# Distributed-Memory Parallel JointNMF


Srinivas Eswar*
seswar@anl.gov
Argonne National Laboratory
Lemont, IL, USA

Benjamin Cobb*
Koby Hayashi
bcobb33@gatech.edu
khayashi9@gatech.edu
Georgia Institute of Technology
School of Computational Science and Engineering
Atlanta, GA, USA

Ramakrishnan Kannan
kannanr@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Grey Ballard
ballard@wfu.edu
Wake Forest University
Department of Computer Science
Winston-Salem, NC, USA

Richard Vuduc
Haesun Park
richie@cc.gatech.edu
hpark@cc.gatech.edu
Georgia Institute of Technology
School of Computational Science and Engineering
Atlanta, GA, USA



## ABSTRACT

Joint Nonnegative Matrix Factorization (JointNMF) is a hybrid method for mining information from datasets that contain both feature and connection information. We propose distributed-memory parallelizations of three algorithms for solving the JointNMF problem based on Alternating Nonnegative Least Squares, Projected Gradient Descent, and Projected Gauss-Newton. We extend well-known communication-avoiding algorithms using a single processor grid case to our coupled case on two processor grids. We demonstrate the scalability of the algorithms on up to 960 cores (40 nodes) with 60% parallel efficiency. The more sophisticated Alternating Nonnegative Least Squares (ANLS) and Gauss-Newton variants outperform the first-order gradient descent method in reducing the objective on large-scale problems. We perform a topic modelling task on a large corpus of academic papers that consists of over 37 million paper abstracts and nearly a billion citation relationships, demonstrating the utility and scalability of the methods.


*Both authors contributed equally to this research.




*ICS '23, June 21–23, 2023, Orlando, FL, USA*





ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593733



## CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

## KEYWORDS

High Performance Computing, Multimodal Inputs, Nonnegative Matrix Factorization

**ACM Reference Format:**
Srinivas Eswar, Benjamin Cobb, Koby Hayashi, Ramakrishnan Kannan, Grey Ballard, Richard Vuduc, and Haesun Park. 2023. Distributed-Memory Parallel JointNMF. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577193.3593733


## 1 INTRODUCTION

This paper describes the first distributed-memory parallel algorithms for the Joint Nonnegative Matrix Factorization (JointNMF) problem [7]. JointNMF has been proposed for unsupervised learning and data mining tasks on *attributed graph datasets*, where data points naturally have both nonnegative features *and* connectivity (relational) information expressed by a graph [10]. This form appears in many datasets, including those arising in the analysis of social networks, document corpora, gene regulatory networks, and image datasets, among others. For example, consider a corpus of scholarly documents that one wishes to cluster. The document features might be represented by a term-document matrix with entries derived from Term Frequency-Inverse Document Frequency (tf-idf) scores, which are nonnegative [19]; and the citations among documents would form the edges of a graph. One might want both the features and the graph to inform the clustering, which JointNMF accomplishes. However, it has been difficult to apply these algorithms at more than a modest scale due to their high computational and memory costs, thus motivating our work.

The simplest form of JointNMF is as follows. (Section 3 generalizes this formulation, which our new algorithms and software

can handle via straightforward preprocessing.) Let $\mathbf{X} \in \mathbb{R}_+^{m\times n}$ be the nonnegative features matrix and $\mathbf{S} \in \mathbb{R}_+^{n\times n}$ a symmetric nonnegative connections matrix with $\mathbf{S} = \mathbf{S}^\mathsf{T}$. JointNMF is a Constrained Low Rank Approximation (CLRA) method and a natural extension of the popular data mining methods, Nonnegative Matrix Factorization (NMF) and Symmetric Nonnegative Matrix Factorization (SymNMF), to handle this type of multimodal form of input [7, 16, 18]. It formulates the optimization problem as

$$\min_{\mathbf{W}\geq 0,\mathbf{H}\geq 0} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2 + \alpha \left\|\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2, \tag{1}$$

where $\mathbf{W} \in \mathbb{R}_+^{m\times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k\times n}$ are low-rank matrices to be found with $k \ll \min(m, n)$. The hyperparameter $\alpha \geq 0$ can be tuned to emphasize which objective, either the features fit or connections fit, is of more importance to the user. Fusing the two information sources in the objective permits the use of the CLRA machinery developed for NMF and SymNMF and makes the results interpretable without any additional preprocessing or clustering steps [8, 9].

The main sequential algorithms for JointNMF use Block Coordinate Descent (BCD) [7]. They include an Alternating Nonnegative Least Squares (ANLS) type algorithm for Eq. (1) using a regularized approach similar to the one developed for SymNMF [18]. However, direct optimization techniques based on Newton-like methods are believed to have prohibitive computational and memory costs.

We investigate new parallel algorithms for distributed-memory systems for both BCD and direct methods, namely, the ANLS variant of Du et al. [7] and two parallel direct methods Projected Gradient Descent (PGD) and Projected Gauss-Newton using Conjugate Gradient (PGNCG) (Section 3). Unlike the inexact Gauss-Newton method developed for SymNMF [9], PGNCG has stronger guarantees for convergence while still enjoying low computational and memory requirements. Our methods facilitate reuse of existing algorithms and software developed for the standard NMF case. We adopt the communication-optimal ANLS method due to Kannan et al. and extend it to handle multimodal inputs [14].

We evaluate these methods experimentally (Section 4). Our serial experiments highlight the superior performance of the ANLS and PGNCG methods compared to PGD. The parallelization exercise highlights the different computational challenges that arise when there are two large input matrices involved in the bottleneck matrix multiplications. We explore the different processor grid layouts, optimized for multiplication with $\mathbf{X}$, with $\mathbf{S}$, or both, needed to handle this case. With appropriate choices, we observe over 60% parallel efficiency on up to 960 cores distributed across 40 nodes.

These novel JointNMF methods permit scaling to problem sizes that were previously computationally infeasible. For instance, we perform topic discovery using JointNMF on a large corpus of over 37 million academic paper abstracts and 0.9 billion citation relationships for the first time [30, 32]. The PGNCG method achieved a speedup of 28× when scaling to 48 cores on this sparse dataset with a memory footprint of 59 GB. This initial demonstration paves the way for new applications of JointNMF, including as a potentially simpler and faster alternative to tensor-based methods for unsupervised learning and data mining.

**Table 1: Costs of certain MPI collectives.**

| Operation | Cost |
|---|---|
| All-Gather | $\nu \cdot \log p + \phi \cdot \frac{p-1}{p} n$ |
| Reduce-Scatter | $\nu \cdot \log p + (\phi + \omega) \cdot \frac{p-1}{p} n$ |
| All-Reduce | $2\nu \cdot \log p + (2\phi + \omega) \cdot \frac{p-1}{p} n$ |

# 2 PRELIMINARIES

## 2.1 Notation

We use bold lowercase fonts for vectors (e.g., $\mathbf{x}$) and bold uppercase for matrices (e.g., $\mathbf{A}$). For $\mathbf{A}$, its $i$th column is $\mathbf{a}_i$. $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ are also used for denoting the $i$th row and $j$th column of $\mathbf{A}$, respectively. Elements of $\mathbf{A}$ are interchangeably denoted by $a_{ij}$ and $\mathbf{A}(i,j)$. A vector $\mathbf{x}$ or matrix $\mathbf{X}$ that changes during an iterative algorithm will be represented as $\mathbf{x}^{(t)}$ and $\mathbf{X}^{(t)}$ at iteration $t$. We use logical indexing 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. The comparison $\mathbf{X} \geq 0$ is performed element-wise and $[\mathbf{X}]_+$ for projection on to the nonnegative orthant. The vec($\mathbf{X}$) operator vectorizes 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

$$(\mathbf{A} \otimes \mathbf{B})_{i,j} = a_{\lceil i/p\rceil,\lceil j/q\rceil} b_{(i-1)\%p+1,(j-1)\%q+1}.$$

$\mathbf{P}_{p,q}$ denotes a commutation matrix that converts a column-major vectorization of $\mathbf{A} \in \mathbb{R}^{p\times q}$ to row-major, that is

$$\mathbf{P}_{p,q}\text{vec}(\mathbf{A}) = \text{vec}\left(\mathbf{A}^\mathsf{T}\right).$$

The distributed-memory algorithms we consider here utilize the All-Gather, Reduce-Scatter, and All-Reduce Message Passing Interface (MPI) Collectives [4], whose costs are summarized in Table 1. The costs are modelled using the MPI model with a fixed cost of $\omega$ per flop and $\nu + \phi n$ for sending $n$ words between two processors, where $\nu$ is the per-message latency cost and $\phi$ is the per-word bandwidth cost.

Throughout the paper we assume that the $p$ available processors are logically arranged in a $p = p_r \times p_c$ grid $\Pi$. Global distributed matrices are shown with the usual bold font, like $\mathbf{X}$ and $\mathbf{W}$, with local matrices shown with the processor subscripts (e.g. $\mathbf{X}_{ij}$ or $\mathbf{W}_r$) The input data matrices, $\mathbf{X}$ and $\mathbf{S}$, are 2D distributed across the grid with $\mathbf{W}$ and $\mathbf{H}$ in conformal 1D distributions (see Fig. 1). Other matrix distributions are mentioned when required. In Section 3, we consider the case of using two logical grids of $p$ processors, $\Pi = p_r \times p_c$ and $\Gamma = q_r \times q_c$.

## 2.2 BCD for Nonlinear Optimization

BCD is an iterative process to solve an optimization problem. It groups variables into several disjoint subgroups and iteratively solves the variables in a subgroup keeping the others fixed. This grouping helps when subproblems can be solved quickly or exactly.

