I/P)$ words and factor matrices of cumulative size $O((k/P)\sum I_n)$, each processor must be able to store a redundant subset of the rows of the factor matrices it needs to perform MTTKRP computations. This corresponds to storing P/P_n redundant copies of every factor matrix, which results in a local memory requirement of $O(k\sum I_n/P_n)$ for a general processor grid. The processor grid that minimises communication also minimises local memory, and the extra memory requirement can be as low as $O(NkI^{1/N}/P^{1/N})$, which is typically dominated by $O(I/P)$.

The dimension-tree algorithm also requires extra temporary memory space, but the space required tends to be much smaller than what is required to store the local tensor. If the tensor dimensions can be partitioned into two parts with approximately equal geometric means, the extra memory requirement for running a dimension tree is as small as $O(k\sqrt{I/P})$, which is also typically dominated by $O(I/P)$.

5.4.2 Dimension Trees

An important optimisation of the alternating-updating algorithm for NCP (and unconstrained CP) is to reuse temporary values across inner iterations [118, 121, 131, 144]. To illustrate the idea, consider a 3-way tensor $\mathcal{X}$ approximated by $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ and the two

MTTKRP computations $\mathbf{M}^{(1)} = \underline{\mathbf{X}_{(1)}(\mathbf{W}} \odot \mathbf{V})$ and $\mathbf{M}^{(2)} = \underline{\mathbf{X}_{(2)}(\mathbf{W}} \odot \mathbf{U})$ used to update factor matrices $\mathbf{U}$ and $\mathbf{V}$, respectively. The underlined parts of the expressions correspond to the computations which are shared between $\mathbf{M}^{(1)}$ and $\mathbf{M}^{(2)}$ and depend only on fixed quantities $\mathfrak{X}$ and the third factor matrix $\mathbf{W}$.

Indeed, a temporary quantity, which we refer to as a *partial MTTKRP*, can be computed and reused across the two MTTKRP expressions. We refer to the computation that combines the temporary quantity with the other factor matrix to complete the MTTKRP computation as a multi-tensor-times-vector or *multi-TTV*, as it consists of multiple operations that multiply a tensor times a set of vectors, each corresponding to a different mode.

To understand the steps of the partial MTTKRP and multi-TTV operations in more detail, we consider $\mathfrak{X}$ to be $I \times J \times K$ and $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ to have R columns. Then

$$m_{ir}^{(1)} = \sum_{j,k} x_{ijk} v_{jr} w_{kr} = \sum_{j} v_{jr} \sum_{k} x_{ijk} w_{kr} = \sum_{j} v_{jr} t_{ijr},$$

where $\mathfrak{T}$ is an $I \times J \times R$ tensor that is the result of a partial MTTKRP between tensor $\mathfrak{X}$ and the single factor matrix $\mathbf{W}$. Likewise,

$$m_{jr}^{(2)} = \sum_{i,k} x_{ijk} u_{ir} w_{kr} = \sum_{i} u_{ir} \sum_{k} x_{ijk} w_{kr} = \sum_{i} u_{ir} t_{ijr},$$

and we see that the temporary tensor $\mathfrak{T}$ can be reused and is the underlined part of $\mathbf{M}^{(1)} = \underline{\mathbf{X}_{(1)}(\mathbf{W}} \odot \mathbf{V})$ and $\mathbf{M}^{(2)} = \underline{\mathbf{X}_{(2)}(\mathbf{W}} \odot \mathbf{U})$. From these expressions, we can also see that computing $\mathfrak{T}$ (a partial MTTKRP) corresponds to a matrix-matrix multiplication, and computing each of $\mathbf{M}^{(1)}$ and $\mathbf{M}^{(2)}$ from $\mathfrak{T}$ (a multi-TTV) corresponds to R independent matrix-vector multiplications. In this case, we compute $\mathbf{M}^{(3)}$ using a full MTTKRP.

For a larger number of modes, a more general approach can organise the temporary quantities to be used over a maximal number of MTTKRPs. The general approach can yield significant benefit, decreasing the computation by a factor of approximately $N/2$

$$\begin{tikzcd}[column sep=small]
 & & & \{1,2,3,4,5\} \arrow[dll, dash, "\text{PM}"'] \arrow[drr, dash, "\text{PM}"] & & & & \\
 & \{1,2\} \arrow[dl, dash, "\text{mTTV}"'] \arrow[dr, dash, "\text{mTTV}"] & & & & \{3,4,5\} \arrow[dl, dash, "\text{mTTV}"'] \arrow[dr, dash, "\text{mTTV}"] & & \\
\{1\} & & \{2\} & & \{3\} & & \{4,5\} \arrow[dl, dash, "\text{mTTV}"'] \arrow[dr, dash, "\text{mTTV}"] & \\
 & & & & & \{4\} & & \{5\}
\end{tikzcd}$$

Figure 5.4: Dimension tree example for $N = 5$. The data associated with the root node is the original tensor, the data associated with the leaf nodes are MTTKRP results, and the data associated with internal nodes are temporary tensors. Edges labeled with PM correspond to partial MTTKRP computations, and edges labeled with mTTV correspond to multi-TTV computations.

for dense N-way tensors. The idea is introduced in Phan et al. [131], but we adopt the terminology and notation of *dimension trees* used by Kaya et al. [121, 144, 145]. In this notation, the root node is labeled $\{1, \ldots, N\}$ (we also use the notation $[N]$ for this set) and corresponds to the original tensor, a leaf is labeled $\{n\}$ and corresponds to the n^{th} MTTKRP result $\mathbf{M}^{(n)}$, and an internal node is labeled by a set of modes $\{i, \ldots, j\}$ and corresponds to a temporary tensor whose values contribute to the MTTKRP results of modes $i, \ldots, j$.

Figure 5.4 illustrates a dimension tree for the case $N = 5$. Various shapes of binary trees are possible [131, 144]. For dense tensors, the computational cost is dominated by the root's branches, which correspond to partial MTTKRP computations. We perform the splitting of modes at the root so that modes are chosen contiguously with the respect to the layout of the tensor entries in memory. In this way, each partial MTTKRP can be performed via BLAS's GEMM interface without reordering tensor entries in memory. All other edges in a tree correspond to multi-TTVs and are typically much cheaper. By organising the memory layout of temporary quantities, the multi-TTV operations can be performed via a sequence of calls using BLAS's GEMV interface. By using BLAS in our implementation, we are able to obtain high performance and on-node parallelism. For further reading, we direct the reader to the articles by Hayashi et al. [112, 146].

5.5 Software

In this section, we give a brief overview of the PLANC software package structure and ways we expect users to interact with it. PLANC consists of the following modules: shared-memory NMF, shared-memory NTF, distributed-memory NMF, and distributed-memory NTF. A detailed description of the NTF module is presented with the NMF modules following a similar hierarchy. We expect users of PLANC to add new NLS solvers and use PLANC to quickly prototype their efficacy on large problems (see [37, 113], for example). A case study of this interaction is shown. Another possible extension would be to improve the MTTKRP and matrix multiplication operations, as done by Moon et al. [123]. These routines are implemented in the abstract class, so any performance improvement immediately benefits all NLS solvers. Another possible interaction with the code would be to utilise the optimised MTTKRP and matrix multiplication operations. While they are not explicitly exposed by PLANC, extending these classes has been demonstrated by others [147].

5.5.1 Class Organisation

We briefly describe the overall class hierarchy of the PLANC package as illustrated in Fig. 5.5. PLANC offers both shared and distributed-memory implementations of NTF and the classes used in each type are distinguished by the prefix `Dist` in their names (e.g., `DistAUNTF` versus `AUNTF`). We shall cover the distributed implementation of NTF in this section. Most of the descriptions can be directly applied to the shared memory case as well.

There are broadly two types of classes present. Utility classes are primarily for managing data, setting up the processor grid, and interacting with the user. Algorithm classes perform all the computations needed for NTF and implement the different NLS solvers.

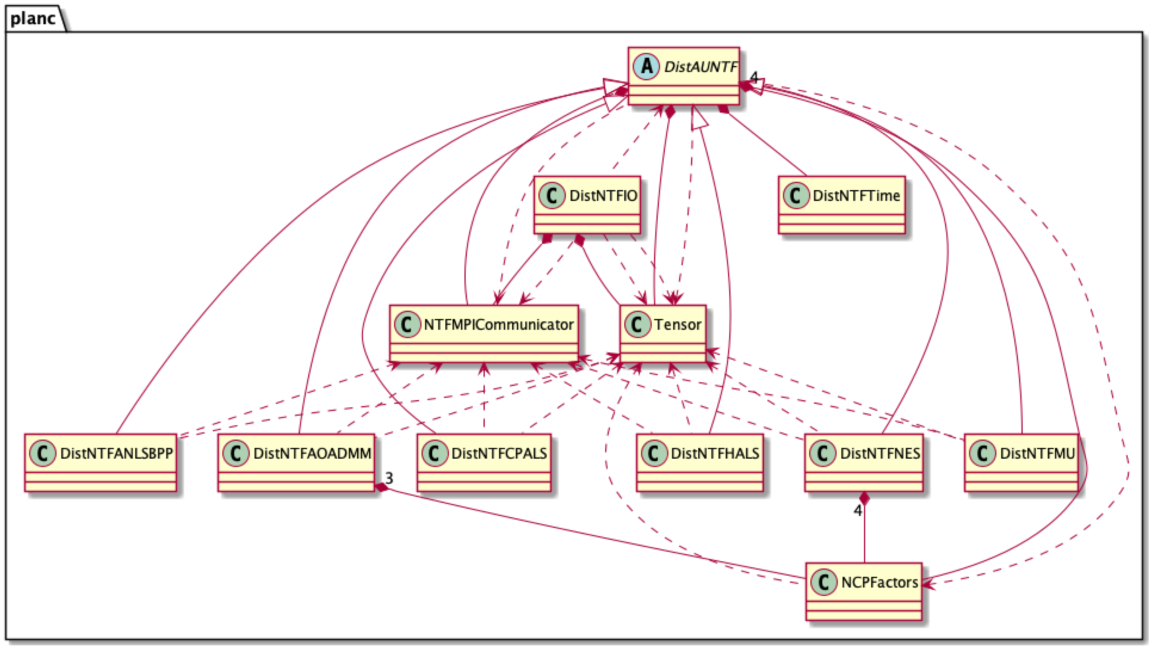


Figure 5.5: PLANC UML class diagram. Solid lines with arrow heads represent is-a relationships (inheritance): the six algorithm classes near the bottom of the diagram all derive from the abstract class `DistAUNTF` at the top. Solid lines with diamond heads represent has-a relationships (composition): NES and AOADMM classes have extra `NCPFactors` objects compared to the abstract `DistAUNTF` class, for example. Dotted lines represent dependencies.

