Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Distributed Non-Negative Matrix Factorization with Determination of the Number of Latent Features

Gopinath Chennupati[1]* · Raviteja Vangara[2] · Erik Skau[1] · Hristo Djidjev[1] · Boian Alexandrov[2]

Received: date / Accepted: date

Abstract The holistic analysis and understanding of the latent (that is, not-directly observable) variables and patterns buried in large datasets is crucial for data-driven science, decision making and emergency response. Such exploratory analyses require devising unsupervised learning methods for data mining and extraction of the latent features, and non-negative matrix factorization (NMF) is one of the prominent such methods. NMF is based on compute-intense non-convex constrained minimization, which, for large datasets requires fast and distributed algorithms. However, current parallel implementations of NMF fail to estimate the number of latent features. In practice, identifying these features is both difficult and significant for pattern recognition and latent feature analysis, especially for large dense matrices. In this paper, we introduce a distributed NMF algorithm coupled with distributed custom clustering followed by a stability analysis on dense data, which we call *DnMFk*, to determine the number of latent variables. The results on synthetic data and the classical *Swimmer* data set demonstrate the accuracy of model

* Corresponding author.

G. Chennupati
Tel: +1-505-667-7046
E-mail: gchennupati@lanl.gov

R. Vangara
E-mail: rvangara@lanl.gov

E. Skau
E-mail: ewskau@lanl.gov

H. Djidjev
E-mail: djidjev@lanl.gov

B. Alexandrov
E-mail: boian@lanl.gov
[1]Information Sciences (CCS-3) Group
[2]Theoretical Division (T-1) Group
Los Alamos National Laboratory (LANL), Los Alamos, NM, 87545 USA

determination while scaling nearly linearly across multiple processors for large data. Further, we employ *DnMFk* to determine the number of hidden features from a tera byte (TB) matrix.

Keywords NMF, latent features, distributed processing, clustering, parallel programming, silhouette, big data.

1 Introduction

Extracting useful information from big datasets requires understanding of the processes generating the data [24]. While such datasets are formed through recording observable quantities, the underlying processes and variables often are not known and not directly observable, i.e., they are hidden, or *latent*. The latent variables can be seen as the hidden causes for the resulting set of observed variables. Many types of statistical models contain latent variables, including factor analytic models and finite mixture models. The methods for inferring latent variables include Hidden Markov Models [9], Partial Least Squares Regression [59], Latent Semantic Analysis [18], and Factor Analysis [51], while the unsupervised learning methods [7] aim to extract hidden patterns and groups based on defined similarities without prior training.

The most common unsupervised method is clustering [27], while other unsupervised methods approximate the low-rank factors of input data (usually, subject to various constraints). These methods include Principal Component Analysis (PCA) [30], Singular Value Decomposition (SVD) [26], Independent Component Analysis (ICA) [6], Non-Negative Matrix Factorization (NMF) [48], and their high-dimensional tensor variations [15]. The latent variables (aka latent or hidden features) can be computed from the observables of a low-rank matrix factorization that maps the columns of the initial data (the observables) to a smaller set of columns (the latent features). There is always uncertainty in determining the number of these latent variables, most often, heuristic methods answer the question of latent feature identification.

Nonetheless, PCA/SVD/ICA have limitations in relating the extracted latent features to physically interpretable quantities. In contrast, NMF extracts features that are parts of the original data, which are easy to understand and interpret. This makes NMF invaluable for various applications, such as features extraction, dimensionality reduction, blind source separation, image recognition, text categorization and many others (see [17]). However, while non-negative factorization algorithms with determining the correct number of latent features can extract extremely valuable information from the data, it is also computationally very demanding task that requires fast and distributed implementation, especially for large datasets.

In this paper, we present an efficient distributed algorithm for non-negative matrix factorization, *DnMFk*, that determines the number of latent features in large dense datasets. *DnMFk* contains several distributed algorithms. To that end, the main contributions of this paper are:

- The first distributed model determination (number of latent feature identification) algorithm for large and dense non-negative matrices.
- A distributed implementation to create an ensemble of non-negative matrices from a original matrix through a resampling technique.
- Parallelizing a custom clustering required to group the factorized solutions that are already distributed across multiple processors.
- Evaluate the stability of the obtained clusters in a distributed manner, which, in turn, determines the number of latent variables in the data.

We evaluate our approach on a number of synthetic matrices, and show that in all cases, the predicted factor matrices are highly correlated with the known solutions. The scalability results of *DnMFk* show that the distributed algorithm scales nearly linearly for large datasets across multiple processors. Furthermore, we apply *DnMFk* to identify the number of hidden features in large dense data. We find the latent features of $1\,TB$ matrix in almost 14 hours using 4096 processors (Intel Xeon E5-2695 with $2.1GHz$ speed) with a reconstruction error rate of 6.1%. To the best of our knowledge, *DnMFk* is the first to identify the hidden features in $1\,TB$ data.

The remainder of the paper is organized as follows: Section 2 describes NMF, the importance of model determination, and the existing parallel NMF implementations; Section 3 explains the preliminary concepts; Section 4 presents our distributed algorithm; Section 5 shows the cost analysis; Section 6 validates the *DnMFk*; and Section 7 concludes and suggests future directions.

2 Background

2.1 Non-Negative matrix factorization (NMF)

Non-Negative matrix factorization (NMF) [48] is a well-known unsupervised method that approximates a given non-negative data-matrix, $\boldsymbol{A} \in \mathbb{R}_{+}^{m \times n}$, with a product of two factor matrices, $\boldsymbol{W} \in \mathbb{R}_{+}^{m \times k}$ and $\boldsymbol{H} \in \mathbb{R}_{+}^{k \times n}$, such that $\boldsymbol{A} \approx \boldsymbol{W}\boldsymbol{H}$. The factor matrices, $\boldsymbol{W}$ and $\boldsymbol{H}$, are both non-negative with one small dimension, k. The usual interpretation of NMF is as a method for a low-rank decomposition that minimizes a given divergence or distance metric, $||...||_{dist}$, i.e., for finding $\min_{\boldsymbol{W},\boldsymbol{H}} ||\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}||_{dist}$ constrained with the non-negativity of $\boldsymbol{W}$ ($\boldsymbol{W} \geq 0$) and $\boldsymbol{H}$ ($\boldsymbol{H} \geq 0$).

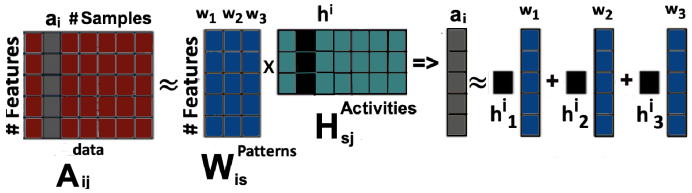

Fig. 1: NMF decomposition of a non-negative matrix.