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

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

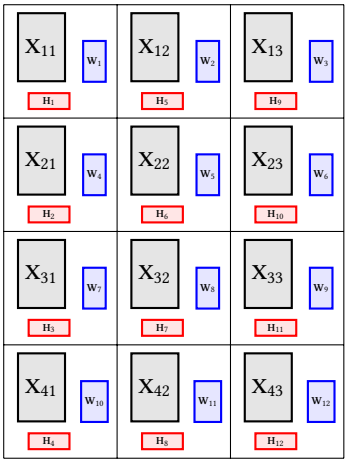



**Figure 1: Data distribution of the NMF matrices on a $4 \times 3$ processor grid. The $m \times n$ input matrix X is 2D partitioned, whereas the factor matrices are 1D partitioned. The rows of W are placed in row-major order and the columns of H in column-major order across the grid. The smallest dimension $k$ is never split. The local portions of the matrices have the following sizes: $\mathbf{X}_{ij}$ is $\frac{m}{p_r} \times \frac{n}{p_c}$, $\mathbf{W}_i$ is $\frac{m}{p} \times k$, and $\mathbf{H}_j$ is $k \times \frac{n}{p}$.**

We assume that $C \subseteq \mathbb{R}^n$ can be represented as the Cartesian product, $C = C_1 \times C_2 \times \dots C_m$ , where $C_i \subseteq \mathbb{R}^{n_i}$ is closed and convex with $\sum_{i=1}^{m} n_i = n$. The input vector is similarly partitioned as $\mathbf{x} = \begin{bmatrix}\mathbf{x}_1^\mathsf{T} & \mathbf{x}_2^\mathsf{T} \cdots & \mathbf{x}_m^\mathsf{T}\end{bmatrix}^\mathsf{T}$. The BCD method generates the next iterate $\mathbf{x}^{(t+1)}$ given the current iterate $\mathbf{x}^{(t)}$ according to the update rule

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

This scheme uses the most recently updated values of all the fixed blocks and ensures that the objective value never increases. The convergence of BCD is discussed elsewhere [3, 12, 16].

## 2.3 Direct Optimization Overview

We use the Projected Gradient Descent and an inexact Gauss-Newton (GN) methods for the optimization, $\min_{\mathbf{x} \geq 0} f(\mathbf{x})$. For PGD, we use the momentum variant, which accelerates gradient descent and dampens oscillations [26, 27]. Let $\mathbf{p}^{(t)}$ be the step taken by the algorithm at iteration $t$. PGD's updates take the form,

$$\mathbf{p}^{(t)} = \gamma \mathbf{p}^{(t-1)} + \frac{\nabla_{\mathbf{x}} f}{\|\nabla_{\mathbf{x}} f\|} ,$$
$$\mathbf{x}^{(t)} = \left[\mathbf{x}^{(t-1)} - \lambda \mathbf{p}^{(t)}\right]_+ .$$

Here, $\gamma \geq 0$ is the momentum parameter and $\lambda \geq 0$ is the step size, which is found via experiment. $\mathbf{p}^{(0)}$ is initialized as $\mathbf{0}$.

The GN method minimizes a sum of squares of residual functions of the form $f(\mathbf{x}) = \frac{1}{2} \sum_l r_l(\mathbf{x})^2$. It starts with an initial guess $\mathbf{x}^{(0)}$ and follows the iteration:

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

Here $\mathbf{J}^{(t)}$ is the Jacobian matrix defined as $\mathbf{J}_{lq}^{(t)} = \frac{\partial r_l}{\partial x_q}(\mathbf{x}^{(t)})$ 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 optimization method where the matrix $\mathbf{J}^\mathsf{T}\mathbf{J}$ acts as an approximate Hessian matrix for $f$ [3]. We can use an iterative method, like Conjugate Gradient (CG), to efficiently solve for $\mathbf{p}$ by only applying $\mathbf{J}$ and $\mathbf{J}^\mathsf{T}$ such that,

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

The right hand side of the equation is the gradient evaluated at $\mathbf{x}^{(t)}$. If the least-squares problem for $\mathbf{p}$ is only approximately solved, the method is known as an *inexact* GN algorithm.

To converge in the constrained case, i.e., $\mathbf{x} \geq 0$, we need to modify $\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}$ to handle variables whose constraints are *active* [2]. Denote these active variables by

$$\mathcal{A} = \left\{ i : 0 \leq x_i^{(t)} \leq \epsilon \text{ and } \frac{\partial f}{\partial x_i^{(t)}} > 0 \right\},$$

where $\epsilon$ is a small constant. The complement of $\mathcal{A}$ is the set of free variables $\mathcal{F} = \{1, \dots, n\} \setminus \mathcal{A}$. Without loss of generality, assume $\mathbf{x}^{(t)}$ is permuted like $\begin{bmatrix}\mathbf{x}_{\mathcal{F}}^{(t)} \\ \mathbf{x}_{\mathcal{A}}^{(t)}\end{bmatrix}$. The step direction solves

$$\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)_{\mathcal{F}}\mathbf{p}_{\mathcal{F}} = \mathbf{g}_{\mathcal{F}}^{(t)} \text{ and } \mathbf{p}_{\mathcal{A}} = \mathbf{g}_{\mathcal{A}}^{(t)} ,$$

where $\left(\mathbf{J}^{(t)\mathsf{T}}\mathbf{J}^{(t)}\right)_{\mathcal{F}}$ is the approximate Hessian whose rows and columns are restricted to the ones corresponding to the free variables. Second-order information is captured for the free variables whereas the active ones are kept fixed at the constraints. Bertsekas shows that these scaling matrices result in $\mathbf{p}$ being a descent direction, and with an appropriate step size $\lambda$, such an iterative procedure converges to a stationary point [2].

We use a variant of backtracking line search to step in a direction that reduces the objective for both PGD and PGNCG [3, 29].

## 2.4 Parallel NMF Algorithms

Our implementation of JointNMF is built on top of Parallel Low-rank Approximation with Nonnegativity Constraints (PLANC), an open-source library for NMF [8, 14]. PLANC is designed to solve the optimization problem

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

for dense or sparse, nonnegative input matrices $\mathbf{X} \in \mathbb{R}_+^{m \times n}$ and low-rank matrices $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ with $k \ll \min(m, n)$. It contains various algorithms for computing NMF, including both BCD and direct optimization variants.

**Algorithm 1** Gram Computation

**Require:** $\Pi$ a $p_r \times p_c$ processor grid.
**Require:** $\mathbf{B} \in \mathbb{R}_+^{p \times k}$ is row-wise distributed across the grid.
**Ensure:** $\mathbf{B}^\mathsf{T}\mathbf{B}$ is stored redundantly on all processors.
1: **function** $[\mathbf{G}]$ = GRAM($\Pi, \mathbf{B}$)
2: $(i, j)$ = MyRank($\Pi$) ▷ Processor rank
3: $r = (i-1)\,p_c + j$ ▷ Row-major rank
4: $\hat{\mathbf{G}}_r = \mathbf{B}_r^\mathsf{T}\mathbf{B}_r$
5: $\mathbf{G}$ = All-Reduce($\hat{\mathbf{G}}_r$)
6: **end function**

**Algorithm 2** 2D Matrix Multiplication

**Require:** $\Pi$ a $p_r \times p_c$ processor grid.
**Require:** $\mathbf{A} \in \mathbb{R}_+^{p \times q}$ is 2D distributed across the grid.
**Require:** $\mathbf{B} \in \mathbb{R}_+^{p \times k}$ is row-wise distributed across the grid.
**Ensure:** $\mathbf{B}^\mathsf{T}\mathbf{A}$ is column-distributed across the grid.
1: **function** $[\mathbf{C}]$ = 2Dmatmul($\Pi, \mathbf{A}, \mathbf{B}$)
2: $(i, j)$ = MyRank($\Pi$) ▷ Processor rank
3: $r = (i-1)\,p_c + j$ ▷ Row-major rank
4: $c = (j-1)\,p_r + i$ ▷ Col-major rank
5: $\hat{\mathbf{B}}_i$ = All-Gather($\mathbf{B}_r, \Pi(i, :)$)
6: $\hat{\mathbf{C}}_j = \hat{\mathbf{B}}_i^\mathsf{T}\mathbf{A}_{ij}$
7: $\mathbf{C}_c$ = Reduce-Scatter($\hat{\mathbf{C}}_j, \Pi(:, j)$) ▷ Column-wise distributed
8: **end function**

Matrix multiplication is the computational bottleneck of NMF with most algorithms needing to compute $\mathbf{W}^\mathsf{T}\mathbf{W}$, $\mathbf{H}\mathbf{H}^\mathsf{T}$, $\mathbf{W}^\mathsf{T}\mathbf{X}$, and $\mathbf{H}\mathbf{X}^\mathsf{T}$. PLANC uses communication-optimal matrix multiplication algorithms for each of these operations [1, 5], via a 2D distribution of $\mathbf{X}$ over a $p_r \times p_c$ grid of processors with conformal 1D distributions for the factor matrices $\mathbf{W}$ and $\mathbf{H}$ as shown in Fig. 1. A 2D algorithm is used to compute $\mathbf{W}^\mathsf{T}\mathbf{X}$ and $\mathbf{H}\mathbf{X}^\mathsf{T}$. PLANC does not communicate any of the data matrix elements. In the case of dense $\mathbf{X}$, Kannan et al. show that selecting $\frac{p_r}{p_c} \approx \frac{m}{n}$ minimizes the number of words transferred in the multiplications [13].

The full two-block BCD algorithm for NMF is shown in Algorithm 3. In each outer iteration, Line 6 in Algorithm 3, we alternately fix $\mathbf{H}$ and update $\mathbf{W}$ using a Nonnegative Least Squares (NLS) solver (Line 9) and then solve for $\mathbf{H}$ fixing $\mathbf{W}$ (Line 12). The pseudocode for the two different types of matrix multiplication are shown in Algorithm 2 and Algorithm 1. Algorithm 1 computes the Gram matrices $\mathbf{W}^\mathsf{T}\mathbf{W}$ and $\mathbf{H}\mathbf{H}^\mathsf{T}$, whereas Algorithm 2 involves multiplications with the large data matrix $\mathbf{X}$. Notice that the NLS updates, Lines 9 and 12, are local and do not involve any communication.