Utility Classes

Data: The `Tensor` and `NCPFactors` classes contain the input tensor $\mathcal{X}$ and the factor matrices $[\![\boldsymbol{\lambda}; \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$. The `Tensor` class stores the input tensor as a standard data array. The tensor $\mathcal{X}$ is stored as its mode-1 unfolding $\mathbf{X}_{(1)}$ in column major order. Each processor contains its local part of the tensor (see Section 5.4.1). The `NCPFactors` class contains all the factor matrices. Each factor matrix is an Armadillo matrix [68]. The matrices are usually column normalised and the column norms are stored in the vector $\boldsymbol{\lambda}$, which is present as a member of this class (see Section 5.2.1). The vector $\boldsymbol{\lambda}$ is replicated in all processors whereas the rows of the factor matrices are distributed across the processor grid (see Section 5.4.1). There is no global view of the entire input tensor or factor matrices and care must be taken to communicate parts of either among the processor grid.

Communication: The `NTFMPICommunicator` class creates the MPI processor grid for Algorithm 9. In addition to the communicator for the entire grid, it contains a slice communicator for each mode of the processor grid. The slice communicators are used in the Reduce-Scatter of Line 13 and All-Gather of Line 19 in Algorithm 9.

I/O: The `DistNTFIO` utility class is used to read in the input tensor from user-specified files. `DistNTFIO` also contains methods to generate random tensors and to write out the factor matrices to disk. The `ParseCommandLine` class contains all the command line options available in PLANC. As the name suggests it parses the different combinations of user inputs to instantiate the driver class and run the NCP algorithm. Some example user inputs are the target rank of the decomposition, number of outer iterations, NLS solver, and regularisation parameters.

Algorithm Classes

`DistAUNTF`: This is the major workhorse class of the package. It is used to implement Algorithm 9. Some of the important member functions are:

- `computeNTF`: This is the outer iteration (Line 8 in Algorithm 9).
- `distmttkrp`: Computes the distributed MTTKRP in Line 12 and Line 13 in Algorithm 9.
- `gram_hadamard`, `update_global_gram`: These functions are used to compute the Gram matrix used in the NLS solvers.
- `computeError`: This function calculates the relative objective error of the factorisation. This can be computed efficiently by reusing the MTTKRP computation from the previous iteration (see [36, 52, 109]).

Derived classes: There exist derived classes, such as `DistNTFANLSBPP`, `DistNTFMU`, etc., for each of the NLS solvers described in Section 2.2.2. There are two main functions which are present in the derived classes which are described below. Auxiliary variables needed to implement certain NLS solvers like ADMM and Nesterov-type algorithm are also maintained in this class.

- `update`: This function is the NLS solve function. It returns the updated factor matrix using the current local MTTKRP result and global Gram matrix (see Section 2.2.2).
- `accelerate`: This implements the outer iteration acceleration (Line 8 in Algorithm 9). Currently only the Nesterov-type algorithm has an outer acceleration step.

5.5.2 Algorithm Extension

Extending PLANC to include different solvers involves the following steps.

1. Create a derived class with the newly implemented update function. This is the new NLS method needed to update the factor matrices.
2. The constructor for the new class should contain information on whether the algorithm requires an outer acceleration step. If the method requires an outer acceleration step it needs to be implemented in the derived class.
3. Update the command line parsing class `ParseCommandLine` to include additional configuration options for the algorithm.
4. Include the new algorithm as an option in the utilities and the driver files.

Case Study: We describe the different steps needed to extend PLANC to include the Nesterov-type algorithm.

1. We first create the `DistNTFNES` class which is derived from `DistAUNTF`.
2. We implement the `update` and `accelerate` functions in the derived class.
    - The Nesterov NLS updates require the previous iterate values (for the auxiliary term as described in Section 2.2.2) which may be thought of as a persistent "state" of the algorithm. We utilise an extra `NCPFactors` object to hold these variables.
    - The Nesterov update function needs synchronisation in order to terminate its local (iterative) NLS solve. This synchronisation step involves accessing the communicators found in `DistNTFMPICommunicator` class for the distributed algorithm.
    - Finally, Nesterov-type algorithms generally involve an outer acceleration step which is also implemented in the derived class `DistNTFNES`.
3. We then update the `ParseCommandLine` class to include Nesterov as an algorithm.

4. We update the driver file `distntf.cpp` to include the Nesterov algorithm.

5.6 Experiments

5.6.1 Experimental Setup

The entire study was performed on Titan, a supercomputer at the Oak Ridge Leadership Computing Facility. Titan is a hybrid-architecture Cray® XK7™ system that contains both advanced 16-core AMD® Opteron™ central processing units (CPUs) and NVIDIA® Kepler graphics processing units (GPUs). It features 299,008 CPU cores on 18,688 compute nodes, a total system memory of 710 terabytes with 32 GB on each node, and Cray's high-performance Gemini™ network.

We use Armadillo for matrix representations and operations [68]. In Armadillo, the elements of the dense matrix are stored in column-major order. For dense BLAS and LAPACK operations, we linked Armadillo with the default LAPACK/BLAS wrappers from Cray. We use the GNU C++ Compiler (g++ (GCC) 6.3.0) and Cray's MPI library. The code can also compile and run on other commodity clusters with entirely open source libraries such as OpenBLAS and OpenMPI.

5.6.2 Datasets

Mouse Data

The "Mouse" data is a 3D dataset that images a mouse's brain over time and over a sequence of identical trials [148]. Each entry of the tensor represents a measure of calcium fluorescence in a particular pixel during a time step of a single trial[3]. Each image has dimension $1{,}040 \times 1{,}392$, and the minimum number of time steps across 25 trials is 69. By flattening the pixel dimensions and discarding time steps after 69 for each trial, we obtain a tensor of size $1{,}446{,}680 \times 69 \times 25$. Every trial is performed with the same mouse and

[3]The calcium imaging is performed using an epi-fluorescence macroscope viewing the brain through an artificial crystal skull

tracks the same task. The mouse is presented with visual simulation (starting at frame 3), and after a delay is rewarded with water (starting at frame 25). The CP decomposition can be used as an unsupervised learning technique to discover underlying patterns within the data. Because the data is nonnegative, an NCP can be more easily interpreted. We show an example interpretation of a component in Fig. 5.14.

Synthetic

Our synthetic datasets are constructed from a CP model with an exact low rank with no additional noise. In this case we can confirm that the residual error of our algorithm with a random start converges to zero. For the purposes of benchmarking, we run a fixed number of iterations of the NCP algorithms rather than using a convergence check.

5.6.3 Performance Breakdown Categories

The list below gives a brief description of all the categories shown in the breakdown plots and their role in the overall algorithm.

1. Gram: The Gram matrix computation includes both the Gram computation of the local factor matrices and the Hadamard product of global Gram matrices for each factor matrix. This computation is performed on each inner iteration but is cheap under the assumption that k is small relative to the tensor dimensions.

2. NLS: The cost of a nonnegative least-squares update can vary drastically with the algorithm used. The various characteristics that may affect run time for each NLS algorithm are discussed in Section 5.6.4.

3. MTTKRP: The (partial) MTTKRP is a purely local computation performed on each node and can be offloaded to the GPU. Using the dimension-tree optimisation (Section 5.4.2), we perform two partial MTTKRPs for each outer iteration, regardless of

the number of modes N. Both operations are cast as GEMM calls, where the dimensions are given by the product of the first S mode dimensions (S is the split mode), the product of the last $N - S$ mode dimensions, and the rank k.

4. MultiTTV: The MultiTTVs are purely local computations performed on each node. Each MultiTTV is cast as a set of k GEMV calls which are typically memory bandwidth bound.

5. ReduceScatter: the ReduceScatter collective is used to sum MTTKRP results and distribute portions of the sum appropriately across processors. It is called for each inner iteration.

6. AllGather: The AllGather collective is used to collect the updated factor matrices to each processor in the slice corresponding to the mode being updated. It is called after each inner iteration.

7. AllReduce: The AllReduce is used to compute the Gram matrices and for computing norms and other quantities required for stopping criteria of some algorithms.

5.6.4 Updating Algorithm Distinctions

Table 5.2 highlights the distinct aspects of each updating algorithm that can affect performance. The rows of Table 5.2 denote the different local update algorithms implemented in PLANC. The algorithms names and acronyms in order from top to bottom in Table 5.2 are as follows: Unconstrained CP (UCP), Multiplicative Update (MU), Hierarchical Least Squares (HALS), Block Principal Pivoting (BPP), Alternating Direction Method of Multipliers (ADMM), and Nesterov-type algorithm (NES). The aspects of each algorithm that are displayed in Table 5.2 are as follows:

- Communication: A check mark and description in this column indicates that the local update algorithm requires some amount of communication. For example, the

Table 5.2: Characteristics of the various update algorithms that can potentially affect performance. The columns are as follows: 1) if the local update requires communication, 2) if the update requires additional MTTKRP computations, 3) if the local update itself is iterative, 4) if the algorithm's performance is significantly impacted by parameter tuning. A ✓ corresponds to the algorithm having the characteristic, and a $\times$ means it does not.