Fig. 1 shows the decomposition of the input matrix $\mathbf{A}$ into low dimensional factors $\boldsymbol{W}$ and $\boldsymbol{H}$. This also shows, expressing a sample ($\boldsymbol{a}_i$) as a linear combination of the latent variables in $\boldsymbol{W}$ and $\boldsymbol{H}$. Importantly, NMF is underpinned with a statistical model of superimposed components (the number of these components is equal to the size of the small dimension, k) that can be treated as latent features in Gaussian, Poisson, or other mixture models [23]. NMF minimization is equivalent to the expectation-minimization (EM) algorithm [19]. In this probabilistic interpretation of NMF, the manifested variables are the columns $\mathbf{a}_1, \ldots, \mathbf{a}_n$ of the matrix $\boldsymbol{A}$ generated from the latent variables, $\mathbf{h}_1, \ldots, \mathbf{h}_k$, that are the columns of the matrix $\boldsymbol{H}$. Specifically, each observable $\mathbf{a}_i$ is generated from a probability distribution with mean $\langle \mathbf{a}_i \rangle = \sum_{s=1}^{k} \boldsymbol{W}_s \boldsymbol{h}_{s,i}$, where k is the number of the latent variables [40]. Thus, the influence of $\mathbf{h}_{s,i}$ on $\mathbf{a}_i$ is through the basis patterns represented using the columns $\mathbf{w}_1, ..., \mathbf{w}_k$ of the matrix $\boldsymbol{W}$ (see Fig. 1). A mathematically rigorous formalism of the probabilistic NMF can be found in [37].

2.2 Solving the NMF minimization problem

The NMF problem is, given a matrix $\boldsymbol{A}$, to find non-negative factors, $\boldsymbol{W}$ and $\boldsymbol{H}$, that solve the optimization problem in Eq. 1

$$\text{Minimize } O(\boldsymbol{W}, \boldsymbol{H}) = \frac{1}{2} \|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|^2_{dist}\,, \ \mathbf{W} \geqslant \mathbf{0}; \ \ \mathbf{H} \geqslant \mathbf{0}. \tag{1}$$

Here we use the Frobenius norm, $\|\mathbf{X}\|_F = \sqrt{\sum_i \sum_j x_{ij}^2}$, to measure the distance between two matrices. The NMF optimization problem, (Eq. 1.), can be solved using various algorithms such as Multiplicative Update (MU) [40], Alternating Non-negative Least Squares (ANLS) based Block Principal Pivoting (BPP) algorithm that uses Karush-Kuhn-Tucker (KKT) conditions to estimate the optimal factors [33,34], Hierarchical Alternating Least Squares (HALS) [16], alternating direction method of multipliers (ADMM) [54], AO-ADMM [28], and others. These algorithms alternatively update the low rank factors ($\boldsymbol{W}$ and $\boldsymbol{H}$) as in Eq. 2

$$\begin{aligned} \boldsymbol{W} &\leftarrow \underset{\boldsymbol{W} \geqslant 0}{\arg\min} \|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2 \\ \boldsymbol{H} &\leftarrow \underset{\boldsymbol{H} \geqslant 0}{\arg\min} \|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2, \end{aligned} \tag{2}$$

where each of the Non-negative Least Squares (NLS) sub-problems is optimized to produce a better reconstruction. For initial values of $\boldsymbol{W}$ and $\boldsymbol{H}$, one can use random guesses, although there are other available methods [56].

2.3 Model determination: estimating the number (k) of latent features

One of the prerequisites of the NMF algorithm is the need to have prior knowledge of the number of latent features, k, or a method to estimate that number.

A broadly accepted method to estimate the latent dimensionality in NMF is a Bayesian modeling method that exemplifies the concept of relevance, namely, Automatic Relevance Determination (ARD). ARD was originally introduced for neural networks NMF [44], then for Bayesian PCA [12], and later for NMF [55,57]. For NMF, ARD requires knowledge of the prior distributions of $\boldsymbol{W}$, $\boldsymbol{H}$ and their variances. However, the prior distributions of the variances of $\boldsymbol{W}$ and $\boldsymbol{H}$ are not directly connected with the observation data and guessing them can strongly influence the estimation of the number of latent factors. Thus, the Bayesian model selection approaches [55,57] introduce additional hyper-parameters, which in practice are generally unknown. The method we use in our implementation, NMFk, is discussed in the next subsection.

2.4 NMFk: NMF with custom clustering to find the number of latent features

A recent model determination technique, *NMFk* [3,5], complements classical NMF with custom k-means clustering and Silhouette statistics [49]. NMFk works on the principle of finding a trade-off between the stability of the extracted features (from several NMF-minimizations at a given k) and the accuracy of the minimization in order to estimate the optimal number of latent features. Brunet et al. [13] showed a similar method to identify the number of clusters in the observational matrix $\boldsymbol{A}$ using NMF. However, recent findings [60] demonstrate the inability of that method to find the correct number of features.

Compared to the method of Brunet et al. [13], the NMFk method works by (i) determining the stability of the main patterns (the columns of $\boldsymbol{W}$), rather than clustering stability of $\boldsymbol{A}$, and (ii) using a specific random resampling of the initial data matrix, $\boldsymbol{A}$, Fig. 2, consistent with the latent features models.

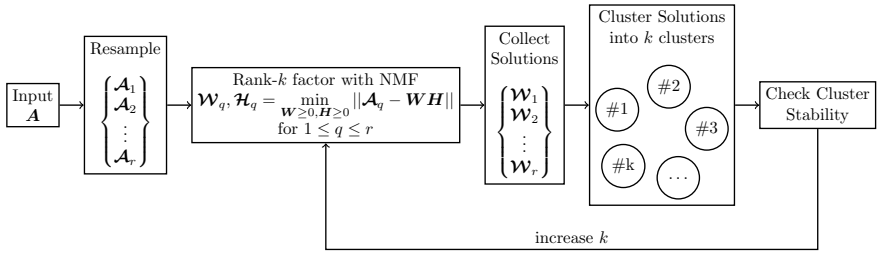

Fig. 2: NMFk with resampling of initial matrix, $\boldsymbol{A}$.

Specifically, NMFk creates an ensemble of matrices: $\boldsymbol{A}_1$, $\boldsymbol{A}_2$,..., $\boldsymbol{A}_r$ such that $\boldsymbol{A}_q = \boldsymbol{A} \odot \Delta_q, q = 1, ..., r$, where, for example, Δ_q is a random sample from uniform distribution, and then it computes clusters of the NMF solutions of the ensemble while $\odot$ stands for element-wise multiplication. Laurberg et al. [39] proved that when the NMF of a matrix $\boldsymbol{A} = \boldsymbol{W}\boldsymbol{H}$ is unique, small

noise perturbations of $\boldsymbol{A}$ result in small perturbations in the factors $\boldsymbol{W}$ and $\boldsymbol{H}$. Hence, we expect that the NMF solutions of the ensemble $\boldsymbol{A}_1$, $\boldsymbol{A}_2$,..., $\boldsymbol{A}_r$, namely, the column vectors of the matrices $\boldsymbol{W}_i$, where $\boldsymbol{A}_i = \boldsymbol{W}_i\boldsymbol{H}_i$, are well clustered into k clusters, at the correct value of k. Based on this observation, NMFk analyzes, for each k in an acceptable range, the quality of the corresponding clustering as well as the error of approximation of $\boldsymbol{A}_i$ by its factors, to determine the correct value of k.