## 2.5 Additional Related Work

The fusion of multiple information sources is common in CLRA. Various formulations of fusing multiple features and connections matrices with different constraints exist for clustering [7, 21, 31, 34] and anomaly detection [10, 22, 25]. 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, however Whang et al. [34] has also extended this approach to hypergraphs and to the semi-supervised case. For more information on the different formulations, we refer the reader elsewhere [6].

**Algorithm 3** Communication-avoiding parallel NMF [13]

**Ensure:** $\mathbf{W}, \mathbf{H} \approx \underset{\mathbf{W}\geq 0, \mathbf{H}\geq 0}{\operatorname{argmin}} \|\mathbf{X} - \mathbf{W}\mathbf{H}\|_F^2$.
1: **function** $[\mathbf{W}, \mathbf{H}]$ = ParNMF($\Pi, \mathbf{X}, k$)
2: $(i, j)$ = myrank($\Pi$) ▷ Processor rank
3: $r = (i-1)\,p_c + j$ ▷ Row-major rank
4: $c = (j-1)\,p_r + i$ ▷ Column-major rank
5: Initialise $\mathbf{H}_c^{(0)}$
6: **while** $t = 1, 2, \ldots$ **do** ▷ Till some stopping condition is met.
**% Compute W given H**
7: $\mathbf{G}_\mathbf{H}$ = gram $\left(\Pi^\mathsf{T}, \left(\mathbf{H}^{(t-1)}\right)^\mathsf{T}\right)$
8: $\mathbf{R}_\mathbf{H}$ = 2Dmatmul $\left(\Pi^\mathsf{T}, \mathbf{X}^\mathsf{T}, \left(\mathbf{H}^{(t-1)}\right)^\mathsf{T}\right)$
9: $\mathbf{W}_r^{(t)}$ = update $\left(\mathbf{G}_\mathbf{H}, \mathbf{R}_{\mathbf{H}_r}\right)$ ▷ NLS update
**% Compute H given W**
10: $\mathbf{G}_\mathbf{W}$ = gram $\left(\Pi, \mathbf{W}^{(t)}\right)$
11: $\mathbf{R}_\mathbf{W}$ = 2Dmatmul $\left(\Pi, \mathbf{X}, \mathbf{W}^{(t)}\right)$
12: $\mathbf{H}_c^{(t)}$ = update $\left(\mathbf{G}_\mathbf{W}, \mathbf{R}_{\mathbf{W}_c}\right)$ ▷ NLS update
13: **end while**
14: **end function**

The PGD and projected Gauss-Newton methods are staples of modern optimization [3, 23] and have been used for CLRA [20, 29, 33]. In this context, we are primarily focused on the momentum variant of PGD [26, 27] and the box-constrained version of Gauss-Newton [2]. While these methods have been developed for general optimization, our approach is the first known treatment of these methods for JointNMF in the distributed-memory setting.

# 3 ALGORITHMS FOR JOINTNMF

## 3.1 Extending to Multiple Inputs

The most common form of JointNMF is the optimization problem involving a single features and connections matrix, respectively denoted $\mathbf{X}$ and $\mathbf{S}$, as shown in Eq. (1). This formulation can be extended to more than two inputs as follows:

$$\min_{\{\mathbf{W}_1,\ldots,\mathbf{W}_p,\mathbf{H}\}\geq 0} \sum_{i=1}^{p} \gamma_i \|\mathbf{X}_i - \mathbf{W}_i\mathbf{H}\|_F^2 + \sum_{j=1}^{q} \alpha_j \left\|\mathbf{S}_j - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2 \quad (3)$$

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. (3) and Eq. (1). In the features term, we can combine the $p$ different terms as follows:

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

Thus, optimizing with $\hat{\mathbf{X}}$ is the same.

The same trick for the connection matrices is not as straightforward. That can be seen by induction, working first with two matrices, $\mathbf{S}_1$ and $\mathbf{S}_2$. 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 $\hat{\alpha}\left\|\hat{\mathbf{S}} - \mathbf{Y}\right\|_F^2 - \sum_{j=1}^2 \alpha_j \left\|\mathbf{S}_j - \mathbf{Y}\right\|_F^2$. The $(i, j)$ entry of this difference can be shown to be equal to $-\alpha_1\alpha_2 \left(\mathbf{S}_1(i,j) - \mathbf{S}_2(i,j)\right)^2$. 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. (1) will result in the same solution as using $\mathbf{S}_1$ and $\mathbf{S}_2$ in Eq. (3). 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. (1), is versatile enough to handle multiple information sources via simple preprocessing, so we can focus on this case for parallelization.

## 3.2 JointNMF via ANLS

Du et al. solve Eq. (1) by dropping the symmetric constraint and using a penalty term [7]. They propose the surrogate optimization,

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

where $\hat{\mathbf{H}} \in \mathbb{R}_+^{k\times n}$ and $\beta \geq 0$ is the regularization parameter. This formulation is motivated by a similar one for the SymNMF problem [9, 17]. It 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.

$$\min_{\mathbf{W}\geq 0} \left\|\mathbf{H}^\mathsf{T}\mathbf{W}^\mathsf{T} - \mathbf{X}^\mathsf{T}\right\|_F^2 \tag{4}$$

$$\min_{\hat{\mathbf{H}}\geq 0} \left\|\begin{bmatrix}\sqrt{\alpha}\mathbf{H}^\mathsf{T} \\ \sqrt{\beta}\mathbf{I}_k\end{bmatrix}\hat{\mathbf{H}} - \begin{bmatrix}\sqrt{\alpha}\mathbf{S} \\ \sqrt{\beta}\mathbf{H}\end{bmatrix}\right\|_F^2 \tag{5}$$

$$\min_{\mathbf{H}\geq 0} \left\|\begin{bmatrix}\mathbf{W} \\ \sqrt{\alpha}\hat{\mathbf{H}}^\mathsf{T} \\ \sqrt{\beta}\mathbf{I}_k\end{bmatrix}\mathbf{H} - \begin{bmatrix}\mathbf{X} \\ \sqrt{\alpha}\mathbf{S} \\ \sqrt{\beta}\hat{\mathbf{H}}\end{bmatrix}\right\|_F^2 \tag{6}$$

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 $\mathbf{H}\mathbf{X}^\mathsf{T}$, $\alpha\mathbf{H}\mathbf{S} + \beta\mathbf{H}$, $\mathbf{W}^\mathsf{T}\mathbf{X} + \alpha\hat{\mathbf{H}}\mathbf{S} + \beta\hat{\mathbf{H}}$. We use the Block Principal Pivoting algorithm as the NLS solver [15]. The computational cost of solving $\ell$ NLS problems, i.e. the multiple right hand sides, in $r$ variables is $O\left(\ell r^3 s_{\text{AS}}\right)$, where $s_{\text{AS}} \leq 2^r$ is the number of active sets explored [12]. In practice however, $s_{\text{AS}}$ is typically much smaller than $2^r$.

## 3.3 JointNMF via PGD

PGD is a straightforward way to address Eq. (1). The gradients with respect to the factor matrices are

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

**Algorithm 4** Compute Gradient

**Require:** $\mathbf{C} \in \mathbb{R}_+^{k\times n}$ is a copy of $\mathbf{H}$ column-wise distributed across the grid $\Gamma$.
**Ensure:** Gradient $\nabla_\mathbf{W} f = 2(\mathbf{W}\mathbf{H}\mathbf{H}^\mathsf{T} - \mathbf{X}\mathbf{H}^\mathsf{T}) \in \mathbb{R}^{m\times k}$ row-wise distributed across the grid $\Pi$.
**Ensure:** Gradient $\nabla_\mathbf{H} f = 2(\mathbf{W}^\mathsf{T}\mathbf{W}\mathbf{H} - \mathbf{W}^\mathsf{T}\mathbf{X}) + 4\alpha(\mathbf{H}\mathbf{H}^\mathsf{T}\mathbf{H} - \mathbf{H}\mathbf{S}) \in \mathbb{R}^{k\times n}$ column-wise distributed across the grid $\Pi$.

**function** $[\nabla_\mathbf{W}, \nabla_\mathbf{H}] =$ ComputeGradient$(\mathbf{X}, \Pi, \mathbf{S}, \Gamma, \mathbf{W}, \mathbf{H})$
  $(i, j) =$ myrank$(\Pi)$ ▷ Processor rank
  $r = (i-1)\,p_c + j$ ▷ Row-major rank
  $c = (j-1)\,p_r + i$ ▷ Column-major rank
  **% Compute $\nabla_\mathbf{W}$**
  $\mathbf{G}_\mathbf{H} =$ gram$(\Pi^\mathsf{T}, \mathbf{H}^\mathsf{T})$ ▷ Gram redundantly stored on $\Pi$
  $\mathbf{L}_{\mathbf{W}r} = \mathbf{W}_r\mathbf{G}_\mathbf{H}$
  $\mathbf{R}_\mathbf{W} =$ 2Dmatmul$(\Pi^\mathsf{T}, \mathbf{X}, \mathbf{H}^\mathsf{T})$ ▷ $\mathbf{H}$ row-wise on $\Pi^\mathsf{T}$
  $\nabla_{\mathbf{W}r} = 2(\mathbf{L}_{\mathbf{W}r} - \mathbf{R}_{\mathbf{W}r})$
  **% Compute $\nabla_\mathbf{H}$**
  $\mathbf{G}_\mathbf{W} =$ gram$(\Pi, \mathbf{W})$ ▷ Gram redundantly stored on $\Pi$
  $\mathbf{L}_{\mathbf{H}c} = (2\mathbf{G}_\mathbf{W} + 4\alpha\mathbf{G}_\mathbf{H})\,\mathbf{H}_c$
  $\mathbf{R}_{\mathbf{H}\mathbf{X}} =$ 2Dmatmul$(\Pi, \mathbf{X}, \mathbf{W})$
  $\mathbf{R}_{\mathbf{H}\mathbf{S}} =$ 2Dmatmul$(\Gamma^\mathsf{T}, \mathbf{S}^\mathsf{T}, \mathbf{C}^\mathsf{T})$ ▷ $\mathbf{C}$ is the copy of $\mathbf{H}$ on $\Gamma$
  $\mathbf{R}_{\mathbf{H}\mathbf{S}} =$ redist$\left(\mathbf{R}_{\mathbf{H}\mathbf{S}}, \Gamma, \Pi^\mathsf{T}\right)$ ▷ Send from grid $\Gamma$ to $\Pi^\mathsf{T}$
  $\nabla_{\mathbf{H}c} = (2\mathbf{R}_{\mathbf{H}\mathbf{X}c} + 4\alpha\mathbf{R}_{\mathbf{H}\mathbf{S}c}) - \mathbf{L}_{\mathbf{H}c}$
**end function**