Alg	Communication	Extra MTTKRPs	Iterative	Tuning
UCP	$\times$	$\times$	$\times$	$\times$
MU	$\times$	$\times$	$\times$	$\times$
HALS	✓ Column Norms	$\times$	$\times$	$\times$
BPP	$\times$	$\times$	✓	$\times$
ADMM	✓ Stopping Criteria	$\times$	✓	✓ Step Size
NES	✓ Stopping Criteria	✓	✓	✓ See Section 2.2.2

HALS algorithm requires the communication of the updated column norms. Additional communication requirements can affect performance by incurring additional latency and bandwidth costs. These penalties become significant when the number of processors is high.

- Extra MTTKRPs: the NES algorithm has an acceleration step which requires an additional MTTKRP to be performed. This extra computation can potentially increase the run time if the acceleration step does not decrease the objective function. Experimentally, on both real and synthetic datasets, we observe that NES run time is significantly increased by the additional MTTKRP computations.

- Iteration: This column indicates the iterative nature of the local update algorithm. Note that all of the algorithms we present here are iterative in terms of the outer iteration. UCP, MU, and HALS all have closed-form formulas for the inner iteration, meaning the number of flops can be explicitly computed as a function of the problem size. The rest of the algorithms have flop and communication requirements dependent on the number of iterations it takes the algorithm to converge for a particular local update.

- Tuning: Many optimisation algorithms require some tunable input parameters which can impact performance. For example, setting a step size is a frequent requirement for gradient based optimisation algorithms when an exact line search is too computationally expensive.

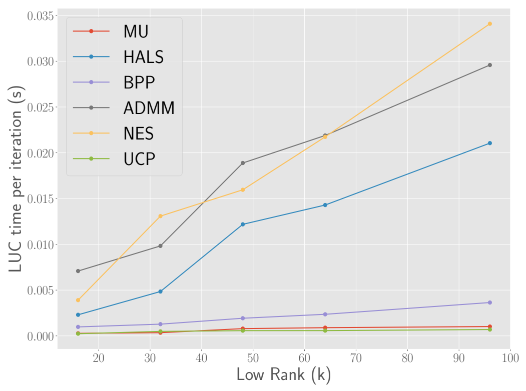
5.6.5 Microbenchmarks

Per-Iteration Timing Comparison across Algorithms

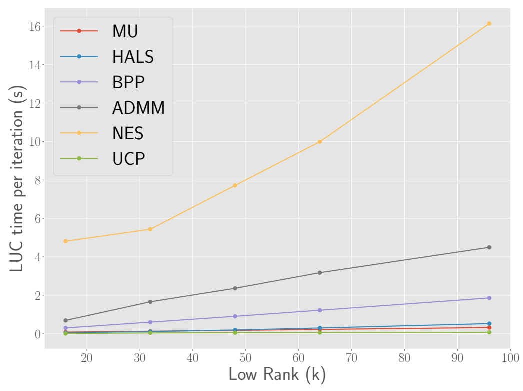
Fig. 5.6 shows the "local update computation" time taken by the different updating algorithms for various low-rank values on a synthetic dataset and the Mouse dataset. The synthetic tensor (Fig. 5.6a) involves about 20 LUCs per processor per iteration whereas the Mouse data (Fig. 5.6b) has about 20,000 updates per processor per iteration, which accounts for the difference in the scales of the time seen in the figures. MU and UCP are the cheapest algorithms with NES being the most expensive. HALS, ADMM and NES algorithms all communicate in their update steps and this significantly affects their runtimes, see Fig. 5.12b. NES has the most expensive inner iteration, as it involves an eigendecomposition of the Gram matrix and up to 20 iterations of the NLS updater. ADMM has the second most expensive inner iteration with up to 5 iterations of the acceleration step. HALS on the other hand, doesn't have a very expensive inner iteration but needs a synchronization to normalise every updated column of the factor matrix before proceeding to the next column, causing a slowdown.

Comparison across Processor Grids

Figure 5.7 gives a processor grid comparison for a 3D cubical tensor of size 512. The distributed MTTKRP time dominates the overall run time and we observe that an even processor distribution results in the best achieved performance for all update algorithms. This difference in run times is partially accounted for by GEMM performance due to the different shapes of the matrices involved. Besides choosing an even processor grid we see



(a) Synthetic data: 384×384×384×384 tensor on 81 nodes arranged in 3×3×3×3 grid



(b) Mouse data: 1,447,680 × 69 × 25 tensor on 64 nodes arranged in 64 × 1 × 1 grid

Figure 5.6: Per Iteration Local Update Computation (LUC) comparison of NLS algorithms on 4D synthetic and and 3D Mouse tensors

that configurations 1 through 6 have quite stable run times with the exception of the NES algorithm, which is due to variable number of MTTKRPs needed for acceleration.

Comparison between CPU and GPU Matrix Multiplication Offloading

Figure 5.8 shows comparison in run times between performing partial MTTKRPs on the CPU versus offloading to the GPU as rank increases. We only plot NES and UCP algorithms as they capture all the behaviours exhibited in offloading the matrix multiplication. All the other NLS algorithms perform similarly to UCP. As expected, the CPU run times increase linearly with k as the operation count for UCP is dominated by the MTTKRP which is linear in k. In the case of the GPU execution, the tested sizes of k are never large enough to saturate the GPU, yielding flat run times even as the rank increases. The NES algorithm takes additional time for both the CPU and GPU executions due to the additional MTTKRPs. In this case, for the chosen tensor size and rank, it is always beneficial to offload the GEMM calls to the GPU, and the maximum achieved speedup with GPU offloading is about 7×. However, we have observed in other experiments that NVBLAS can make the incorrect decision to offload the computation to the GPU when it is faster to perform it on the CPU.

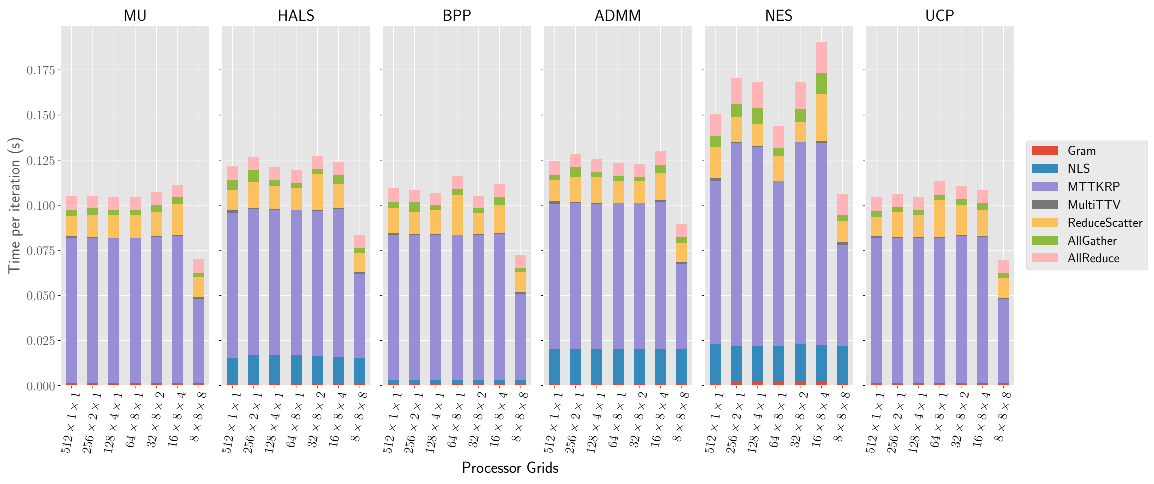


Figure 5.7: Processor grid sweep of a $512 \times 512 \times 512$ synthetic 3D low-rank tensor on 512 nodes with low rank 96.

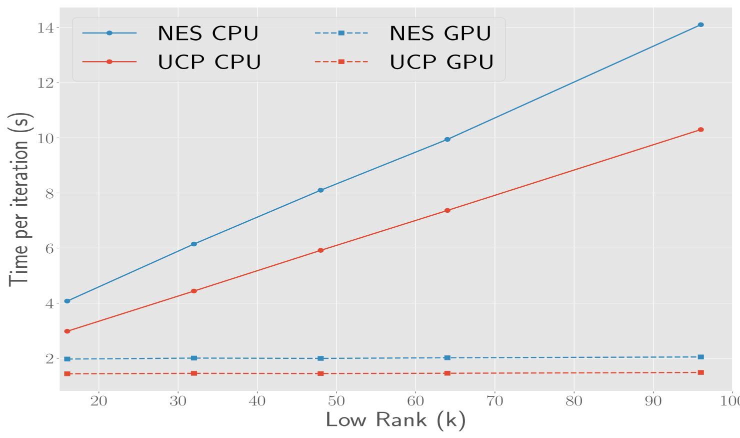
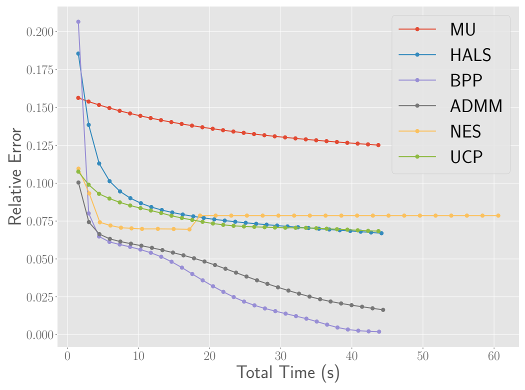
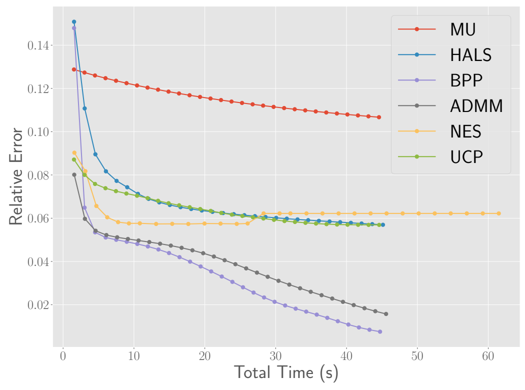


Figure 5.8: Timing comparison of CPU and GPU offloading on 4D synthetic low-rank tensor of size $384{\times}384{\times}384{\times}384$ on $3{\times}3{\times}3{\times}3$ processor grid with varying ranks. Only two updating algorithms are shown because all other algorithms had results similar to UCP.