NMFk has been successfully applied to decompose the largest available dataset of human cancer genome [4]; for extraction of subsurface pressure transients [3] and contaminants [58] originating from an unknown number of sources that may propagate with a finite speed in nondispersive [29] or dispersive media [53], as well as to extract original crystal structures and phase diagram of X-ray spectra of material combinatorial libraries [52].

2.5 Existing parallel NMF implementations

Existing parallel NMF algorithms exploit shared memory processors (such as multi (CPU) or many (GPU) core) and distributed memory processors. Of these algorithms, Battenberg and Wessel [8] implemented multiplicative update (MU) based parallel NMF with OpenMP (on CPUs) and CUDA (on GPUs), which were used independently for audio source separation. Fairbanks et al. [22] developed a BPP based OpenMP implementation of NMF that analyzed the temporal behavior of graphs through clustering of vertices.

Other MU based GPU implementations include the following. Lopes and Ribeiro [43] exploited all the cores of a single GPU for the task of face recognition, while the parallel implementations used both Euclidean and Kullback-Leibler (KL) divergence objective functions; Mejía-Roa et al. [45,46] proposed NMF-mGPU that exploited both single and multiple GPUs using MPI in order to cluster large gene expression data. Other GPU implementations of NMF were used in information extraction from text [38], and Koitka et al. [35] presented an R implementation. Recently, Moon et al. [47] introduced PL-NMF for both GPUs and CPUs, minimizing the data movement through local matrix-matrix computations.

The distributed NMF algorithms in literature use MPI or Hadoop. Dong et al. [20] proposed an MPI based NMF that partitioned the factors into smaller blocks, where multiple threads operated on each sub-block. At the end, MPI collective communication performed a reduction operation on the result of threads. Liu et al. [42] partitioned the factor matrices along the lower rank (k) rather than other dimensions (m, n), which improved the data locality and reduced the communication costs across MPI processes. Recently, Kannan et al. [32] presented an MPI based generic framework for NMF algorithms with alternating updates. Their implementation partitioned the data into blocks on a virtual processor grid. This reduced the communication costs through the use of MPI collectives.

Hadoop based implementations include the following. Gemmula et al. [25] implemented a scalable, fast converging distributed stochastic gradient based NMF for large scale datasets. Later, FlexiFact in [11] employed MapReduce for distributing the decomposition of matrices and tensors using stochastic gradient descent. Liao et al. [41] introduced CloudNMF, a MapReduce implementation of MU updates, which was eventually applied on large scale biological datasets. Yin et al. [61] implemented a scalable NMF using Hadoop MapReduce framework, where the matrix operations were performed on blocks.

All of the previous distributed NMF implementations require an explicit knowledge of the number of latent features k, which, in general, is unknown. For large datasets, the task to find k is especially cumbersome. Therefore, the distributed NMF with model determination, *DnMFk*, is the first distributed implementation to offer the functionality of NMFk for model determination. We are using alternating update framework introduced in [32] along with a distributed AO-ADMM optimization algorithm for NMF optimization. Finally, we demonstrate that *DnMFk* is able to extract the correct number of hidden features from synthetic matrices with predetermined number of factors and from the well-known *Swimmer* dataset [21].

3 Foundations

We describe the notations, MPI collectives, *NMFk* algorithm and the AO-ADMM optimization used in the paper.

3.1 Preliminaries

Table 1 summarizes the notations used in this paper. We denote a cube (three-dimensional tensor) with *bold upper case script* letter, matrices with *bold upper case* letters, and vectors with *bold lower case* letters. For example, $\boldsymbol{\mathcal{X}}$ is a cube, $\boldsymbol{X}$ is a matrix, and $\boldsymbol{x}$ is a vector. $\boldsymbol{\mathcal{A}}_i$ is the i^{th} perturbation of input matrix $\boldsymbol{A}$. $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{H}}$ denote cubes with each slice representing a matrix. $\boldsymbol{\mathcal{W}}_i$ represents the left low rank of k for i^{th} perturbation. The median low rank matrices are $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{H}}$. $\boldsymbol{\mathcal{W}}_q^{(i)}$ is the sub slice ($\boldsymbol{W}^{(i)}$) of q^{th} perturbation on i^{th} processor. Finally, $\boldsymbol{s}$ represents the *Silhouette statistics* of each data point in all the clusters. The vector $\boldsymbol{e}$ stands for the reconstruction error for a given perturbation at a given k.

3.2 MPI terminology

In our distributed implementation, we use MPI specific collective communication operations, specifically, **all_gather**, **all_reduce** and **reduce_scatter**. For example, let **a** be a vector with n elements distributed across p processes, that is, n/p elements per process. The **all_gather** collective gathers all the

Table 1: Notations

Notation	Dimensions	Description
$\boldsymbol{A}$	$m \times n$	Input matrix
$\boldsymbol{W}$	$m \times k$	Left low rank factor
$\boldsymbol{H}$	$k \times n$	Right low rank factor
$\boldsymbol{X}_i$	$m \times 1$	i^{th} column of matrix $\boldsymbol{X}$
$\boldsymbol{\mathcal{X}}_i$	$m \times n$	i^{th} slice (matrix) of cube $\boldsymbol{\mathcal{X}}$
m	scalar	Number of rows of a matrix
n	scalar	Number of columns of a matrix
k	scalar	Low rank
k_l	scalar	Lower bound of low rank
k_u	scalar	Upper bound of low rank
p	scalar	Count of parallel processes
p_r	scalar	Count of rows in processor grid
p_c	scalar	Count of columns in processor grid
r	scalar	Number of perturbations
$\boldsymbol{\mathcal{W}}$	$m \times k \times r$	A cube of left low rank factors ($\boldsymbol{W}$)
$\boldsymbol{\mathcal{H}}$	$m \times k \times r$	A cube of right low rank factors ($\boldsymbol{H}$)
$\boldsymbol{\mathcal{W}}_q$	$m \times k \times 1$	The q^{th} slice of cube $\boldsymbol{\mathcal{W}}$.
$\boldsymbol{\mathcal{W}}^{(i)}$	$\frac{m}{p} \times k \times r$	sub cube of $\boldsymbol{\mathcal{W}}$ on the i^{th} processor.
$\boldsymbol{\mathcal{W}}_q^{(i)}$	$\frac{m}{p} \times k$	The q^{th} sub slice of $\boldsymbol{\mathcal{W}}$ on i^{th} processor.
$\boldsymbol{\mathcal{H}}$	$k \times n \times r$	All right low rank factors for r perturbations
$\widetilde{\boldsymbol{W}}$	$m \times k$	Median left low rank factor
$\widetilde{\boldsymbol{H}}$	$k \times n$	Median right low rank factor
$\boldsymbol{s}$	$k \times 1$	Average silhouettes of each cluster
$\boldsymbol{e}$	$k \times 1$	Average reconstruction error

local data vectors to all the processes, thereby each process holds a copy of the entire vector, **a**. For a vector of size n on each process, the **all_reduce** performs an element-wise sum across all processes, where a copy of the resultant n dimensional vector is present in all the processes. The **reduce_scatter** collective performs the addition as in **all_reduce**, but then the resultant vector is spread across the processes, where each process holds a sum vector of size n/p. A detailed description of MPI collectives is in [14].