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

$$\mathbf{P}_\mathbf{W}^{(t)} = \gamma\mathbf{P}_\mathbf{W}^{(t-1)} + \frac{\nabla_\mathbf{W} f}{\|\nabla_\mathbf{W} f\|_F}$$
$$\mathbf{W}^{(t)} = \left[\mathbf{W}^{(t-1)} - \lambda\mathbf{P}_\mathbf{W}^{(t)}\right]_+$$
$$\mathbf{P}_\mathbf{H}^{(t)} = \gamma\mathbf{P}_\mathbf{H}^{(t-1)} + \frac{\nabla_\mathbf{H} f}{\|\nabla_\mathbf{H} f\|_F}$$
$$\mathbf{H}^{(t)} = \left[\mathbf{H}^{(t-1)} - \lambda\mathbf{P}_\mathbf{H}^{(t)}\right]_+ .$$

As in the ANLS case, the computational bottlenecks are a subset of matrix multiplications, $\mathbf{W}^\mathsf{T}\mathbf{W}$, $\mathbf{H}\mathbf{H}^\mathsf{T}$, $\mathbf{W}^\mathsf{T}\mathbf{X}$, $\mathbf{X}\mathbf{H}^\mathsf{T}$, and $\mathbf{H}\mathbf{S}$.

## 3.4 JointNMF via PGNCG

To apply PGNCG to the JointNMF objective in Eq. (1), consider the vectorized residuals,

$$\mathbf{r} = \begin{bmatrix}\mathbf{r}_\mathbf{X} \\ \mathbf{r}_\mathbf{S}\end{bmatrix} = \begin{bmatrix}\text{vec}(\mathbf{X} - \mathbf{W}\mathbf{H}) \\ \sqrt{\alpha}\text{vec}\left(\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\right)\end{bmatrix} \in \mathbb{R}^{mn+n^2} .$$

The Jacobian for Eq. (1) is a $2\times 2$ block matrix of the form

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

We can easily verify that $2\mathbf{J}^\mathsf{T}\mathbf{r}$ gives the gradient. Applying $\mathbf{J}^\mathsf{T}\mathbf{J}$ to a vector $\mathbf{x} = \begin{bmatrix}\text{vec}(\mathbf{X}_\mathbf{W}) \\ \text{vec}(\mathbf{X}_\mathbf{H})\end{bmatrix}$ results in $\mathbf{y} = \begin{bmatrix}\text{vec}(\mathbf{Y}_\mathbf{W}) \\ \text{vec}(\mathbf{Y}_\mathbf{H})\end{bmatrix}$. Exploiting the

structure of $\mathbf{J}$ we get

$$\mathbf{Y_W} = \mathbf{X_W H H^T} + \mathbf{W X_H H^T}$$
$$\mathbf{Y_H} = \left(\mathbf{W^T X_W H} + \mathbf{W^T W H}\right) + 2\alpha\left(\mathbf{H H^T X_H} + \mathbf{H X_H^T H}\right).$$

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.

We drop the extra constant and work with $\mathbf{J^T r}$ and $\mathbf{J^T J}$ for the PGNCG step. 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.

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

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 $\mathbf{H}(i,j) = 0$, a small fixed scalar $\epsilon$ prevents zigzagging of the solution [3]. To compute the correct step, before starting the CG iterations we mask the gradients as $\nabla_\mathbf{W} f(\mathcal{A}_\mathbf{W}) = 0$ and $\nabla_\mathbf{H} f(\mathcal{A}_\mathbf{H}) = 0$. When applying $\mathbf{J^T J}$, we mask the outputs $\mathbf{Y_W}(\mathcal{A}_\mathbf{W}) = 0$ and $\mathbf{Y_H}(\mathcal{A}_\mathbf{H}) = 0$. Finally, the step directions of the active variables are set to the gradient [2, 29, 33]. We limit the inner CG solver to a maximum of $s_{\text{CG}}$ iterations.

**Algorithm 5** Apply approximate Hessian

**Require:** $\mathbf{X_W} \in \mathbb{R}_+^{m\times k}$ row-wise distributed across the grid $\Pi$.
**Require:** $\mathbf{X_H} \in \mathbb{R}_+^{k\times n}$ column-wise distributed across the grid $\Pi$.
**Ensure:** $\mathbf{Y_W} = \mathbf{X_W H H^T} + \mathbf{W X_H H^T} \in \mathbb{R}^{m\times k}$ row-wise distributed across the grid $\Pi$.
**Ensure:** $\mathbf{Y_H} = \mathbf{W^T X_W H} + \mathbf{W^T W H} + 2\alpha\left(\mathbf{H H^T X_H} + \mathbf{H X_H^T H}\right) \in \mathbb{R}^{k\times n}$ column-wise distributed across the grid $\Pi$.

1: **function** $[\mathbf{Y_W}, \mathbf{Y_H}]$ = ApplyHessian($\mathbf{X}, \Pi, \mathbf{S}, \Gamma, \mathbf{W}, \mathbf{H}$)
2: $(i,j)$ = myrank($\Pi$) ▷ Processor rank
3: $r = (i-1)p_c + j$ ▷ Row-major rank
4: $c = (j-1)p_r + i$ ▷ Column-major rank
**% Compute $\mathbf{Y_W}$**
5: $\mathbf{G_H}$ = gram($\Pi^\mathsf{T}, \mathbf{H}^\mathsf{T}$) ▷ Redundantly stored
6: $\mathbf{G_{XH}}$ = innerprod($\Pi^\mathsf{T}, \mathbf{X}_\mathbf{H}^\mathsf{T}, \mathbf{H}^\mathsf{T}$) ▷ Redundantly stored
7: $\mathbf{Y_{W}}_r = \mathbf{X_W}_r \mathbf{G_H} + \mathbf{W}_r \mathbf{G_{XH}}$
8: maskactiveset($\mathbf{Y_W}_r$)
**% Compute $\mathbf{Y_H}$**
9: $\mathbf{G_W}$ = gram($\Pi, \mathbf{W}$) ▷ Redundantly stored
10: $\mathbf{G_{WX}}$ = innerprod($\Pi, \mathbf{W}, \mathbf{X_W}$) ▷ Redundantly stored
11: $\mathbf{Y_H}_c = \mathbf{G_{WX}}\mathbf{H}_c + \mathbf{G_W}\mathbf{H}_c + 2\alpha(\mathbf{G_H}\mathbf{X_H}_c + \mathbf{G_{XH}}\mathbf{H}_c)$
12: maskactiveset($\mathbf{Y_H}_c$)
13: **end function**

We compare the costs of the different JointNMF algorithms in Table 2 for dense inputs $\mathbf{X}$ and $\mathbf{S}$. The only difference in the sparse case is the computation costs of multiplication with the input matrices (e.g. $\mathbf{W^T X}$ will cost $2\text{nnz}(\mathbf{X})k$ instead of $2mnk$). The communication costs remain the same as only dense matrices are transmitted.

## 3.5 Changes from NMF and SymNMF

Differences in parallelization strategies for JointNMF arise from having two large input matrices. The communication costs for the four multiplications are given below, assuming we are working with a $p_r \times p_c$ grid of $p$ processors.[1]

$$T_{\text{comm}}\left(\mathbf{W^T X}\right) = 2\nu \log p + \phi\left(\frac{mk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right)$$
$$T_{\text{comm}}\left(\mathbf{H X^T}\right) = 2\nu \log p + \phi\left(\frac{mk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right)$$
$$T_{\text{comm}}\left(\hat{\mathbf{H}}^\mathsf{T}\mathbf{S}\right) = 2\nu \log p + \phi\left(\frac{nk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right)$$
$$T_{\text{comm}}\left(\mathbf{HS}\right) = 2\nu \log p + \phi\left(\frac{nk}{p}(p_c - 1) + \frac{nk}{p}(p_r - 1)\right)$$

The total words communicated is $\frac{(2m+2n)k}{p}(p_c - 1) + \frac{4nk}{p}(p_r - 1)$, 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, and for the PGD and PGNCG algorithms, $\frac{p_r}{p_c} \approx \frac{2m+n}{3n}$.

An alternative is 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 $\mathbf{S}$. 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.

We analyze the differences in the bandwidth term for the single and double grid configuration for the ANLS algorithm. Using the optimal grid configuration, $\frac{p_r}{p_c} = \frac{m+n}{2n}$, we have $p_r = \sqrt{\frac{m+n}{2n}}\sqrt{p}$ and $p_c = \sqrt{\frac{2n}{m+n}}\sqrt{p}$. Plugging these into the bandwidth component of the communication costs, we have

$$W_{1\text{G}} = \frac{4nk-4}{p}(p_c - 1) + \frac{2mk+2nk-4}{p}(p_r - 1) > \frac{4\sqrt{2}k}{\sqrt{p}}\sqrt{mn+n^2}.$$

Similarly, we can get the words communicated in the two grids case as $W_{2\text{G}} \approx \frac{4k}{\sqrt{p}}\left(\sqrt{mn} + n\right)$. Now using the 1-norm inequality in $d$ dimensions, $\|\cdot\|_2 \le \|\cdot\|_1 \le \sqrt{d}\,\|\cdot\|_2$, we get $1 \le \frac{W_{1\text{G}}}{W_{2\text{G}}} \le \sqrt{2}$. Thus the two-grid configuration always communicates less data with a maximum savings of 41% in terms of words communicated.