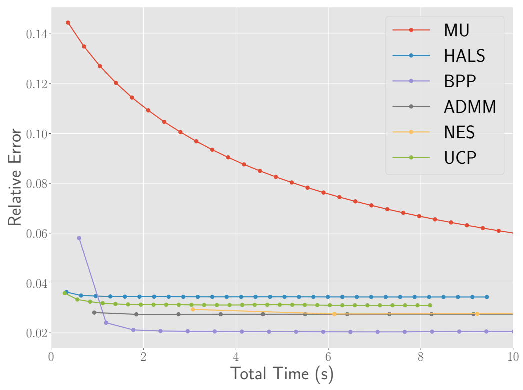


(a) Low rank $k = 64$

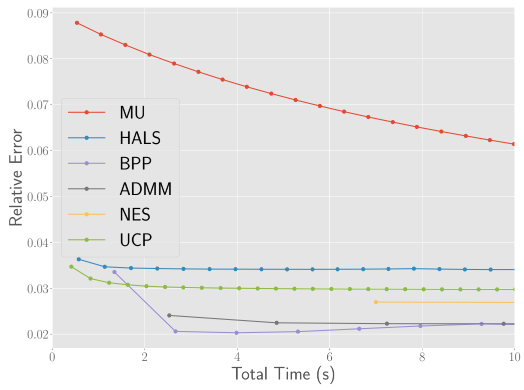


(b) Low rank $k = 96$

Figure 5.9: Relative error over time comparison of updating algorithms on 4D synthetic low-rank tensor of size $384{\times}384{\times}384{\times}384$ on $3{\times}3{\times}3{\times}3$ processor grid. Each data point corresponds to an outer iteration.



(a) Low rank $k = 16$



(b) Low rank $k = 48$

Figure 5.10: Relative error comparison of updating algorithms on 3D real-world low-rank tensor of size $1447680 \times 69 \times 25$ on a 64 Titan nodes as a $64 \times 1 \times 1$ processor grid for 10 seconds. Each data point corresponds to an outer iteration.

5.6.6 Convergence comparison across Algorithms

Figures 5.9 and 5.10 show convergence comparisons (error versus time) for each of the updating algorithms on synthetic low-rank and Mouse datasets, using two different target ranks each. Every algorithm is run for a fixed number (30) of outer iterations for fair comparison. For the Mouse data in Fig. 5.10, we show only the first 10 seconds because nearly all algorithms are converging within 30 iterations. The initialised random factors are the same for all algorithms in both tests, and the synthetic tensor is the same for all algorithms.

In both the synthetic and real-world cases BPP achieves the lowest approximation error in the shortest amount of time. Overall results are as expected, such as MU achieving the worst error and ADMM achieving the second best in all cases. On the real-world dataset the best algorithms, ADMM and BPP, achieve relative errors of 2-3%.

5.6.7 Scaling Studies

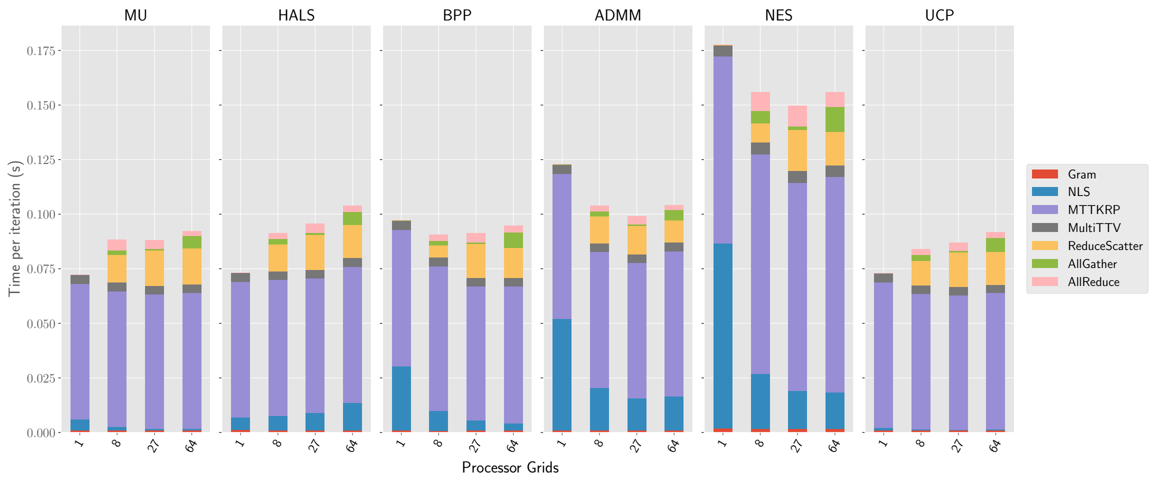
Weak Scaling (Synthetic Data)

We performed weak-scaling analysis on two different cubical tensors with 3 and 4 modes (by cubical, we mean all modes have the same dimension). Fig. 5.11 shows the time breakdown for scaling up to 64 nodes of Titan for the 3D case and 16,384 nodes for the 4D tensor. In each experiment the size of the local tensor is kept constant at dimension 128 in each mode for all the runs. As expected, the run time is dominated by the cost to compute the MTTKRP, and the domination is more extreme for higher mode tensors. Moreover, we see reasonable weak scaling as the figures remain relatively flat over all processor sizes.

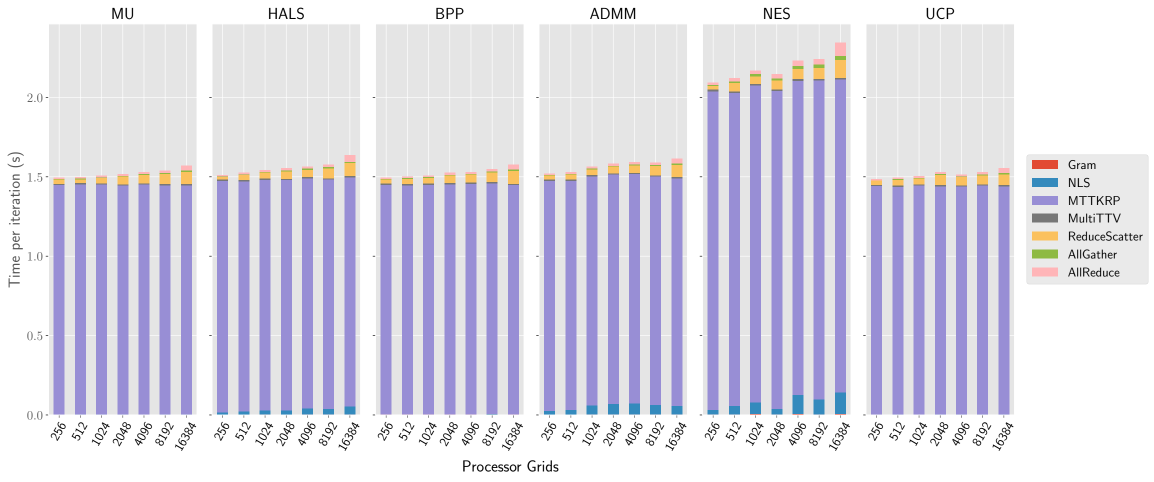
The variations occur mainly due to the NLS and communication portions of the algorithm. These do matter in general and especially for the 3D case where the MTTKRP cost is often comparable to NLS times, especially for smaller number of processors. However NLS times scale well since they split along processor slices rather than fibers and soon become negligible for large processor grids. The amount of communication per processor remains constant but latency costs increase slowly as we scale up.

Strong Scaling (Synthetic Data)

We run strong-scaling experiments on two synthetic cubical tensors, one 3D and one 4D. Figure 5.12 contains these results for each of the local update algorithms ranging from 1 to 16,384 processors. Since the tensors are cubical, we try to maintain the processor grids to be as close to cubical as well. For the 3D case the grids used are $2 \times 2 \times 2$, $4 \times 2 \times 2$,



(a) Synthetic 3D low rank



(b) Synthetic 4D low rank

Figure 5.11: Weak scaling on synthetic 3D and 4D low-rank tensors. For the 3D case, the input tensors are of size $128\times128\times128$, $256\times256\times256$, $378\times378\times378$ and $512\times512\times512$ on 1, 8, 27 and 64 Titan Nodes. The 4D input tensors are $128\times128\times128\times128$, $256\times256\times256\times256$, $512\times512\times512\times512$, $1024\times512\times512\times512$, $1024\times512\times1024\times512$, $1024\times1024\times1024\times512$, $1024\times1024\times1024\times1024$, $2048\times1024\times1024\times1024$, $2048\times1024\times2048\times1024$ on 1, 16, 256, 512, 1024, 2048, 4096, 8192, and 16384 Titan nodes. For all experiments, the low rank is 96.