3.3 Generic NMFk algorithm

In order to identify k, we compute r ($r > 30$) NMF-minimizations for each candidate value for k. Moreover, we let $k \in (k_l, k_u)$ for suitable values of k_l and k_u. Finally, we apply cluster analysis to determine the optimal candidate. In the worst case, $(k_l, k_u) = [1, \min(m, n))$, but in practice $k_u \ll \min(m, n)$.

Specifics are given in Algorithm 1, which describes the generic (sequential) NMFk implementation. For each k in the interval (k_l, k_u), we execute NMF for r runs. Before each run of NMF (line 4), we perturb the original input $\boldsymbol{A}$ (line 3) using a random sample from a distribution (for example, *uniform*). Note that each NMF run uses different random initial conditions. As a result, we get a set of r solutions at each k, which we store in cubes (tensors) $\boldsymbol{\mathcal{W}}$

Algorithm 1 NMFk($\boldsymbol{A}, k_l, k_u, r$) – Generic *NMFk*

Require: $\boldsymbol{A} \in \mathbb{R}_+^{m \times n}$, k_l, k_u, r
1: **for** k **in** k_l to k_u **do**
2: **for** q **in** 1 to r **do**
3: $\boldsymbol{A'} = Perturb(\boldsymbol{A}, \delta)$ ▷ Resampling to create ensembles
4: $\boldsymbol{\mathcal{W}}_q, \boldsymbol{\mathcal{H}}_q = \text{NMF}(\boldsymbol{A'}, k)$
5: **end for**
6: $\boldsymbol{\mathcal{W}'}, \boldsymbol{\mathcal{H}'} = \text{customCluster}(\boldsymbol{\mathcal{W}}, \boldsymbol{\mathcal{H}})$ ▷ Custom clustering of factor solutions
7: $\widetilde{\boldsymbol{W}}$, $\widetilde{\boldsymbol{H}} = \text{mediansWH}(\boldsymbol{\mathcal{W}'}, \boldsymbol{\mathcal{H}'})$
8: save($\widetilde{\boldsymbol{W}}$, $\widetilde{\boldsymbol{H}}$) ▷ Save medians at k
9: $\boldsymbol{s}_k = \text{clusterStability}(\boldsymbol{\mathcal{W}'})$ ▷ Quality of clusters
10: $\boldsymbol{e}_k = \text{reconstructErr}(\boldsymbol{A}, \widetilde{\boldsymbol{\mathcal{W}}}, \widetilde{\boldsymbol{\mathcal{H}}})$
11: **end for**
12: $k_{opt} = optRank(\{(\boldsymbol{s}_k, \boldsymbol{e}_k) \mid k \in [k_l, k_u]\})$
Ensure: $k = k_{opt}$, $\widetilde{\boldsymbol{W}}[k] \in \mathbb{R}_+^{m \times k}$, $\widetilde{\boldsymbol{H}}[k] \in \mathbb{R}_+^{k \times n}$, $\mathbf{A} \approx \widetilde{\mathbf{W}}[\mathbf{k}]\,\widetilde{\mathbf{H}}[\mathbf{k}]$

$= (\boldsymbol{W}, \boldsymbol{W}, \ldots, \boldsymbol{W})$ and $\boldsymbol{\mathcal{H}} = (\boldsymbol{H}, \boldsymbol{H}, \ldots, \boldsymbol{H})$. Each left low-rank matrix, $\boldsymbol{\mathcal{W}}_i, i = 1, \ldots, r$ contains k columns, and each of these columns are treated as points in an m-dimensional space. If the perturbation (line 3) is not too large, then each vector of the matrix $\boldsymbol{W}$ will be close to a corresponding vector in the solution space of the unperturbed matrix. Therefore, we can define k such groups $C_1, \ldots, C_k$.

In order to cluster the r NMF solutions, we use a variation of k-means clustering [27] to reorder the columns of $\boldsymbol{\mathcal{W}}_q, \forall q$, so that the first column of each matrix belongs to one cluster, all the second columns belong to another, and so on. The optimal reordering is the permutation that maximizes the cosine similarities (a metric to find the similarity between two non-zero vectors) between the columns of $\boldsymbol{\mathcal{W}}_q$ and the columns of the current centroid. The rows of each $\boldsymbol{\mathcal{H}}_q$ are simultaneously reordered so that the product $\boldsymbol{\mathcal{W}}_q \boldsymbol{\mathcal{H}}_q$ is invariant. Our clustering is detailed later in section 4.3.

In order to determine the correct rank k_{opt}, we compute $\boldsymbol{s}_k$ and $\boldsymbol{e}_k$ at each k. Here, $\boldsymbol{s}_k$ (line 9) quantifies the quality of the clusters while $\boldsymbol{e}_k$ (line 10) quantifies the quality of reconstruction, which is the relative error defined as $\frac{\|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F}{\|\boldsymbol{A}\|_F}$. Using $\boldsymbol{s}_k$ and $\boldsymbol{e}_k$, *optRank* (line 12) determines the number of latent features, k_{opt}, in the input matrix, $\boldsymbol{A}$. In the case $k > k_{opt}$, the k clusters are not separated well and the average cluster quality $\boldsymbol{s}$ will be low (bad). On the other hand, for $k < k_{opt}$, the reconstruction error ($\boldsymbol{e}$) will be too high since the product of $\widetilde{\boldsymbol{W}}[k]$ and $\widetilde{\boldsymbol{H}}[k]$ (k columns and k rows, respectively) will not approximate close to $\boldsymbol{A}$. Therefore, $k = k_{opt}$ when the cluster quality $\boldsymbol{s}_k$ is high and relative error $\boldsymbol{e}_k$ is low. Distributed implementations of the procedures used in lines 3, 4, 6–9 are described later in section 4.5.

3.4 Optimization algorithm (AO-ADMM)

AO-ADMM [28] is an alternating optimization algorithm that solves the sub-problems in Eq. 2 using ADMM. Since these two sub-problems are identical

up to a transpose, we limit our discussion to the $\boldsymbol{H}$ sub-problem,

$$\underset{\boldsymbol{H} \geqslant 0}{\arg\min} \|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2 \,.$$

ADMM incorporates the non-negative constraint by splitting the variable $\boldsymbol{H}$ with an auxiliary variable $\boldsymbol{R}$, thus the sub-problem becomes,

$$\begin{aligned} \underset{\boldsymbol{H},\boldsymbol{R}}{\arg\min} \quad & \frac{1}{2}\|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2 + g(\boldsymbol{R}) \\ \text{subject to} \quad & \boldsymbol{H} = \boldsymbol{R} \,, \end{aligned}$$

where

$$g(\mathbf{X}) = \begin{cases} 0 & \text{if } \mathbf{X} \in \mathbb{R}_+^{k \times n} \\ \infty & \text{otherwise} \end{cases}$$

is an indicator function for the non-negative orthant. ADMM iterates updating the primal $\boldsymbol{H}$, auxiliary $\boldsymbol{R}$, and dual variables $\boldsymbol{S}$,

$$\begin{aligned} \boldsymbol{H} \quad & \leftarrow \underset{\boldsymbol{H}}{\arg\min} \frac{1}{2}\|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2 + \frac{\rho}{2}\|\boldsymbol{H} - \boldsymbol{R} + \boldsymbol{S}\|_F^2 \\ \boldsymbol{R} \quad & \leftarrow \underset{\boldsymbol{R}}{\arg\min}\, g(\boldsymbol{R}) + \frac{\rho}{2}\|\boldsymbol{H} - \boldsymbol{R} + \boldsymbol{S}\|_F^2 \\ \boldsymbol{S} \quad & \leftarrow \boldsymbol{S} + \boldsymbol{H} - \boldsymbol{R} \,, \end{aligned}$$

until converged. While ADMM is proven to converge under mild constraints, proof of convergence with AO-ADMM is less certain. In practice, AO-ADMM converges rapidly, especially when only a few iterations of the inner ADMM loops are executed.