Similar analysis of the PGD and PGNCG algorithms, which only employ three large matrix multiplications, yields the ratio

$$\frac{W_{1\text{G}}}{W_{2\text{G}}} = \sqrt{3}\frac{\sqrt{2mn+n^2}}{2\sqrt{mn}+n}.$$

The minimum value for this ratio is 1, obtained when $m = n$. Hence, the two-grid setting always communicates less data, and we will see minimal gains when $m \approx n$. However, when $n \gg m$ we can expect an improvement factor up to $\sqrt{3}$, and $\sqrt{\frac{3}{2}}$ in the $m \gg n$ scenario.

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

**Table 2: Per-iteration costs for the different JointNMF algorithms for $m \times n$ and $n \times n$ data matrices and rank $k$.**

| Algorithm (Hyperparameters) | Large Matmuls | Messages | Words | Computation |
|---|---|---|---|---|
| ANLS ($\beta$) | 4 | $O(\log p)$ | $O\left(k^2 + (k\sqrt{mn} + nk)/\sqrt{p}\right)$ | $O\left(\left(4mnk + 4n^2k + s_{\text{AS}}(m+n)k^3\right)/p\right)$ |
| PGD ($\gamma$) | 3 | $O(\log p)$ | $O\left(k^2 + (k\sqrt{mn} + nk)/\sqrt{p}\right)$ | $O\left(\left(4mnk + 2n^2k + (m+n)k\right)/p\right)$ |
| PGNCG ($s_{\text{CG}}$) | 3 | $O(s_{\text{CG}} \log p)$ | $O\left(s_{\text{CG}}k^2 + (k\sqrt{mn} + nk)/\sqrt{p}\right)$ | $O\left(\left(4mnk + 2n^2k + s_{\text{CG}}(m+n)k^2\right)/p\right)$ |

**Algorithm 6** JointNMF via ANLS (two grids)

**Require:** $\hat{\mathbf{H}}^\mathsf{T} \in \mathbb{R}_+^{n \times k}$ is row-wise distributed across grid $\Gamma$.
**Require:** $\mathbf{C} \in \mathbb{R}_+^{k \times n}$ is a copy of $\mathbf{H}$ column-wise distributed across grid $\Gamma$.
**Ensure:** $\mathbf{W}, \mathbf{H} \approx \min_{\mathbf{W} \geq 0, \mathbf{H} \geq 0} \|\mathbf{X} - \mathbf{WH}\|_F^2 + \alpha \left\|\mathbf{S} - \mathbf{H}^\mathsf{T}\mathbf{H}\right\|_F^2$.

**function** $[\mathbf{W}, \mathbf{H}, \hat{\mathbf{H}}]$ = ParJANLS($\mathbf{X}, \Pi, \mathbf{S}, \Gamma, k, \alpha, \beta$)
  $(i, j)$ = myrank($\Pi$) ▷ $\Pi$ Processor rank
  $r = (i-1)\,p_c + j$ ▷ $\Pi$ Row-major rank
  $c = (j-1)\,p_r + i$ ▷ $\Pi$ Column-major rank
  $(x, y)$ = myrank($\Gamma$) ▷ $\Gamma$ Processor rank
  $u = (x-1)\,q_c + y$ ▷ $\Gamma$ Row-major rank
  $v = (y-1)\,q_r + x$ ▷ $\Gamma$ Column-major rank
  Initialise $\mathbf{H}^{(0)}$
  $\mathbf{C}^{(0)}$ = redist($\mathbf{H}^{(0)}, \Pi^\mathsf{T}, \Gamma^\mathsf{T}$) ▷ $\mathbf{C}$ is a copy of $\mathbf{H}$ in the $\Gamma$ grid
  **while** $t = 1, 2, \ldots$ **do** ▷ Till some stopping condition is met
  **% Compute W given H**
    $\mathbf{G}_\mathbf{H}$ = gram$\left(\Pi^\mathsf{T}, \left(\mathbf{H}^{(t)}\right)^\mathsf{T}\right)$
    $\mathbf{R}_\mathbf{H}$ = 2Dmatmul$\left(\Pi^\mathsf{T}, \mathbf{X}^\mathsf{T}, \left(\mathbf{H}^{(t)}\right)^\mathsf{T}\right)$
    $\mathbf{W}_r^{(t)}$ = update($\mathbf{G}_\mathbf{H}, \mathbf{R}_{\mathbf{H}r}$) ▷ NLS update
  **% Compute $\hat{\mathbf{H}}$ given W, H**
    $\mathbf{L}_{\hat{\mathbf{H}}} = \mathbf{G}_\mathbf{H} + \beta\mathbf{I}_k$
    $\mathbf{M}_{\hat{\mathbf{H}}}$ = 2Dmatmul$\left(\Gamma^\mathsf{T}, \mathbf{S}^\mathsf{T}, \left(\mathbf{C}^{(t)}\right)^\mathsf{T}\right)$
    $\mathbf{N}_{\hat{\mathbf{H}}}$ = redist($\mathbf{C}, \Gamma^\mathsf{T}, \Gamma$) ▷ Swap within $\Gamma$ grid
    $\mathbf{R}_{\hat{\mathbf{H}}u} = \alpha\mathbf{M}_{\hat{\mathbf{H}}u} + \beta\mathbf{N}_{\hat{\mathbf{H}}u}$
    $\hat{\mathbf{H}}_u^{(t)}$ = update$\left(\mathbf{L}_{\hat{\mathbf{H}}}, \mathbf{R}_{\hat{\mathbf{H}}u}\right)$ ▷ NLS update
  **% Compute H given W, $\hat{\mathbf{H}}$**
    $\mathbf{G}_\mathbf{W}$ = gram$(\Pi, \mathbf{W}^{(t)})$
    $\mathbf{G}_{\hat{\mathbf{H}}}$ = gram$\left(\Gamma, \hat{\mathbf{H}}^{(t)}\right)$
    $\mathbf{L}_\mathbf{H} = \mathbf{G}_\mathbf{W} + \alpha\mathbf{G}_{\hat{\mathbf{H}}} + \beta\mathbf{I}_k$
    $\mathbf{M}_\mathbf{H}$ = 2Dmatmul$(\Pi, \mathbf{X}, \mathbf{W}^{(t)})$
    $\mathbf{K}_\mathbf{H}$ = 2Dmatmul$(\Gamma, \mathbf{S}, \hat{\mathbf{H}}^{(t)})$
    $\mathbf{N}_\mathbf{H}$ = redist($\hat{\mathbf{H}}^{(t)}, \Gamma^\mathsf{T}, \Gamma$) ▷ Swap within $\Gamma$ grid
    $\mathbf{D}_{\mathbf{H}v} = \alpha\mathbf{K}_{\mathbf{H}v} + \beta\mathbf{N}_{\mathbf{H}v}$
    $\mathbf{D}_\mathbf{H}$ = redist($\mathbf{D}_\mathbf{H}, \Gamma, \Pi^\mathsf{T}$) ▷ Swap from $\Gamma$ to $\Pi$ grid
    $\mathbf{R}_{\mathbf{H}c} = \mathbf{M}_{\mathbf{H}c} + \mathbf{D}_{\mathbf{H}c}$
    $\mathbf{H}_c^{(t)}$ = update($\mathbf{L}_\mathbf{H}, \mathbf{R}_{\mathbf{H}c}$) ▷ NLS update
    $\mathbf{C}^{(t)}$ = redist($\mathbf{H}^{(t)}, \Pi^\mathsf{T}, \Gamma^\mathsf{T}$)
  **end while**
**end function**

Both grid choices introduce extra communication. First, the distribution of a symmetric matrix $\mathbf{S}$ in a rectangular processor grid causes the swaps to occur when incorporating symmetric regularization (i.e. when the matrix product $\mathbf{HS}$ changes to $\alpha\mathbf{HS} + \beta\mathbf{H}$). Second, computing the gradient of $\mathbf{H}$ requires additions with column-order 1D distributed $\mathbf{W}^\mathsf{T}\mathbf{X}$ and row-order 1D distributed $\mathbf{HS}$. Third, synchronizing the shared $\mathbf{H}$ between the grids in the double grid case needs extra communication. It can be shown that a single processor will need to send information to at most two other processors and similarly receive messages from at most two others. For example, this can be seen for communicating $\mathbf{H}$ by noticing that each processor owns $\left\lceil \frac{n}{p} \right\rceil$ or $\left\lfloor \frac{n}{p} \right\rfloor$ contiguous columns of $\mathbf{H}$. This property is violated when transferring these columns to three other contiguous processors. Thus these "swap" communications incur a small additive cost.

# 4 EXPERIMENTS

## 4.1 Experimental Setup

The Matlab experiments were performed on a server with two Intel® Xeon® E5-2680 v3 CPUs and 377 GB of DDR4-2,133 MHz DRAM. All distributed-memory experiments were run on the PACE Phoenix cluster at the Georgia Institute of Technology, wherein each node is equipped with a single Dual Intel® Xeon® Gold 6226 2.7 CPU and between 192-768 GB of DDR4-2,933 MHz DRAM [24]. Each CPU has 2 sockets each with 12 cores per socket for a total of 24 cores per node. The implementations were compiled using GCC 8.3.0 and CMake 3.20.3. PLANC uses the Armadillo linear algebra library for matrix representations and operations [28] and was linked to Armadillo 11.1 for all experiments. Sparse matrix operations utilized Armadillo's default sparse matrix functionality, whilst OpenBLAS 0.3.13 was used for all dense operations [35]. All experiments were run using OpenMPI 3.1.6 in the "flat" MPI setting [11], i.e. each core is assigned to a different single-threaded MPI process.

## 4.2 Datasets

The scaling experiments for this study were conducted on dense and sparse synthetic inputs. These dense matrices are created as nonnegative low-rank matrices by multiplying a randomly generated $\mathbf{W}$ and $\mathbf{H}$. For the sparse case, we created uniformly random matrices with a density of 0.05. All our timings are averaged over 4 different runs of 10 and 5 outer iterations for the dense and sparse cases respectively. We use $\gamma = 0.9$ for PGD as suggested by Ruder [27]. 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 [7]. This gives equal importance to both the features and connections objectives. For PGNCG we found that setting the inner CG iterations ($s_{\text{CG}}$) to 20 gave us the best results in terms of minimizing the residual.