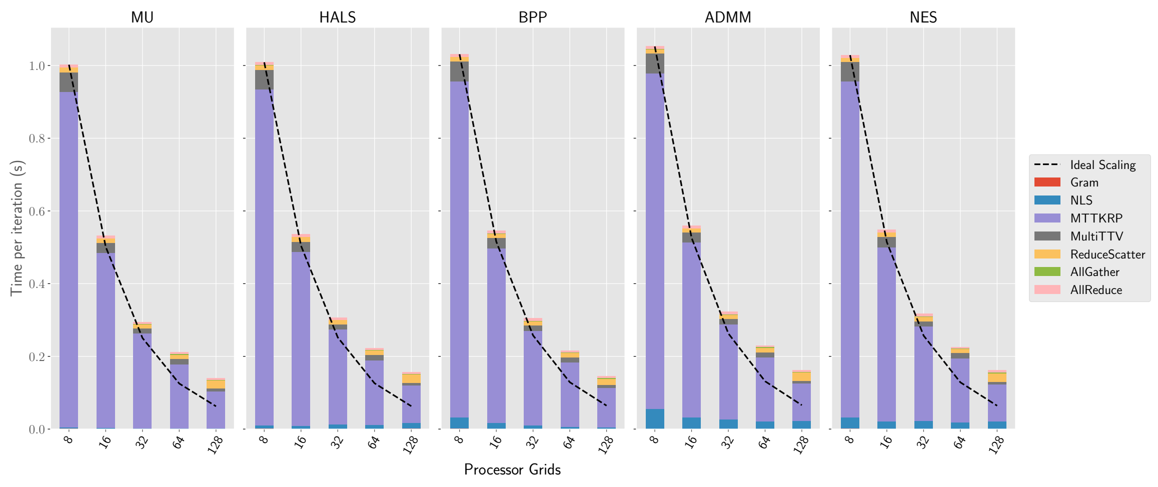
$4 \times 4 \times 2$, $4 \times 4 \times 4$ and $8 \times 4 \times 4$. Similarly, the grids used for the 4D case are $4 \times 4 \times 4 \times 4$, $8 \times 4 \times 4 \times 4$, $8 \times 4 \times 8 \times 4$, $8 \times 8 \times 8 \times 4$, $8 \times 8 \times 8 \times 8$, $16 \times 8 \times 8 \times 8$ and $16 \times 8 \times 16 \times 8$.

We see similar behaviour for the 3D and 4D case. For the 3D tensor (Fig. 5.12a), we observe good strong scaling up to about 32 nodes and continue to see speed up through 128 nodes. Similarly for the 4D case (Fig. 5.12b), the algorithms scale well up to about 1024 nodes and continue to reduce time until 8,192 nodes; we observe a slowdown when scaling to 16,384 nodes.

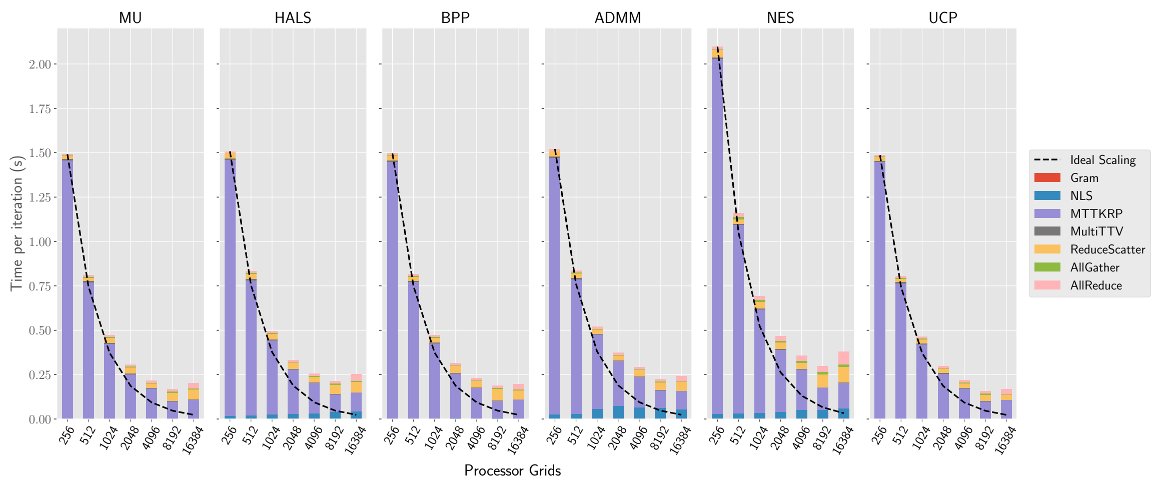
One reason for the limit of strong scaling is the communication overheads of AllGather, AllReduce, and ReduceScatter, which become more significant for more processors. The stronger effect is the performance of the local matrix multiplications within MTTKRP's dimension tree. The smallest dimension in the matrix multiplication is typically the low-rank k, which is 96 in these experiments. For a cubical tensor of odd dimension, in our case three, the dimension tree optimisation is often forced to cast partial MTTKRP into a very rectangular matrix multiplication, depending on the processor grid. Two of the local dimensions must be grouped together while the other is left alone. This means that the largest dimension would need to be close to the product of the other two in order for there to be an approximately square matrix (multiplying a tall-skinny matrix with k columns, for example). The shape and size of these local multiplications hurts the efficiency of the local computation cost and is the biggest hindrance to strong scalability for these examples.

Strong Scaling (Real World)

Figure 5.13 show strong-scaling results on the Mouse dataset. We use a 1D $P \times 1 \times 1$ processor grid throughout the experiment. The results are in line with the synthetic results. We achieve near-perfect scaling up to around 32 nodes and still improve runtimes through to 512 nodes. At 1,024 nodes the NLS algorithms, which communicate during the solve steps, perform far worse and show up to $2\times$ slowdown. The non-communicating solvers also degrade in performance but more gracefully.



(a) Synthetic 3D low rank - $1024 \times 1024 \times 1024$ tensor



(b) Synthetic 4D low rank - $512 \times 512 \times 512 \times 512$ tensor

Figure 5.12: Strong scaling on synthetic 3D and 4D low rank tensors with low rank 96.

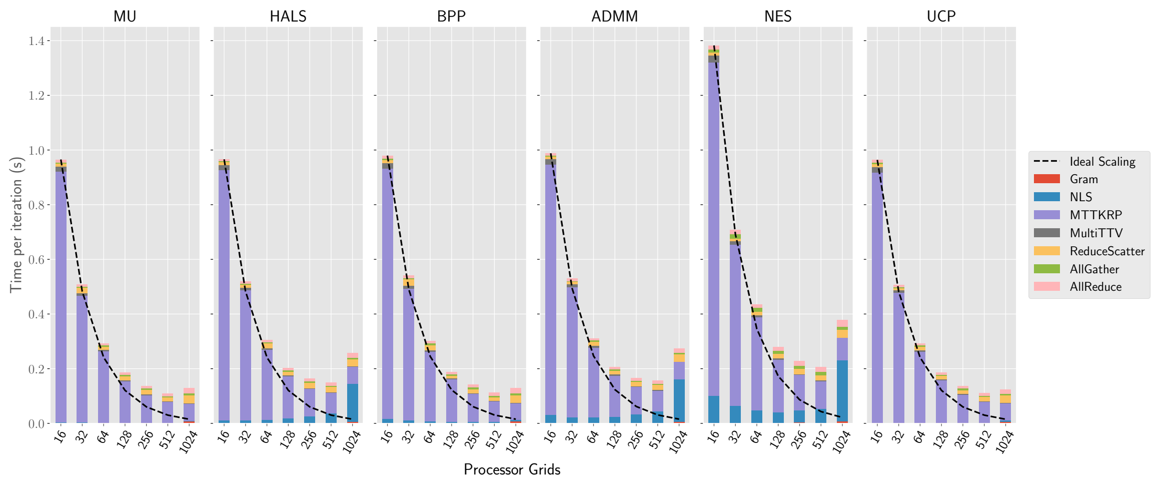
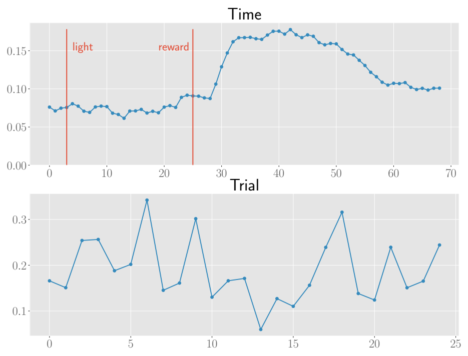
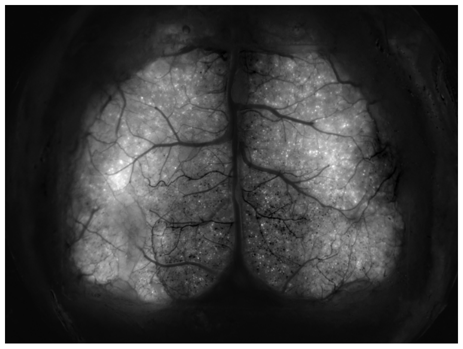


Figure 5.13: Strong scaling on Mouse dataset.



(a) Time and trial factors



(b) Brain image factor

Figure 5.14: Visualisation of component 22 of rank-32 NCP decomposition of Mouse data. The time factor visualisation has been annotated with key time points, showing when the light stimulus was applied and when the water reward was given. The y-axes of the time and trial factors are unitless loading weights. The brain image factor has been reshaped back into original dimensions to visualise pixels with large weights in the component.