4 DnMFk

We present the distributed algorithms for various components in generic *NMFk*, consisting of: perturbations, NMF algorithm, custom clustering, median factor matrices, and cluster stability (Silhouette statistics).

4.1 Distributed NMF

One of the main components of our implementation is the distributed NMF algorithm. Kannan et al. [32] proposed a state-of-the-art efficient distributed implementation of NMF, which we use in this paper. We follow their data partitioning scheme reported in [32]. Fig. 3 shows the distribution of an input matrix $\boldsymbol{A}$ onto a virtual 2D grid of processors. The output factors $\boldsymbol{W}$ and $\boldsymbol{H}$ are partitioned into a 1D grid. $\boldsymbol{A}$ is distributed across a $p_r \times p_c$ grid of processors, where $p = p_r \times p_c$. The resultant factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ are partitioned into $m/p \times k$ and $k \times n/p$ matrices. We use the open-source distributed AO-ADMM [31] for NMF minimization.

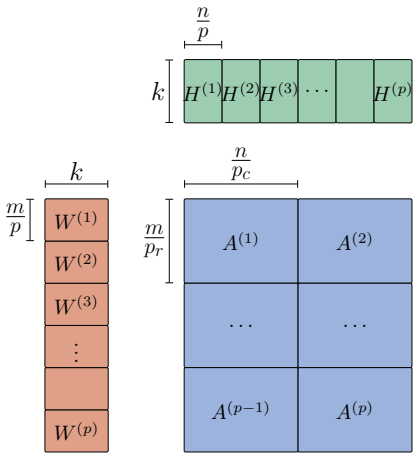

Fig. 3: The input matrix $\boldsymbol{A}$ distributed in a virtual 2D grid, the factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ in a 1D grid of processors.

Algorithm 2 Perturb($\boldsymbol{A}$, δ): Distributed resampling (perturbation) of input matrix

Require: $\boldsymbol{A}^{(i)} \in \mathbb{R}_{+}^{\frac{m}{p_r} \times \frac{n}{p_c}}$, $\delta \in \mathbb{R}_{+}$
1: Initialize $\boldsymbol{\Delta} \in \text{uniformRandom}(\dim(\boldsymbol{A}), \delta)$ ▷ Uniform random from $[1-\delta, 1+\delta]$
2: $\boldsymbol{A'}^{(i)} = \boldsymbol{A}^{(i)} \odot \boldsymbol{\Delta}$ ▷ Element-wise product
Ensure: $\boldsymbol{A'}^{(i)} \in \mathbb{R}_{+}^{\frac{m}{p_r} \times \frac{n}{p_c}}$ resampled representative of $\boldsymbol{A}^{(i)}$

4.2 Perturbation

For a fixed k, at each perturbation $\boldsymbol{A'}$ (total r) of $\boldsymbol{A}$, we resample $\boldsymbol{A'}$ from a distribution determined by $\boldsymbol{A}$. Algorithm 2 presents this resampling, termed *perturbation* hereafter. A common distribution is a uniform multivariate distribution that perturbs each element (for example, $\boldsymbol{\mathcal{A}}_q = \boldsymbol{A} \pm 10\%$, line 1). Since the matrix $\boldsymbol{A}$, and the resultant $\boldsymbol{\mathcal{A}}_q$ are distributed in a 2D grid of processors, the perturbation step does not require communication among different MPI ranks. The local (per MPI process) matrices are of size $\frac{m}{p_r} \times \frac{n}{p_c}$. A different random seed is used to generate uniform random numbers per MPI process. Hereafter, the MPI process specific matrices are preceded with a superscript; for example, $\boldsymbol{W}^{(i)}$ is the size $\frac{m}{p} \times k$ matrix on the i^{th} processor.

4.3 Distributed clustering with equal cluster size

We store all the low-rank factors (of r NMF perturbation runs), $\boldsymbol{W}$ and $\boldsymbol{H}$ at a given low rank approximation, k (see line 4 in Algorithm 1, $\boldsymbol{\mathcal{W}}$). Fig. 4 shows the distribution of all the $\boldsymbol{W}$ and $\boldsymbol{H}$ matrices at a given k for r perturbations. The $\boldsymbol{\mathcal{W}}$ is a distributed (row-wise, across p processors) cube with r number

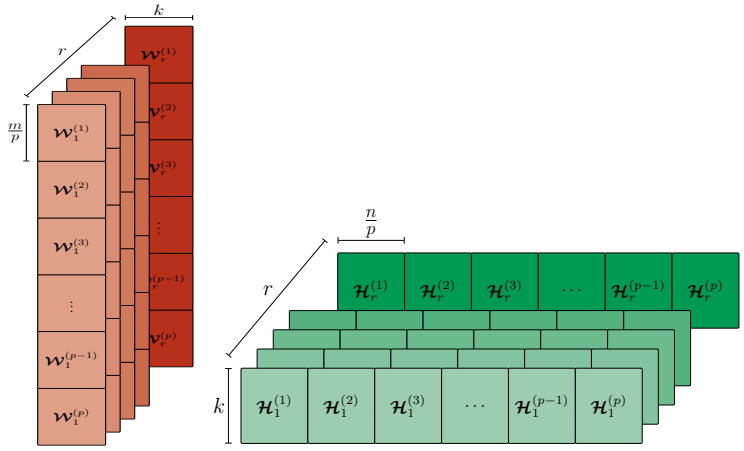

Fig. 4: The distributed $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{H}}$ across p processors for r perturbations, where $\boldsymbol{\mathcal{W}}^{(i)}$ is an $m/p \times k \times r$ cube and $\boldsymbol{\mathcal{W}}_q^{(i)}$ is an $m/p \times k$ left low-rank matrix corresponding to the q^{th} resampling. Similarly, $\boldsymbol{\mathcal{H}}^{(i)}$ is a cube of size $k \times n/p \times r$, and $\boldsymbol{\mathcal{H}}_q^{(i)}$ is a $k \times n/p$ right low-rank factor corresponding to the q^{th} resampling.

of slices, where each slice represents a perturbation ($\boldsymbol{\mathcal{W}}_q$). Similarly, $\boldsymbol{\mathcal{H}}$ cube is a distributed column-wise. Since the cube is distributed, each MPI process has access to $\boldsymbol{\mathcal{W}}^{(i)}$ with $m/p \times k \times r$ dimensions. In fact, the non-distributed $\boldsymbol{\mathcal{W}}$ cube is of size $m \times k \times r$, where the distributed slice per process ($\boldsymbol{\mathcal{W}}_q^{(i)}$) is of size $m/p \times k$. Similarly, $\boldsymbol{\mathcal{H}}$ cube is of size $k \times n \times r$, where the distributed slice per process ($\boldsymbol{\mathcal{H}}_q^{(i)}$) is of size $k \times n/p$.