**Table 3: Convergence studies on the different JointNMF algorithms. The relative objective, time taken, and number of function evaluations. The best performing method is highlighted in bold.**

| Input | Algorithm | Rel. Obj. | Time | Func. Eval. |
|---|---|---|---|---|
| Dense | PGD | 0.5117 | 56.78 s | 28,659.0 |
| | ANLS | **0.0002** | 36.27 s | **1,000.0** |
| | PGNCG | 0.0042 | **22.77** s | 1,288.2 |
| Sparse | PGD | 0.9587 | 49.62 s | 11,947.6 |
| | ANLS | **0.8589** | 55.03 s | **1,000.0** |
| | PGNCG | 0.8597 | **30.01** s | 1,036.0 |
| Y04 | PGD | 0.9948 | 420.98 s | 1,716.0 |
| | ANLS | 0.9028 | 101.41 s | **100.0** |
| | PGNCG | **0.8870** | **49.86** s | **100.0** |

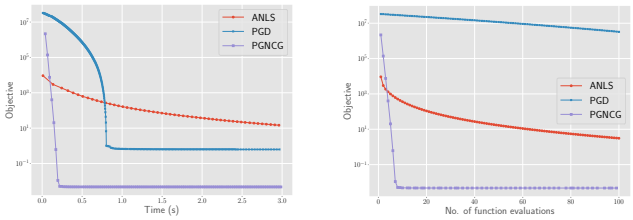
Two real world datasets were used in these experiments. Matlab convergence experiments were run on a patent dataset [7]. Each patent is encoded via tf-idf and its citations generate the connections matrix. We use the Y04 collection of patents which has $\mathbf{X} \in \mathbb{R}_+^{8,142\times3,242}$ and $\mathbf{S} \in \mathbb{R}_+^{3,242\times3,242}$. The densities for $\mathbf{X}$ and $\mathbf{S}$ are 0.0141 and 0.0041, respectively.

Large scale real world experiments were run on a subset of the Microsoft Open Academic Graph (OAG) [36], a dataset consisting of a unification of the Microsoft Academic Graph (MAG) [30] and ArnetMiner (AMiner) [32] academic graphs each respectively containing 166,192,182 and 154,771,162 papers. From this dataset, a subset of 37,732,477 papers with available abstracts and citation information were selected. These abstracts were preprocessed using stop words and stemming to form a vocabulary of 1,333 unique words. Together this vocabulary and corpus of papers were used to form a sparse $1{,}333 \times 37{,}732{,}477$ term-document matrix with 1,295,114,641 nonzeros, wherein each column represents a paper as a tf-idf vector. The resulting matrix was used as the $\mathbf{X}$ in the real world experiments. The symmetric graph Laplacian matrix $\mathbf{S}$ was then formed from the citation graph. Each of the 966,206,008 nonzeros of the resulting $37{,}732{,}477 \times 37{,}732{,}477$ matrix represents a citation between two papers. The Matrix Market file size of the $\mathbf{X}$ and $\mathbf{S}$ matrices were respectively 41 GB and 17.5 GB. The resulting dataset is referred to as the *AMinerMAG* dataset from here on. Fig. 7 and Table 5 respectively contain scaling and text clustering results from this dataset.[2]

## 4.3 Convergence Studies

We test the Matlab performance of our proposed JointNMF algorithms, PGD and PGNCG, on three different datasets and compare against the ANLS version of Du et al. [7]. The dense synthetic input consists of true low-rank inputs $\mathbf{X} = \mathbf{WH}$ and $\mathbf{S} = \mathbf{H}^T\mathbf{H}$ which are perturbed 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 $\mathbf{S}$ generated 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 of 0.1. Finally, we test convergence on the Y04 patent dataset with $k = 76$.

[2]Code and dataset information can be found at: https://github.com/ramkikannan/planc.




**(a) Convergence versus time.** **(b) Convergence versus function evaluations.**

**Figure 2: JointNMF convergence 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 surpassing PGNCG around 25 seconds (out of the graph). PGD initially decreases the objective, but plateaus soon after.**

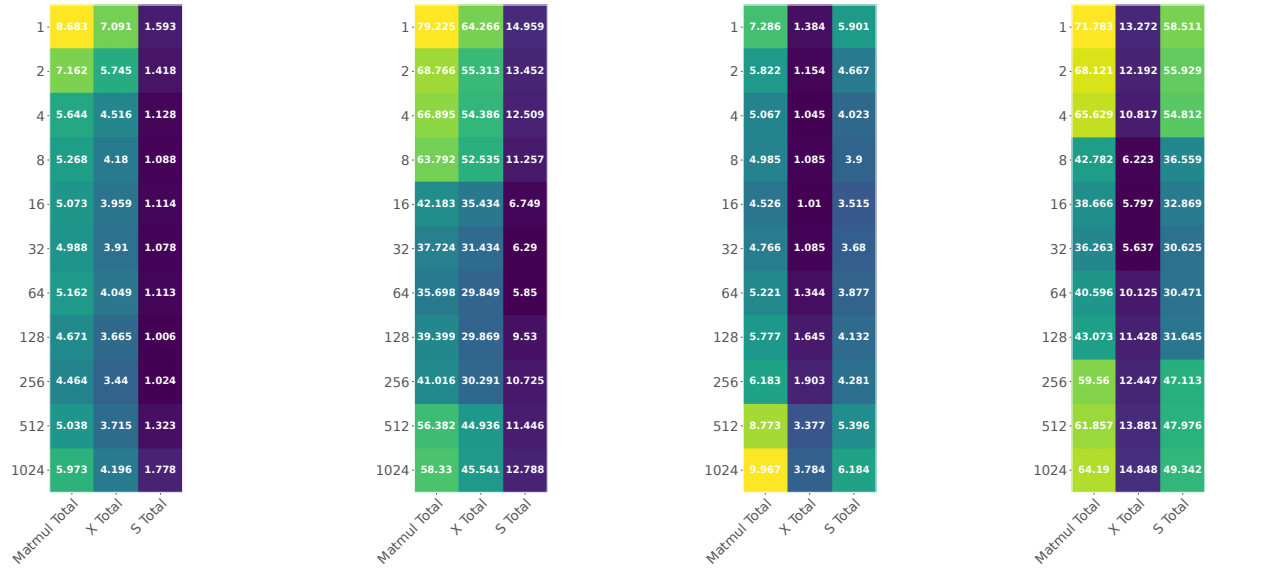
We run each algorithm five times with different random seeds. We used 1000 and 100 outer iterations for the synthetic and Y04 datasets respectively. PGD needed 2× and 10× more outer iterations to reduce the objective to a reasonable amount. All the methods are initialised with the same starting guesses $\mathbf{W}^{(0)}$ and $\mathbf{H}^{(0)}$. The average results over the runs are shown in Table 3. Apart from relative objective and running time, we also capture the number of times the objective is evaluated. 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.

Table 3 shows the final relative objective achieved by the three algorithms. It is evident that PGD converges more slowly than the other two methods. Regarding the rate of convergence for the dense case in Fig. 2, while PGD is able to perform each update quickly it is 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 suggests the use of more accurate update methods than simple gradient descent.

It is more difficult to distinguish between the PGNCG and ANLS. A surprising finding is that PGNCG runs approximately 37-51% faster even though it performs more function evaluations than 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 Grid Choices

Matrix multiplication consumes the majority of the time for these methods [9, 13]. Therefore, the choice of grid layout is a crucial one for JointNMF. First, we sweep all possible combinations of grids




**(a) Dense inputs with $\frac{m}{n} = 4$.** **(b) Sparse inputs with $\frac{m}{n} = 4$.** **(c) Dense inputs with $\frac{m}{n} = \frac{1}{4}$.** **(d) Sparse inputs with $\frac{m}{n} = \frac{1}{4}$.**

**Figure 3: Sweeping all grid configurations for running the ANLS variant of JointNMF on 1,024 processors. The experiment was run on 43 nodes (1,024 cores) and the y-axis shows the $p_r$ value for the grid chosen (with $p_c = \frac{1024}{p_r}$). The aspect ratio shows which matrix will cause the bottleneck computation. The best running times are always near the middle of the heatmaps.**

for 1,024 MPI processes (43 nodes) and benchmark the different matrix multiplication times for the ANLS method with a single grid configuration. Two different aspect ratios, 4 and $\frac{1}{4}$, were used for the experiments. Input dimensions $m \times n$ for the dense and sparse cases were (1,474,560×368,640), (179,200×716,800) and (2,457,600×614,400),(307,200 × 1,228,800), respectively. All experiments were run with $k = 50$. The times are normalized by the number of function evaluations and shown in Fig. 3.

Fig. 3 shows that depending on the aspect ratio, $\frac{m}{n}$, we can easily determine which input matrix, **X** or **S**, dominates the computation. The best runtimes appear near the middle of the heatmaps away from the degenerate 1D distributions ($p_r = 1$ or $p_c = 1$). Investigating a bit further, in Fig. 5 we can see the breakdown of the computation and communication times for the sparse input with aspect ratio 4. The communication times vary smoothly from the 1D to 2D distributions, with the minimum occurring when grid dimensions mimic the inputs (see Fig. 5b). The computation times, in Fig. 5a, are less smooth but the trend still applies. The effects of grid selection on communication times (25×) is more dramatic than on computation times (50%) as expected.