5.6.8 Mouse Data Results

The NCP decomposition of the Mouse data can be used to interpret brain patterns in response to the light stimulus and water reward given to the mouse. For example, Fig. 5.14 shows a visualisation of the factors of the 22nd component of the rank-32 NCP decomposition. From the time factor, we see a marked increase in the importance of the component after the reward time frame, which suggests the activity is a response to the reward. Because the same mouse undergoes 25 identical trials, we expect to see no pattern in the time factor of each component. The factors have been normalised, and the absolute magnitude of the y-axis reflects this. The pixel factor has been reshaped to an image of the same dimensions of the original data. We observe higher intensity values in the somatosensory cortex (middle, left), which is associated with bodily sensation. This component, possibly representing a sensory response to the water reward, aligns well with the findings of cell-based analysis, which also identified neurons in the somatosensory cortex with intensities that peaked quickly after the reward time frame [148, Figure 3].

5.7 Summary

We present PLANC, a software library for nonnegative low rank approximations that works for tensors of any number of modes and scales to large datasets and high processor counts. The software framework can be adapted to use any NLS algorithm within the context of alternating-updating algorithms. We use an efficient parallel algorithm that minimises communication cost of the bottleneck MTTKRP computation. Dimension trees are also utilised to avoid unnecessary recomputation of the local MTTKRP.

CHAPTER 6
CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation advocates CLRA as the framework of choice for mining large and complex datasets on massively parallel computing systems. This work builds on the rich history of methods developed by the numerical linear algebra community which can be repurposed for data analytics. In particular, it focuses on the recent paradigm of designing algorithms to counter the growing gap between computation and communication costs on modern computing hardware. By identifying and casting the bottleneck computations in a communication-avoiding manner we are able to scale up our algorithms efficiently.

We focus on NMF, the popular CLRA technique for clustering, as our candidate algorithm and identify its bottlenecks (Section 2.3.2). We then design a communication-optimal distributed matrix-multiplication to overcome this bottleneck. With the addition of this kernel, we are able to implement multiple different solvers for NMF (Section 2.2.2), introduce symmetry constraints (Chapter 3), and extend our algorithms to handle multimodal inputs (Chapter 4). We also create SymNMF and JointNMF algorithms which effectively utilise second-order information without incurring prohibitive computational and memory overheads (Sections 3.4 and 4.3.3). We then develop a communication-optimal MTTKRP kernel used in the computation of the NCP approximation of tensors, a natural extension of NMF (Chapter 5).

All the methods mentioned above are present in the open-source software package PLANC. It is efficient and has been scaled to 16,384 nodes on the Titan supercomputer (Section 5.6.7). This efficiency has enabled the processing of large datasets from various domains including graphs with over a billion edges [35], satellite images with 24 million pixels [37], and text corpora with over 37 million documents and 483 million citations [149], amongst others.

Some avenues for future directions are listed below in addition to those mentioned in the end of chapter summaries.

Variants of NCP. With the addition of the MTTKRP kernel in PLANC, we expect the standard NCP formulation to be extended to multiple different cases in the same manner as the matrix-multiplication kernel. Second-order algorithms in the style of Vervliet et al. are good starting points for developing PLANC further [82, 142]. Coupled matrix and tensor factorisations would benefit with the availability of both distributed kernels [150].

GPU support for solvers. The different NLS solvers described in this dissertation have all been implemented in the CPU. GPUs account for most of the flops in modern computing systems and are already used to compute some of the more structured computations like matrix-multiplications in PLANC. Extending GPU support to all the NLS solvers or designing methods which effectively overlap the CPU and GPU phases of these algorithms is a promising avenue to explore [123].

Structured sparsity. Many CLRA models are formulated with a notion of sparsity baked into the algorithm. By sparsity, we mean that either implicitly or explicitly a sparse matrix is introduced into the objective formulation. Some examples are:

1. One such scenario is when we are faced with missing or incomplete data in the input matrix to be approximated [151]. Typically, we would not want to fit our models to the missing entries. The modified objective for the case of NMF with missing values is shown here.

   $$\min_{\mathbf{W} \geq \mathbf{H} \geq 0} \|\mathbf{M} * (\mathbf{X} - \mathbf{W}\mathbf{H})\|_F^2$$

   $\mathbf{M}$ is a boolean masking matrix which has zeros in the positions of the missing entries.

2. Cross-validation is a popular technique for hyperparamter selection in CLRA [152–154]. The entries being “held-out” are masked out in a similar manner to the missing entries case.

3. In the semi-supervised setting when a certain number of entries have a fixed value, masking matrices are used to ensure that those particular entries are not modified [101].

4. In the “robust” LRA case, we often model the input as a low-rank signal matrix plus an additive corruptions matrix [13, 39, 155]. Masking matrices appear again as thresholding operations in this case.

Designing efficient algorithms for masked computations, especially when the masks can be controlled like in the semi-supervised and cross-validation settings, would aid in making CLRA more practical for the analyst.

REFERENCES

[1] Madeleine Udell et al. “Generalized Low Rank Models”. In: *Foundations and Trends® in Machine Learning* 9.1 (2016), pp. 1–118.

[2] Charu C Aggarwal. *Data mining: the textbook*. Vol. 1. Springer.

[3] Chun-Wei Tsai et al. “Big data analytics: a survey”. In: *Journal of Big data* 2.1 (2015), pp. 1–32.

[4] Aydin Buluc et al. “Randomized algorithms for scientific computing (RASC)”. In: *arXiv preprint arXiv:2104.11079* (2021).

[5] *Wikipedia:Size of Wikipedia*. https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia. Accessed on 2022.04.04.

[6] Nahum Kiryati and Yuval Landau. “Dataset growth in medical image analysis research”. In: *Journal of imaging* 7.8 (2021), p. 155.

[7] *ITER Home Page*. https://www.iter.org. Accessed on 2022.04.04.

[8] Grey Malone Ballard. *Avoiding communication in dense linear algebra*. University of California, Berkeley, 2013.

[9] John David McCalpin. “Memory bandwidth and system balance in hpc systems”. In: *UT Faculty/Researcher Works* (2016).

[10] Nicolas Gillis. *Nonnegative Matrix Factorization*. SIAM, 2020.

[11] I. T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. New York: Springer-Verlag, 2002. ISBN: 0-387-95442-2.

[12] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU press, 2013.

[13] Emmanuel J Candès et al. “Robust principal component analysis?” In: *Journal of the ACM (JACM)* 58.3 (2011), pp. 1–37.

[14] Daniel D Lee and H Sebastian Seung. “Learning the parts of objects by non-negative matrix factorization”. In: *Nature* 401.6755 (1999), pp. 788–791.

[15] Michael W Mahoney and Petros Drineas. “CUR matrix decompositions for improved data analysis”. In: *Proceedings of the National Academy of Sciences* 106.3 (2009), pp. 697–702.

[16] Jack J Dongarra et al. “Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs”. In: *ACM Transactions on Mathematical Software (TOMS)* 14.1 (1988), pp. 18–32.

[17] L Susan Blackford et al. *ScaLAPACK users’ guide*. SIAM, 1997.

[18] Edward Anderson et al. *LAPACK users’ guide*. SIAM, 1999.

[19] Ernie Chan et al. “Collective communication: theory, practice, and experience”. In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1206.

[20] Satish Balay et al. “PETSc users manual”. In: (2019).

[21] Phillip Colella. “Defining software requirements for scientific computing”. In: (2004).

[22] Madeleine Udell and Alex Townsend. “Why are big data matrices approximately low rank?” In: *SIAM Journal on Mathematics of Data Science* 1.1 (2019), pp. 144–160.

[23] Matthew Turk and Alex Pentland. “Eigenfaces for recognition”. In: *Journal of cognitive neuroscience* 3.1 (1991), pp. 71–86.

[24] Ravi Kannan, Santosh Vempala, and Adrian Vetta. “On clusterings: Good, bad and spectral”. In: *Journal of the ACM (JACM)* 51.3 (2004), pp. 497–515.

[25] Ulrike Von Luxburg. “A tutorial on spectral clustering”. In: *Statistics and computing* 17.4 (2007), pp. 395–416.

[26] Hyunsoo Kim and Haesun Park. “Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis”. In: *Bioinformatics* 23.12 (2007), pp. 1495–1502.

[27] Jingu Kim, Yunlong He, and Haesun Park. “Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework”. In: *Journal of Global Optimization* 58.2 (2014), pp. 285–319.

[28] David M Blei. “Probabilistic topic models”. In: *Communications of the ACM* 55.4 (2012), pp. 77–84.

[29] Hanghang Tong and Ching-Yung Lin. “Non-negative residual matrix factorization with application to graph anomaly detection”. In: *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM. 2011, pp. 143–153.

[30] Charu C Aggarwal. “An introduction to outlier analysis”. In: *Outlier analysis*. Springer, 2017, pp. 1–34.

[31] David Patterson et al. “Carbon emissions and large neural network training”. In: *arXiv preprint arXiv:2104.10350* (2021).

[32] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 `[cs.LG]`.

[33] Edo Liberty et al. “Randomized algorithms for the low-rank approximation of matrices”. In: *Proceedings of the National Academy of Sciences* 104.51 (2007), pp. 20167–20172.

[34] Jack J Dongarra et al. “An introduction to the MPI standard”. In: *Communications of the ACM* 18 (1995).

[35] R. Kannan, G. Ballard, and H. Park. “MPI-FAUN: An MPI-Based Framework for Alternating-Updating Nonnegative Matrix Factorization”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.3 (Mar. 2018), pp. 544–558.

[36] Srinivas Eswar et al. “PLANC: Parallel Low Rank Approximation with Nonnegativity Constraints”. In: *ACM Transactions on Mathematical Software (TOMS)* (2021).

[37] Srinivas Eswar et al. “Distributed-memory parallel symmetric nonnegative matrix factorization”. In: *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, pp. 1041–1054.