Algorithm 3 describes our custom clustering (a variation of k-medians clustering) technique in a distributed manner. Note that the clustering actually rearranges the columns of $\boldsymbol{\mathcal{W}}$. The reordering happens in a distributed manner per MPI process on local matrices $\boldsymbol{\mathcal{W}}_q^{(i)}$ and $\boldsymbol{\mathcal{H}}_q^{(i)}$ across all perturbations. We initialize centroid ($\boldsymbol{M}^{(i)}$), with the solution of the first perturbation (line 1). We iterate the clustering algorithm for *n_iter* (lines 2–14) or the convergence criteria (stable clusters) is met. The distance function we are using here is the cosine similarity, which measures the cosine of the angle between two vectors projected in a multi-dimensional space [36] (per i^{th} processor, $\boldsymbol{\mathcal{D}}_q^{(i)}$ at line 4) between the centroids and the NMF solutions ($\boldsymbol{\mathcal{W}}_q^{(i)}$, $q = 1, \ldots, r$). With MPI all_reduce, we measure the global cosine similarity ($\boldsymbol{\mathcal{G}}$, line 6)) across all the processors in the MPI communication world. We have r such similarity matrices (for r perturbations), all of which are stored in a cube ($\boldsymbol{\mathcal{G}}$) with dimensions $k \times k \times r$. This $\boldsymbol{\mathcal{G}}$ cube is same across all MPI processes.

In clustering the columns of $\boldsymbol{W}$, we employ a greedy approach (line 8) on the cosine similarities. For each cosine similarity matrix (for each value of q, lines 7–11), we determine the column of $\boldsymbol{W}$ that has a maximum cosine similarity with the columns of the current centroid. This greedy algorithm

Algorithm 3 $\boldsymbol{\mathcal{W}'}^{(i)}, \boldsymbol{\mathcal{H}'}^{(i)}$ = customCluster($\boldsymbol{\mathcal{W}}^{(i)}, \boldsymbol{\mathcal{H}}^{(i)}$) – distributed clustering to reorder the columns of factors.

Require: $\boldsymbol{\mathcal{W}}^{(i)} \in \mathbb{R}_+^{m/p \times k \times r}$ and $\boldsymbol{\mathcal{H}}^{(i)} \in \mathbb{R}_+^{k \times n/p \times r}$ are processor specific cubes

1: $\boldsymbol{M}^{(i)} = \boldsymbol{\mathcal{W}}_1^{(i)}$ ▷ initialize centroid ($\boldsymbol{M}$) on i^{th} processor
2: **for** *iter* **in** 1 to *n_iter* **do**
3: **for** q **in** 1 to r **do** ▷ for each perturbation
4: $\boldsymbol{\mathcal{D}}_q^{(i)} = {\boldsymbol{M}^{(i)}}^T \times \boldsymbol{\mathcal{W}}_q^{(i)}$ ▷ compute partial similarity $\boldsymbol{\mathcal{D}}$ on i^{th} processor
5: **end for** ▷ $\boldsymbol{\mathcal{D}}^{(i)}$ is a $k \times k \times r$ cube per processor
6: $\boldsymbol{\mathcal{G}} = \sum_{i=1}^{p} \boldsymbol{\mathcal{D}}^{(i)}$ ▷ compute total similarity $\boldsymbol{\mathcal{G}}$ on all processors with all_reduce
7: **for** q **in** 1 to r **do** ▷ for each perturbation
8: $\boldsymbol{porder}$ = permOrder($\boldsymbol{\mathcal{G}}^{(q)}$) ▷ Fig. 5
9: $\boldsymbol{\mathcal{W}'}_q^{(i)} = \boldsymbol{\mathcal{W}}_q^{(i)}[\boldsymbol{porder}]$
10: $\boldsymbol{\mathcal{H}'}_q^{(i)} = \boldsymbol{\mathcal{H}}_q^{(i)}[\boldsymbol{porder}]$
11: **end for**
12: $\widetilde{\boldsymbol{W}}^{(i)}, \widetilde{\boldsymbol{H}}^{(i)}$ = mediansWH($\boldsymbol{\mathcal{W}'}^{(i)}, \boldsymbol{\mathcal{H}'}^{(i)}$) ▷ computes medians of clusters on i^{th} processor
13: $\boldsymbol{M}^{(i)} = \widetilde{\boldsymbol{W}}^{(i)}$ ▷ update centroid on i^{th} processor
14: **end for**

Ensure: $\widetilde{\boldsymbol{W}}^{(i)} \in \mathbb{R}_+^{\frac{m}{p} \times k}$, $\widetilde{\boldsymbol{H}}^{(i)} \in \mathbb{R}_+^{k \times \frac{n}{p}}$ distributed in p processors and columns of each $\boldsymbol{\mathcal{W}'}_q$ are clustered, reordered $\boldsymbol{\mathcal{H}'}_q$.

$$\begin{bmatrix} 0.81 & 0.96 & 0.70 & 0.62 \\ 0.75 & 0.61 & 0.54 & \mathbf{0.97} \\ 0.95 & 0.78 & 0.71 & 0.65 \\ 0.62 & 0.69 & 0.92 & 0.44 \end{bmatrix}$$

(a) $k_j = 1$

$$\begin{bmatrix} 0.81 & \mathbf{0.96} & 0.70 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \\ 0.95 & 0.78 & 0.71 & 0.00 \\ 0.62 & 0.69 & 0.92 & 0.00 \end{bmatrix}$$

(b) $k_j = 2$

$$\begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \\ \mathbf{0.95} & 0.00 & 0.71 & 0.00 \\ 0.62 & 0.00 & 0.92 & 0.00 \end{bmatrix}$$

(c) $k_j = 3$

$$\begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & \mathbf{0.92} & 0.00 \end{bmatrix}$$

(d) $k_j = 4$

Fig. 5: The greedy approach (line 8) to determine the row and column indices of maximum elements in a $\boldsymbol{\mathcal{G}}_q$ matrix for r perturbations. The example describes when $k = 4$ (a) in the first iteration (k_j), the maximum cosine similarity is in 2^{nd} row and 4^{th} column, in bold. For the permutation index, we consider the column, thus $\boldsymbol{porder} = [4]$. (b) we ignore those columns and rows by assigning all the corresponding elements to zero. The next maximum is in 1^{st} row and 2^{nd} column, thus $\boldsymbol{porder} = [4, 2]$ (c) the process is repeated until we find the final maximum, $\boldsymbol{porder} = [4, 2, 1]$ and (d) shows all previous maxima indices are being zeroed out, while the current maximum value is in bold. Finally, the permutation indices for this example are $\boldsymbol{porder} = [4, 2, 1, 3]$. which are used to reorder the columns of $\boldsymbol{\mathcal{W}}_q^{(i)}$ and rows of $\boldsymbol{\mathcal{H}}_q^{(i)}$.

is iterated k times, and in each of these k iterations, we ignore the columns and rows of the previous maximum elements. Fig. 5 describes the process

of ignoring the rows and columns. The greedy method returns a $1 \times k$ vector ($\boldsymbol{porder}$) of permutation indices. Using these indices, the columns of $\boldsymbol{\mathcal{W}}_q^{(i)}$ and the rows of $\boldsymbol{\mathcal{H}}_q^{(i)}$ are reordered. Note that the reordering is per process, while the permutation indices are same across all the processes. Next, the centroids are updated using the median of the clusters and the process is repeated until converged.