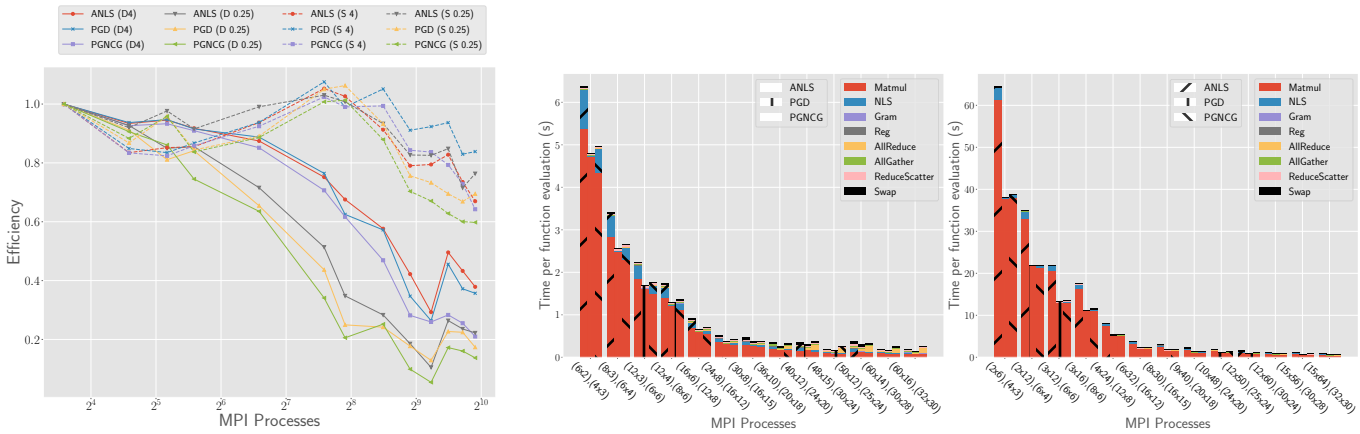
In theory, by adding the best possible times from Fig. 3, the double-grid approach should outperform the single-grid one. We run the same inputs using the best single and double grids, both empirical from Fig. 3 and theoretical, to see if this optimization works. Table 4 contains the different grid choices and their runtimes. The deviation from theoretically optimal to empirical best grid is minimal in the range of 3-11% across different cases even for large matrices in Table 4. We thus use the theoretically determined double grid for the rest of our scaling studies.

**Table 4: Single versus double grids. We compare the runtime between the empirically best grids versus the theoretically optimal ones. The deviation in results were minimal.**

| Input ($\frac{m}{n}$) | Label | X grid | S grid | Time |
|---|---|---|---|---|
| Dense (4) | Emp X | 256 × 4 | - | **47.20** s |
| | Theo X | 64 × 16 | - | 52.45 s |
| | Emp S | - | 128 × 8 | 50.75 s |
| | Theo S | - | 32 × 32 | 48.92 s |
| | Emp double | 256 × 4 | 128 × 8 | 51.26 s |
| | Theo double | 64 × 16 | 32 × 32 | 52.65 s |
| Dense (0.25) | Emp X,S and theo X | 16 × 64 | - | 46.78 s |
| | Theo S | - | 32 × 32 | 46.87 s |
| | Emp double | 16 × 64 | 16 × 64 | 47.16 s |
| | Theo double | 16 × 64 | 32 × 32 | **45.35** s |
| Sparse (4) | Emp X,S and theo X | 64 × 16 | - | 192.22 s |
| | Theo S | - | 32 × 32 | 202.76 s |
| | Emp double | 64 × 16 | 64 × 16 | **191.00** s |
| | Theo double | 64 × 16 | 32 × 32 | 191.16 s |
| Sparse (0.25) | Emp X and theo S | 32 × 32 | - | **182.01** s |
| | Theo X | 16 × 64 | - | 191.88 s |
| | Emp S | - | 64 × 16 | 204.54 s |
| | Emp double | 32 × 32 | 64 × 16 | 185.03 s |
| | Theo double | 16 × 64 | 32 × 32 | 184.14 s |

## 4.5 Scaling Studies

Strong and weak scaling results for the three variants are shown in Figs. 4 and 6 for both dense and sparse inputs with the same aspect ratios as the grid choice experiments. The problem sizes for strong scaling fill up the memory of a single socket of the cluster. The input sizes were (184,320×46,080), (23,040×92,160) and (384,000×96,000),

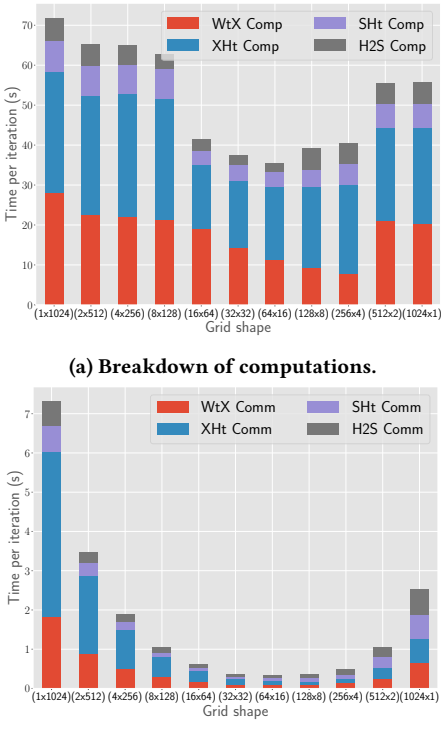



(a) Parallel efficiency.

(b) Dense strong scaling ($\frac{m}{n} = 4$).

(c) Sparse strong scaling ($\frac{m}{n} = \frac{1}{4}$).

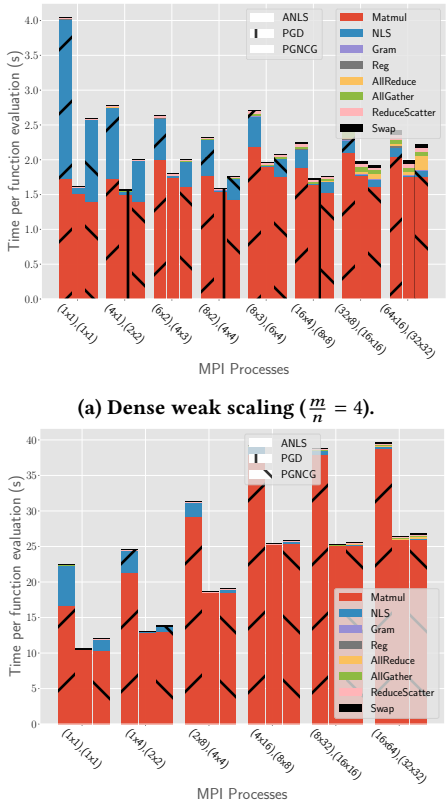
**Figure 4: Strong scaling results for both dense and sparse inputs. Matrix multiplication is the clear bottleneck for JointNMF with NLS times also showing up for the ANLS and PGNCG algorithms. The computations scale well at the rate of $\frac{1}{p}$ while the communication scales at $\frac{1}{\sqrt{p}}$. The sparse methods scale better than the dense case.**




(a) Breakdown of computations.

(b) Breakdown of communication.

**Figure 5: Grid choices for the sparse case with $\frac{m}{n} = 4$. The best performance is observed in between the 1D distributions. The effects of grid selection on communication times (25×) is more dramatic than on computation times (50%).**




(a) Dense weak scaling ($\frac{m}{n} = 4$).

(b) Sparse weak scaling ($\frac{m}{n} = \frac{1}{4}$).

**Figure 6: Weak scaling results for both dense and sparse inputs. The size of the matrices per MPI process is kept constant. Matrix multiply times remain flat once we scale across nodes. The NLS times should scale at the rate of $\frac{1}{\sqrt{p}}$.**

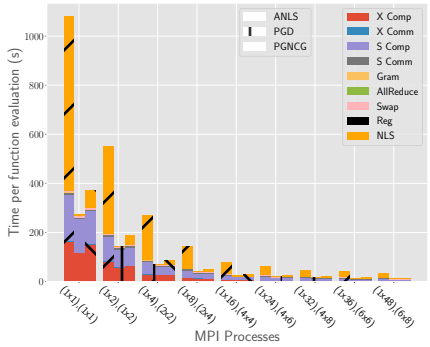



**Figure 7: Strong scaling results on real-world data. The final speedups for ANLS, PGD, and PGNCG were 33×, 25×, and 28×.**

**Table 5: Sample topics discovered from *AMinerMAG*.**

| | Topic 1 | Topic 2 | Topic 3 | Topic 4 |
|---|---|---|---|---|
| ANLS | control<br>systems<br>design<br>motor<br>nonlinear | social<br>media<br>economic<br>society<br>people | education<br>health<br>teaching<br>management<br>students | acid<br>synthesis<br>reaction<br>amino<br>acids |
| PGD | compound<br>spectrum<br>populations<br>pathways<br>correlated | dependent<br>children<br>fold<br>assessed<br>investigate | condition<br>changing<br>formed<br>empirical<br>variation | correlated<br>industrial<br>reliability<br>grade<br>parameter |
| PGNCG | control<br>systems<br>design<br>motor<br>nonlinear | social<br>media<br>economic<br>development<br>society | teaching<br>education<br>students<br>learning<br>training | acid<br>protein<br>activity<br>synthesis<br>dna |

(48,000×192,000) for the dense and sparse cases respectively. Matrix-multiplication dominates the running time with NLS coming in second. Both these computations scale well as we see good scaling till 10 nodes (240 cores) for both types of inputs. Past 240 processes, we can see communication costs start showing up in the dense timing breakdowns (see Fig. 4b) while sparse is still bottlenecked by matrix multiplication and continues to scale (see Fig. 4c). This is seen in the efficiency chart in Fig. 4a, with the sparse cases scaling till 960 processes at 60% parallel efficiency while dense can only achieve 20%.

Fig. 7 shows scaling results on the *AMinerMAG* dataset for all algorithm implementations averaged over 50 iterations. The speedup on 48 cores for the ANLS, PGD, and PGNCG implementations were respectively 33×, 25×, and 28×. From this, we see the NLS, $\mathbf{X}$ computation (X Comp), and $\mathbf{S}$ computation (S Comp) dominate the runtime. Similar to the synthetic experiments, the NLS time is particularly high for the ANLS approach, resulting in both the proposed PGD and PGNCG approaches being faster than it for all processor counts.