[38] Srinivas Eswar et al. “Distributed-memory parallel joint nonnegative matrix factorization”. In: *In preparation*. 2022.

[39] Srinivas Eswar et al. “ORCA: Outlier detection and Robust Clustering for Attributed graphs”. In: *Journal of Global Optimization* (2021).

[40] Pentti Paatero and Unto Tapper. “Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values”. In: *Environmetrics* 5.2 (1994), pp. 111–126.

[41] Stephen A Vavasis. “On the complexity of nonnegative matrix factorization”. In: *SIAM Journal on Optimization* 20.3 (2010), pp. 1364–1377.

[42] Da Kuang, Sangwoon Yun, and Haesun Park. “SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering”. In: *Journal of Global Optimization* 62.3 (2015), pp. 545–574.

[43] Dimitri P Bertsekas. “Nonlinear programming. athena scientific”. In: *Belmont, MA* (1999).

[44] Dimitri P Bertsekas. *Corrections for the book NONLINEAR PROGRAMMING*. Accessed on 2022.05.21.

[45] Luigi Grippo and Marco Sciandrone. “On the convergence of the block nonlinear Gauss–Seidel method under convex constraints”. In: *Operations research letters* 26.3 (2000), pp. 127–136.

[46] Ngoc-Diep Ho. “Nonnegative Matrix Factorization Algorithms and Applications”. PhD thesis. Université Catholique De Louvain, 2008.

[47] Andrzej Cichocki and Anh-Huy Phan. “Fast local algorithms for large scale nonnegative matrix and tensor factorizations”. In: *IEICE transactions on fundamentals of electronics, communications and computer sciences* 92.3 (2009), pp. 708–721.

[48] Chih-Jen Lin. “Projected gradient methods for nonnegative matrix factorization”. In: *Neural computation* 19.10 (2007), pp. 2756–2779.

[49] Jingu Kim and Haesun Park. “Fast nonnegative matrix factorization: An active-set-like method and comparisons”. In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3261–3281.

[50] Kejun Huang, Nicholas D Sidiropoulos, and Athanasios P Liavas. “A flexible and efficient algorithmic framework for constrained matrix and tensor factorization”. In: *IEEE Transactions on Signal Processing* 64.19 (2016), pp. 5052–5065.

[51] Stephen Boyd et al. “Distributed optimization and statistical learning via the alternating direction method of multipliers”. In: *Foundations and Trends® in Machine learning* 3.1 (2011), pp. 1–122.

[52] Athanasios P Liavas et al. “Nesterov-based parallel algorithm for large-scale nonnegative tensor factorization”. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2017, pp. 5895–5899.

[53] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. “Nonnegative matrix and tensor factorization [lecture notes]”. In: *IEEE signal processing magazine* 25.1 (2008), pp. 142–145.

[54] Chao Liu et al. “Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce”. In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 681–690.

[55] Rainer Gemulla et al. “Large-scale matrix factorization with distributed stochastic gradient descent”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 69–77.

[56] Jiangtao Yin, Lixin Gao, and Zhongfei Mark Zhang. “Scalable nonnegative matrix factorization with block-wise updates”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2014, pp. 337–352.

[57] Ruiqi Liao et al. “CloudNMF: a MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets”. In: *Genomics, proteomics & bioinformatics* 12.1 (2014), pp. 48–51.

[58] Alex Beutel et al. “Flexifact: Scalable flexible factorization of coupled tensors on hadoop”. In: *Proceedings of the 2014 SIAM international conference on data mining*. SIAM. 2014, pp. 109–117.

[59] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. “A high-performance parallel algorithm for nonnegative matrix factorization”. In: *ACM SIGPLAN Notices* 51.8 (2016), pp. 1–11.

[60] Hong Jia-Wei and Hsiang-Tsung Kung. “I/O complexity: The red-blue pebble game”. In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 1981, pp. 326–333.

[61] Alok Aggarwal, Ashok K Chandra, and Marc Snir. “Communication complexity of PRAMs”. In: *Theoretical Computer Science* 71.1 (1990), pp. 3–28.

[62] Dror Irony, Sivan Toledo, and Alexander Tiskin. “Communication lower bounds for distributed-memory matrix multiplication”. In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026.

[63] James Demmel et al. “Communication-optimal parallel recursive rectangular matrix multiplication”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 261–272.

[64] Hussam Al Daas et al. “Tight Memory-Independent Parallel Matrix Multiplication Communication Lower Bounds”. In: *arXiv preprint arXiv:2205.13407* (2022).

[65] Hussam Al Daas et al. “Brief Announcement: Tight Memory-Independent Parallel Matrix Multiplication Communication Lower Bounds”. In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 2022, pp. 445–448.

[66] Ramesh C Agarwal et al. “A three-dimensional approach to parallel matrix multiplication”. In: *IBM Journal of Research and Development* 39.5 (1995), pp. 575–582.

[67] Partnership for an Advanced Computing Environment. *Phoenix User Guide*. Accessed on 2022.07.26.

[68] Conrad Sanderson. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Tech. rep. NICTA, 2010.

[69] Arnaud Vandaele et al. “Efficient and non-convex coordinate descent for symmetric nonnegative matrix factorization”. In: *IEEE Transactions on Signal Processing* 64.21 (2016), pp. 5571–5584.

[70] François Moutier, Arnaud Vandaele, and Nicolas Gillis. “Off-diagonal symmetric nonnegative matrix factorization”. In: *Numerical Algorithms* (2021), pp. 1–25.

[71] Rundong Du et al. “Hierarchical community detection via rank-2 symmetric nonnegative matrix factorization”. In: *Computational social networks* 4.1 (2017), p. 7.

[72] Rundong Du, Barry Drake, and Haesun Park. “Hybrid clustering based on content and connection structure using joint nonnegative matrix factorization”. In: *Journal of Global Optimization* 74.4 (2019), pp. 861–877.

[73] Pablo Arbelaez et al. “Contour detection and hierarchical image segmentation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.5 (2010), pp. 898–916.

[74] Scott Deerwester et al. “Indexing by latent semantic analysis”. In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407.

[75] Daniel D. Lee and H. Sebastian Seung. “Algorithms for Non-Negative Matrix Factorization”. In: *Proceedings of the 13th International Conference on Neural Information Processing Systems*. NIPS’00. Denver, CO: MIT Press, 2000, pp. 535–541.

[76] Gordon Christie et al. “Functional map of the world”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 6172–6180.

[77] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342.

[78] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2e. New York, NY, USA: Springer, 2006.

[79] Petros Drineas et al. “Clustering large graphs via the singular value decomposition”. In: *Machine learning* 56.1 (2004), pp. 9–33.

[80] Da Kuang, Chris Ding, and Haesun Park. “Symmetric nonnegative matrix factorization for graph clustering”. In: *Proceedings of the 2012 SIAM international conference on data mining*. SIAM. 2012, pp. 106–117.

[81] Ron Zass and Amnon Shashua. “A unifying approach to hard and probabilistic clustering”. In: *Tenth IEEE International Conference on Computer Vision (ICCV’05) Volume 1*. Vol. 1. IEEE. 2005, pp. 294–301.

[82] N Vervliet and L De Lathauwer. “Numerical optimization-based algorithms for data fusion”. In: *Data Handling in Science and Technology*. Vol. 31. Elsevier, 2019, pp. 81–128.

[83] Navjot Singh et al. *Comparison of Accuracy and Scalability of Gauss-Newton and Alternating Least Squares for CP Decomposition*. Tech. rep. arXiv, 2019.

[84] Zhihui Zhu et al. “Dropping symmetry for fast symmetric nonnegative matrix factorization”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[85] Sudharshan S Vazhkudai et al. “The design, deployment, and evaluation of the CORAL pre-exascale systems”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 661–672.

[86] Erich Strohmaier et al. *Top500*.

[87] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010.

[88] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[89] Hasan Metin Aktulga et al. “Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1213–1222.

[90] Sireyya Emre Kurt et al. “Characterization of data movement requirements for sparse matrix computations on GPUs”. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE. 2017, pp. 283–293.

[91] Changwan Hong et al. “Efficient sparse-matrix multi-vector product on GPUs”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 66–79.

[92] Benjamin C. Lee et al. *Performance Optimizations and Bounds for Sparse Symmetric Matrix - Multiple Vector Multiply*. Tech. rep. UCB/CSD-03-1297. University of California, Berkeley, 2003.

[93] Srinivas Eswar et al. *Pixel Similarity*. Version 1.0. Aug. 2020.

[94] Ming Tan and Larry Eshelman. “Using Weighted Networks to Represent Classification Knowledge in Noisy Domains”. In: *Machine Learning Proceedings 1988*. Ed. by John Laird. San Francisco (CA): Morgan Kaufmann, 1988, pp. 121–134. ISBN: 978-0-934613-64-4.

[95] Hasan Metin Aktulga et al. “Improving the scalability of a symmetric iterative eigensolver for multi-core platforms”. In: *Concurrency and Computation: Practice and Experience* 26.16 (2014), pp. 2631–2651.

[96] Oguz Kaya, Ramakrishnan Kannan, and Grey Ballard. “Partitioning and communication strategies for sparse non-negative matrix factorization”. In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[97] Mark H. Van Benthem and Michael R. Keenan. “Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems”. In: *Journal of Chemometrics* 18.10 (2004), pp. 441–450. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cem.889.

[98] Dimitri P Bertsekas. “Projected Newton methods for optimization problems with simple constraints”. In: *SIAM Journal on control and Optimization* 20.2 (1982), pp. 221–246.

[99] Jiliang Tang, Xufei Wang, and Huan Liu. “Integrating social media data for community detection”. In: *Modeling and Mining Ubiquitous Social Media*. Springer, 2011, pp. 1–20.