4.4 Distributed median

The final factors ($\widetilde{\boldsymbol{W}}$, $\widetilde{\boldsymbol{H}}$) are the solution matrices at a given k, found through the median of all the r factors. Algorithm 4 shows the distributed implemen-

Algorithm 4 $\widetilde{\boldsymbol{W}}^{(i)}, \widetilde{\boldsymbol{H}}^{(i)}$ = mediansWH($\boldsymbol{\mathcal{W}'}^{(i)}, \boldsymbol{\mathcal{H}'}^{(i)}$) – Distributed median of factors

Require: $\boldsymbol{\mathcal{W}'}^{(i)} \in \mathbb{R}_+^{m/p \times k \times r}$ and $\boldsymbol{\mathcal{H}'}^{(i)} \in \mathbb{R}_+^{k \times n/p \times r}$ are the processor specific cubes
1: $\widetilde{\boldsymbol{W}}^{(i)}$ = median($\boldsymbol{\mathcal{W}'}^{(i)}$) ▷ median along perturbation axis on i^{th} processor
2: $\widetilde{\boldsymbol{H}}^{(i)}$ = median($\boldsymbol{\mathcal{H}'}^{(i)}$) ▷ median along perturbation axis on i^{th} processor
Ensure: $\widetilde{\boldsymbol{W}}^{(i)} \in \mathbb{R}_+^{m/p \times k}$, $\widetilde{\boldsymbol{H}}^{(i)} \in \mathbb{R}_+^{k \times n/p}$ distributed in p processors

tation for finding the median factors using the processor specific $\boldsymbol{\mathcal{W}'}^{(i)}$ and $\boldsymbol{\mathcal{H}'}^{(i)}$ cubes. We find the median across the columns along the perturbation axis (which is the r NMF perturbations). Therefore, we gather all rows of a given column across different NMF runs to get $\widetilde{\boldsymbol{W}}^{(i)}$ (line 1). Similarly, we measure $\widetilde{\boldsymbol{H}}^{(i)}$ (line 2) per processor. Since both the low rank factor matrices are distributed in 1D processor grid, without partitioning on the k dimension, each MPI process computes the local median factors. Therefore, the communication overheads among MPI processes is zero. This advantage is because of our distribution strategy, where we distribute $\boldsymbol{\mathcal{W}}$ row-wise and $\boldsymbol{\mathcal{H}}$ column-wise.

4.5 Distributed silhouette statistics

Silhouette statistics is a measure of the stability of clusters, which helps in determining the correct k, the number of latent features. Silhouette statistic returns r data points (one per perturbation) and k clusters. We then measure the mean Silhouette width at a given k, which results in a scalar quantity that ranges in the interval [-1, 1]. Similarly, we can find the minimum Silhouette width, which in addition to the relative error and mean Silhouette width helps in determining the correct k. In order to measure the stability, we rely only on the set of left low rank factors ($\boldsymbol{\mathcal{W}}$).

Algorithm 5 describes the distributed implementation of Silhouette statistic for each data point. In estimating the Silhouette statistic, we measure two

Algorithm 5 $\boldsymbol{s}$ = clusterStability($\boldsymbol{\mathcal{W}'}^{(i)}$) – Distributed cluster stability through *Silhouette* analysis

Require: The clustered left low-rank factors, $\boldsymbol{\mathcal{W}}^{(i)} \in \mathbb{R}_{+}^{\frac{m}{p} \times k \times r}$ distributed row-wise

1: **for** k_α **in** 1 to k **do**
2: $\quad \boldsymbol{U}^{(i)} = \text{getMat}(\boldsymbol{\mathcal{W}'}^{(i)}, k_\alpha)$ ▷ gets $m/p \times r$ matrix for a given cluster
3: $\quad \boldsymbol{\mathcal{D}}_{k_\alpha}^{(i)} = {\boldsymbol{U}^{(i)}}^T \times \boldsymbol{U}^{(i)}$ ▷ Matrix-matrix product
4: **end for** ▷ $\boldsymbol{\mathcal{D}}^{(i)}$ is $r \times r \times k$ tensor of partial similarities on i^{th} processor.
5: $\boldsymbol{\mathcal{G}} = \sum_{i=1}^{p} \boldsymbol{\mathcal{D}}^{(i)}$ ▷ compute total similarity $\boldsymbol{\mathcal{G}}$ on all processors with all_reduce
6: $\boldsymbol{I} = \text{mean}(\boldsymbol{\mathcal{G}})$ ▷ mean along one perturbation axis for k clusters, $\boldsymbol{I}$ is $r \times k$
7: **for** k_α **in** 1 to k **do**
8: $\quad \boldsymbol{U}^{(i)} = \text{getMat}(\boldsymbol{\mathcal{W}'}^{(i)}, k_\alpha)$ ▷ gets $m/p \times r$ matrix for a given cluster
9: $\quad$ **for** k_β **in** 1 to k **do**
10: $\quad\quad$ **if** k_α != k_β **then**
11: $\quad\quad\quad \boldsymbol{V}^{(i)} = \text{getMat}(\boldsymbol{\mathcal{W}'}^{(i)}, k_\beta)$ ▷ gets $m/p \times r$ matrix for a given cluster
12: $\quad\quad\quad \boldsymbol{\mathcal{D}}_{k_\beta}^{(i)} = {\boldsymbol{U}^{(i)}}^T \times \boldsymbol{V}^{(i)}$ ▷ $r \times r \times (k-1)$
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad \boldsymbol{\mathcal{Z}}_{k_\alpha} = \sum_{i=1}^{p} \boldsymbol{\mathcal{D}}_{k_\beta}^{(i)}$ using all_reduce ▷ i^{th} processor has $r \times r \times (k-1)$ cube
16: $\quad \boldsymbol{Y} = \text{mean}(\boldsymbol{\mathcal{Z}}_{k_\alpha})$ ▷ $r \times (k-1)$
17: $\quad \boldsymbol{J}_{k_\alpha} = \min(\boldsymbol{Y})$
18: **end for** ▷ i^{th} processor owns a copy of $\boldsymbol{J}$ of size $r \times k$
19: $\boldsymbol{s}_k = \text{mean}\left(\text{mean}\left(\frac{\boldsymbol{J} - \boldsymbol{I}}{\max(\boldsymbol{J}, \boldsymbol{I})}\right)\right)$ ▷ $\boldsymbol{s}_k \in [-1, 1]$, the inner mean is column-wise

Similarly we can find the minimum Silhouette width

Ensure: $\boldsymbol{s}_k \in \mathbb{R}$

different parameters: the average similarity of all the data points within a given cluster ($\boldsymbol{I}$, see lines 1–6) and the average dissimilarity between clusters ($\boldsymbol{J}$, see lines 7–18). Specifically, we define $\boldsymbol{I}$ as the average distance between all the data points in a given cluster, and $\boldsymbol{J}$ is the smallest average distance of a data point from all the points in all the clusters except the cluster that the current data point is a member. For distance one can use any distance metric such as the Euclidean, Manhattan, or cosine distances. We chose to use the cosine similarity distance. Using $\boldsymbol{I}$ and $\boldsymbol{J}$, we estimate the average Silhouette value at a given k (see line 19).