## 4.6 Text Clustering

To demonstrate the effectiveness of our implementations, we performed text clustering on the *AMinerMAG* dataset utilizing each algorithm, the results of which can be seen in Table 5. In each instance, the algorithm was run on the *AMinerMAG* dataset for 100 iterations with $k = 16$. The top five keywords from four selected topics are provided to give a general intuition for the resulting clusters. From this we observed that all three approaches yielded interpretable clusters, empirically supporting the validity of the algorithms and their implementations. Furthermore, we observed that the clusters found by the ANLS and PGNCG implementations bore striking similarities. This can be seen in Table 5 by the overlaps in top key words between ANLS and PGNCG for the four selected topics. Within the context of these experiments and based upon the conclusions drawn from Fig. 2, it is likely that both ANLS and PGNCG converged to a reasonable solution, whilst PGD did not converge after the 100 iterations. This may serve as an explanation as to similarities between the clusters found by ANLS and PGNCG, as well as their interpretability relative to those found by PGD.

# 5 DISCUSSION

Among these first distributed-memory parallel methods for JointNMF, ANLS and PGNCG outperform the first-order PGD method. In the serial setting PGD remains relatively close to the other methods by performing a large number of inexpensive update steps (see Section 4.3), but this advantage disappears in the distributed scenario. Distinguishing between ANLS and PGNCG is more difficult. PGNCG performs one fewer large matrix multiplication per function evaluation but may perform more such calls during line search.

Performance was not very sensitive to the choice of logical processor grid sizes except at extreme aspect ratios (see Section 4.4). Nevertheless, the flexibility of the double grids is useful when one would like to tune the local input matrix dimensions to either save memory or exploit specific matrix multiply kernels.

The choice of using the Gauss-Newton method is difficult to justify since it is hard to determine *a priori* if the Hessian approximation is close to the true Hessian of JointNMF. The residuals encountered during the course of the PGNCG algorithm can be large, especially in the sparse case. PGNCG performs well empirically, suggesting other second-order methods with fewer assumptions on the Hessian like truncated Newton or L-BFGS for JointNMF could also prove useful. That, as well as new applications of a scalable JointNMF, are perhaps the most interesting avenues for future work.

# ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant Nos. OAC-2106920 and CCF-1942892. It is also based upon work supported by the U.S. Department of Energy, Office of Science under Contract DE-AC02-06CH11357 at Argonne National Laboratory and UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 at Oak Ridge National Laboratory. Additionally, it is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program under Award Number DE-SC-0023296. Koby Hayashi acknowledges support from the United States Department of Energy through the Computational Sciences Graduate Fellowship (DOE CSGF) under grant number: DE-SC0020347. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

# REFERENCES


[1] Hussam Al Daas, Grey Ballard, Laura Grigori, Suraj Kumar, and Kathryn Rouse. 2022. Brief Announcement: Tight Memory-Independent Parallel Matrix Multiplication Communication Lower Bounds. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA) *(SPAA '22)*. Association for Computing Machinery, New York, NY, USA, 445–448. https://doi.org/10.1145/3490148.3538552

[2] Dimitri P Bertsekas. 1982. Projected Newton methods for optimization problems with simple constraints. *SIAM Journal on control and Optimization* 20, 2 (1982), 221–246. https://doi.org/10.1137/0320018

[3] Dimitri P Bertsekas. 1999. *Nonlinear Programming*. Athena Scientific.

[4] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783. https://doi.org/10.1002/cpe.1206

[5] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, IEEE, 261–272. https://doi.org/10.1109/IPDPS.2013.80

[6] Rundong Du. 2018. *Nonnegative matrix factorization for text, graph, and hybrid data analytics*. Ph. D. Dissertation. Georgia Institute of Technology. http://hdl.handle.net/1853/59914

[7] Rundong Du, Barry Drake, and Haesun Park. 2019. Hybrid clustering based on content and connection structure using joint nonnegative matrix factorization. *Journal of Global Optimization* 74, 4 (2019), 861–877. https://doi.org/10.1007/s10898-017-0578-x

[8] Srinivas Eswar, Koby Hayashi, Grey Ballard, Ramakrishnan Kannan, Michael A. Matheson, and Haesun Park. 2021. PLANC: Parallel Low-Rank Approximation with Nonnegativity Constraints. *ACM Trans. Math. Softw.* 47, 3, Article 20 (jun 2021), 37 pages. https://doi.org/10.1145/3432185

[9] Srinivas Eswar, Koby Hayashi, Grey Ballard, Ramakrishnan Kannan, Richard Vuduc, and Haesun Park. 2020. Distributed-Memory Parallel Symmetric Nonnegative Matrix Factorization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 74, 14 pages. https://doi.org/10.1109/SC41405.2020.00078

[10] Srinivas Eswar, Ramakrishnan Kannan, Richard Vuduc, and Haesun Park. 2021. ORCA: Outlier detection and Robust Clustering for Attributed graphs. *Journal of Global Optimization* 81, 4 (2021), 967–989. https://doi.org/10.1007/s10898-021-01024-z

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Springer Berlin Heidelberg, Budapest, Hungary, 97–104. https://doi.org/10.1007/978-3-540-30218-6_19

[12] Nicolas Gillis. 2020. *Nonnegative Matrix Factorization*. SIAM, Philadelphia, PA. https://doi.org/10.1137/1.9781611976410

[13] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. A high-performance parallel algorithm for nonnegative matrix factorization. *ACM SIGPLAN Notices* 51, 8 (2016), 1–11. https://doi.org/10.1145/3016078.2851152

[14] R. Kannan, G. Ballard, and H. Park. 2018. MPI-FAUN: An MPI-Based Framework for Alternating-Updating Nonnegative Matrix Factorization. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (March 2018), 544–558. https://doi.org/10.1109/TKDE.2017.2767592

[15] Dongmin Kim, Suvrit Sra, and Inderjit S Dhillon. 2007. Fast Newton-type methods for the least squares nonnegative matrix approximation problem. In *Proceedings of the 2007 SIAM international conference on data mining*. SIAM, 343–354. https://doi.org/10.1137/1.9781611972771.31

[16] Jingu Kim, Yunlong He, and Haesun Park. 2014. Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework. *Journal of Global Optimization* 58, 2 (2014), 285–319. https://doi.org/10.1007/s10898-013-0035-4

[17] Da Kuang, Chris Ding, and Haesun Park. 2012. Symmetric nonnegative matrix factorization for graph clustering. In *Proceedings of the 2012 SIAM international conference on data mining*. SIAM, 106–117. https://doi.org/10.1137/1.9781611972825.10

[18] Da Kuang, Sangwoon Yun, and Haesun Park. 2015. SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering. *Journal of Global Optimization* 62, 3 (2015), 545–574. https://doi.org/10.1007/s10898-014-0247-2

[19] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2020. *Mining of massive datasets* (3rd ed. ed.). Cambridge University Press. https://doi.org/10.1017/9781108684163

[20] Chih-Jen Lin. 2007. Projected gradient methods for nonnegative matrix factorization. *Neural computation* 19, 10 (2007), 2756–2779. https://doi.org/10.1162/neco.2007.19.10.2756

[21] Jialu Liu, Chi Wang, Jing Gao, and Jiawei Han. 2013. Multi-view clustering via joint nonnegative matrix factorization. In *Proceedings of the 2013 SIAM international conference on data mining*. SIAM, 252–260. https://doi.org/10.1137/1.9781611972832.28

[22] Ninghao Liu, Xiao Huang, and Xia Hu. 2017. Accelerated Local Anomaly Detection via Resolving Attributed Networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 2337–2343. https://doi.org/10.24963/ijcai.2017/325

[23] Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization* (2e ed.). Springer, New York, NY, USA. https://doi.org/10.1007/978-0-387-40065-5

[24] PACE. 2017. *Partnership for an Advanced Computing Environment (PACE)*. https://www.pace.gatech.edu

[25] Zhen Peng, Minnan Luo, Jundong Li, Huan Liu, and Qinghua Zheng. 2018. ANOMALOUS: A Joint Modeling Approach for Anomaly Detection on Attributed Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 3513–3519. https://doi.org/10.24963/ijcai.2018/488

[26] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151. https://doi.org/10.1016/S0893-6080(98)00116-6

[27] Sebastian Ruder. 2017. An overview of gradient descent optimization algorithms. arXiv:1609.04747 [cs.LG]

[28] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26. https://doi.org/10.21105/joss.00026

[29] Mark Schmidt, Dongmin Kim, and Suvrit Sra. 2012. Projected Newton-type Methods in Machine Learning. In *Optimization for Machine Learning*, Suvrit Sra, Sebastian Nowozin, and Stephen J Wright (Eds.). MIT Press, Chapter 11, 305–329.

[30] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15 Companion)*. Association for Computing Machinery, New York, NY, USA, 243–246. https://doi.org/10.1145/2740908.2742839

[31] Jiliang Tang, Xufei Wang, and Huan Liu. 2011. Integrating social media data for community detection. In *Modeling and Mining Ubiquitous Social Media*. Springer, 1–20. https://doi.org/10.1007/978-3-642-33684-3_1

[32] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 990–998. https://doi.org/10.1145/1401890.1402008

[33] N Vervliet and L De Lathauwer. 2019. Numerical optimization-based algorithms for data fusion. In *Data Handling in Science and Technology*. Vol. 31. Elsevier, 81–128. https://doi.org/10.1016/B978-0-444-63984-4.00004-1

[34] Joyce Jiyoung Whang, Rundong Du, Sangwon Jung, Geon Lee, Barry Drake, Qingqing Liu, Seonggoo Kang, and Haesun Park. 2020. MEGA: Multi-view semi-supervised clustering of hypergraphs. *Proceedings of the VLDB Endowment* 13, 5 (2020), 698–711. https://doi.org/10.14778/3377369.3377378

[35] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2012. OpenBLAS.

[36] Fanjin Zhang, Xiao Liu, Jie Tang, Yuxiao Dong, Peiran Yao, Jie Zhang, Xiaotao Gu, Yan Wang, Bin Shao, Rui Li, and Kuansan Wang. 2019. OAG: Toward Linking Large-Scale Heterogeneous Entity Graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2585–2595. https://doi.org/10.1145/3292500.3330785