[100] Jialu Liu et al. “Multi-view clustering via joint nonnegative matrix factorization”. In: *Proceedings of the 2013 SIAM international conference on data mining*. SIAM. 2013, pp. 252–260.

[101] Joyce Jiyoung Whang et al. “MEGA: Multi-view semi-supervised clustering of hypergraphs”. In: *Proceedings of the VLDB Endowment* 13.5 (2020), pp. 698–711.

[102] Ninghao Liu, Xiao Huang, and Xia Hu. “Accelerated local anomaly detection via resolving attributed networks”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017.

[103] Zhen Peng et al. “ANOMALOUS: A Joint Modeling Approach for Anomaly Detection on Attributed Networks.” In: *IJCAI*. 2018, pp. 3513–3519.

[104] Rundong Du. “Nonnegative matrix factorization for text, graph, and hybrid data analytics”. PhD thesis. Georgia Institute of Technology, 2018.

[105] Mark Schmidt, Dongmin Kim, and Suvrit Sra. “11 Projected Newton-type Methods in Machine Learning”. In: *Optimization for Machine Learning* 1 (2012).

[106] Ning Qian. “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1 (1999), pp. 145–151.

[107] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[108] Brett W Bader and Tamara G Kolda. “Efficient MATLAB computations with sparse and factored tensors”. In: *SIAM Journal on Scientific Computing* 30.1 (2008), pp. 205–231.

[109] Tamara G Kolda and Brett W Bader. “Tensor decompositions and applications”. In: *SIAM review* 51.3 (2009), pp. 455–500.

[110] Grey Ballard et al. “Minimizing communication in numerical linear algebra”. In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901.

[111] Grey Ballard, Koby Hayashi, and Kannan Ramakrishnan. “Parallel nonnegative CP decomposition of dense tensors”. In: *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE. 2018, pp. 22–31.

[112] Koby Hayashi et al. “Shared-memory parallelization of MTTKRP for dense tensors”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2018, pp. 393–394.

[113] Lawton Manning et al. “Parallel Hierarchical Clustering using Rank-Two Nonnegative Matrix Factorization”. In: *Proceedings of the 27th IEEE International Conference on High Performance Computing*. 2020.

[114] Pentti Paatero. “A weighted non-negative least squares algorithm for three-way PARAFAC factor analysis”. In: *Chemometrics and Intelligent Laboratory Systems* 38.2 (1997), pp. 223–242.

[115] Max Welling and Markus Weber. “Positive tensor factorization”. In: *Pattern Recognition Letters* 22.12 (2001), pp. 1255–1261.

[116] N. D. Sidiropoulos et al. “Tensor Decomposition for Signal Processing and Machine Learning”. In: *IEEE Transactions on Signal Processing* 65.13 (July 2017), pp. 3551–3582.

[117] S. Smith et al. “SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 61–70.

[118] J. Li et al. “Model-Driven Sparse CP Decomposition for Higher-Order Tensors”. In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS. May 2017, pp. 1048–1057.

[119] Oguz Kaya and Bora Uçar. “High Performance Parallel Algorithms for the Tucker Decomposition of Sparse Tensors”. In: *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*. 2016, pp. 103–112.

[120] Shaden Smith and George Karypis. “A Medium-Grained Algorithm for Distributed Sparse Tensor Factorization”. In: *IEEE 30th International Parallel and Distributed Processing Symposium*. May 2016, pp. 902–911.

[121] Oguz Kaya and Bora Uçar. “Parallel CANDECOMP/PARAFAC Decomposition of Sparse Tensors Using Dimension Trees”. In: *SIAM J. Scientific Computing* 40.1 (2018).

[122] Bing Tang et al. “GPU-accelerated Large-Scale Non-negative Matrix Factorization Using Spark”. In: *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer. 2018, pp. 189–201.

[123] Gordon E Moon et al. “PL-NMF: Parallel Locality-Optimized Non-negative Matrix Factorization”. In: *arXiv preprint arXiv:1904.07935* (2019).

[124] Israt Nisa et al. “Load-balanced sparse MTTKRP on GPUs”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 123–133.

[125] Shaden Smith, Alec Beri, and George Karypis. “Constrained tensor factorization with accelerated AO-ADMM”. In: *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE. 2017, pp. 111–120.

[126] A. P. Liavas et al. “Nesterov-based Alternating Optimization for Nonnegative Tensor Factorization: Algorithm and Parallel Implementation”. In: *IEEE Transactions on Signal Processing* (Nov. 2017).

[127] Grey Ballard, Nicholas Knight, and Kathryn Rouse. “Communication Lower Bounds for Matricized Tensor Times Khatri-Rao Product”. In: *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*. May 2018, pp. 557–567.

[128] Anh Huy Phan and Andrzej Cichocki. “PARAFAC algorithms for large-scale problems”. In: *Neurocomputing* 74.11 (2011), pp. 1970–1984.

[129] Linjian Ma and Edgar Solomonik. “Accelerating Alternating Least Squares for Tensor Decomposition by Pairwise Perturbation”. In: *arXiv preprint arXiv:1811.10573* (2018).

[130] Edgar Solomonik et al. “A massively parallel tensor contraction framework for coupled-cluster computations”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3176–3190.

[131] Anh-Huy Phan, Petr Tichavsky, and Andrzej Cichocki. “Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations”. In: *IEEE Transactions on Signal Processing* 61.19 (Oct. 2013), pp. 4834–4846.

[132] Lars Grasedyck. “Hierarchical singular value decomposition of tensors”. In: *SIAM Journal on Matrix Analysis and Applications* 31.4 (2010), pp. 2029–2054.

[133] Muthu Baskaran et al. “Efficient and scalable computations with sparse tensors”. In: *2012 IEEE Conference on High Performance Extreme Computing*. IEEE. 2012, pp. 1–6.

[134] Petros Drineas et al. “Faster least squares approximation”. In: *Numerische mathematik* 117.2 (2011), pp. 219–249.

[135] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. “Parcube: Sparse parallelizable tensor decompositions”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2012, pp. 521–536.

[136] Yining Wang et al. “Fast and guaranteed tensor decomposition via sketching”. In: *Advances in Neural Information Processing Systems*. 2015, pp. 991–999.

[137] Casey Battaglino, Grey Ballard, and Tamara G Kolda. “A practical randomized CP tensor decomposition”. In: *SIAM Journal on Matrix Analysis and Applications* 39.2 (2018), pp. 876–901.

[138] N Benjamin Erichson et al. “Randomized nonnegative matrix factorization”. In: *Pattern Recognition Letters* 104 (2018), pp. 1–7.

[139] Xiao Fu et al. “Block-randomized stochastic proximal gradient for constrained low-rank tensor factorization”. In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 7485–7489.

[140] Rasmus Bro and Claus A Andersson. “Improving the speed of multiway algorithms: Part II: Compression”. In: *Chemometrics and intelligent laboratory systems* 42.1-2 (1998), pp. 105–113.

[141] Giorgio Tomasi and Rasmus Bro. “A comparison of algorithms for fitting the PARAFAC model”. In: *Computational Statistics & Data Analysis* 50.7 (2006), pp. 1700–1734.

[142] Nico Vervliet, Otto Debals, and Lieven De Lathauwer. “Exploiting Efficient Representations in Large-Scale Tensor Decompositions”. In: *SIAM Journal on Scientific Computing* 41.2 (2019), A789–A815.

[143] Grey Ballard and Kathryn Rouse. “General Memory-Independent Lower Bound for MTTKRP”. In: *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. 2020, pp. 1–11.

[144] Oguz Kaya. “High Performance Parallel Algorithms for Tensor Decompositions”. PhD thesis. University of Lyon, Sept. 2017.

[145] Oguz Kaya and Yves Robert. “Computing Dense Tensor Decompositions with Optimal Dimension Trees”. In: *Algorithmica* 81 (2019), pp. 2092–2121.

[146] Koby Hayashi. “Parallel Algorithms for and Applications of the Dense Canonical Polyadic Decomposition”. PhD thesis. Wake Forest University, 2018.

[147] Gopinath Chennupati et al. “Distributed non-negative matrix factorization with determination of the number of latent features”. In: *The Journal of Supercomputing* 76.9 (2020), pp. 7458–7488.

[148] Tony Hyun Kim et al. “Long-Term Optical Access to an Estimated One Million Neurons in the Live Mouse Cortex”. In: *Cell Reports* 17.12 (2016), pp. 3385–3394.

[149] Haesun Park. “Nonnegativity Constrained Low Rank Approximations for Scalable Data Analytics on Distributed Memory Parallel Environment”. In: (2022). Accessed on 2022.07.14.

[150] Evrim Acar, Tamara G Kolda, and Daniel M Dunlavy. "All-at-once optimization for coupled matrix and tensor factorizations". In: *arXiv preprint arXiv:1105.3422* (2011).

[151] Alex Williams. *Solving Least-Squares Regression with Missing Data*. http://alexhwilliams.info/itsneuronalblog/2018/02/26/censored-lstsq/. 2018.

[152] K Ruben Gabriel. "Le biplot-outil d'exploration de données multidimensionnelles". In: *Journal de la société française de statistique* 143.3-4 (2002), pp. 5–55.

[153] Art B Owen, Patrick O Perry, et al. "Bi-cross-validation of the SVD and the non-negative matrix factorization". In: *The annals of applied statistics* 3.2 (2009), pp. 564–594.

[154] Alex Williams. *How to cross-validate PCA, clustering, and matrix decomposition models*. http://alexhwilliams.info/itsneuronalblog/2018/02/26/crossval/. 2018.

[155] Thierry Bouwmans et al. "Decomposition into low-rank plus additive matrices for background/foreground separation: A review for a comparative evaluation with a large-scale dataset". In: *Computer Science Review* 23 (2017), pp. 1–71.