The r number of distributed left low rank matrices, $\boldsymbol{\mathcal{W}}$, is the input to the distributed Silhouette algorithm. Note that the input cube, $\boldsymbol{\mathcal{W}}$, is clustered by Algorithm 3. We first compute $\boldsymbol{I}$, where we distribute the cosine similarity between different data points in a cluster. Process i calculates the local cosine similarity ($\boldsymbol{\mathcal{D}}^{(i)}$) between the data points for all the clusters (lines 1–3). Using the local cosine similarity, compute the global similarity ($\boldsymbol{\mathcal{G}}^{(i)}$) from all processes (line 5) using MPI all_reduce. All the MPI processes have an exact same copy of the resultant $\boldsymbol{\mathcal{G}}^{(i)}$, which is of size $r \times r \times k$. We then compute the cohesion between data points in a given cluster ($\boldsymbol{I}$, line 6) as the mean, the result of which is an $r \times k$ matrix.

Next, we calculate $\boldsymbol{J}$ for a data point in each cluster. We measure the cosine similarity between the current data point and the rest of the data points in

other clusters, where the current data point is not a member. Similarly to $\boldsymbol{I}$, we calculate the process-specific cosine similarity and the final cosine similarity ($\boldsymbol{\mathcal{Z}}^{(i)}$) (line 15), which is of size $r \times r \times k - 1$. We compute the minimum average distance for a data point in a given cluster ($\boldsymbol{J}$), which is of size $r \times k$. Finally, we calculate the average *Silhouette width* (line 19) from the Silhouette values of each data point in the cluster, which ranges in [–1, 1]. Note the same copy of $\boldsymbol{s}$ is found across all the processes, we return $\mathbf{S}$ from one of the processes. Similarly, we compute the reconstruction error $\boldsymbol{e}_k$ at given k. This procedure is repeated for different k values of NMFk algorithm, which results in average Silhouette width and reconstruction errors. The k at which these two statistics have clear separation is treated as the correct number of latent features.

5 Cost analysis

We analyze the compute, communication, and memory costs of *DnMFk* algorithm. Although the algorithm is distributed across multiple processors, that does not achange the actual complexity of the algorithm. However, we analyze the complexity per processor. Note, Algorithm 1 calls the distributed parts.

5.1 Time analysis

The computational complexity of *DnMFk* is the sum of the time complexities of all the function calls in Algorithm 1.

The function Perturb in Algorithm 2 (line 3 in Algorithm 1) takes $O(\frac{mn}{p})$ per call, and $O(\frac{mnr}{p})$ for r perturbations.

The time for running the distributed version of *NMF* from [32] on line 4 of Algorithm 1 for a fixed k is $O(\frac{mnk}{p} + \frac{(m+n)k^2}{p}) + \mathrm{C}_{\mathrm{ADMM}}(\frac{m+n}{p}, k)$ [32], where $\mathrm{C}_{\mathrm{ADMM}}(m, n)$ denotes the time for solving a Non-negative Least Squares (NLS) problem using ADMM optimization.

Algorithm 3, customCluster (line 6 of Algorithm 1) finds the inner product for cosine similarity, multiplying matrices of sizes $k \times \frac{m}{p}$ and $\frac{m}{p} \times k$ in line 4 takes $O(\frac{k^2 m}{p})$ time. The time taken for all_reduce (line 6) reduction operation across p processors on $k \times k \times r$ cube is $O(k^2 rp)$. The greedy algorithm to find the maxima of a $k \times k$ matrix (lines 8) takes $O(k^2)$, which is iterated for k times, thus the complexity for permOrder is $O(k^3)$. Time taken for reordering the columns of $\boldsymbol{\mathcal{W}}^{(i)}$ and rows of $\boldsymbol{\mathcal{H}}^{(i)}$ (lines 9–10) is $O(\frac{m+n}{p}k)$. The mediansWH (Algorithm 4) takes $O(\frac{m}{p}kr + \frac{n}{p}kr) = O(\frac{m+n}{p}kr)$ time, that includes the time to find the factor matrices, $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{H}}$. The total time for running Algorithm customCluster: $O\Big(n\_iter\,\big(\frac{k^2 mr}{p} + k^2 rp + k^3 r + \frac{m+n}{p}kr + \frac{m+n}{p}kr\big)\Big)$. Assuming $m \geq n$ the time taken for customCluster is $O\Big(n\_iter\,\big(\frac{k^2 mr}{p} + k^2 rp + k^3 r\big)\Big)$.

Algorithm 5 (clusterStability) requires $O(\frac{r^2m}{p})$ time for each execution of line 3, or in total $O(\frac{r^2mk}{p})$ time for the loop on lines 1–4. The time for all_reduce (line 5) reduction operation is $O(r^2kp)$. To calculate the mean (line 6) is $O(r^2k)$. Similarly, the time for the loop on lines 7–18 is dominated by the time to execute line 12, which is $O(\frac{r^2m}{p})$, so the total time for the loop is $O(\frac{r^2k^2m}{p})$. The time taken for all_reduce (line 15) is $O(r^2k^2p)$. For the mean (line 16) is $O(r^2k^2)$ and the min (line 17) is $O(k^2r)$. The total time of clusterStability: $O\left(\frac{r^2mk}{p} + \frac{r^2k^2m}{p} + r^2k^2p + r^2k^2 + k^2r\right)$. Since the line 12 is more dominating, the total time for Algorithm *clusterStability* is $O(\frac{r^2k^2m}{p})$.

Adding together all time bounds and taking into account that $k \leq \min\{m, n\}$ and the number of perturbations of *NMF*, we get a bound on the time for one k-iteration of Algorithm 1 as $O\left(\frac{mnkr}{p} + r\mathrm{C}_{\mathrm{ADMM}}\left(\frac{m+n}{p}, k\right) + n\_iter\left(\frac{k^2mr}{p} + k^2rp + k^3r\right) + \frac{r^2k^2m}{p}\right)$. Assuming the number n_iter of iterations in customCluster does not exceed r, which is true in our case, and setting $k_l = 1$ to simplify the formula, we get a time bound for *DnMFk* as $O\left(\frac{mnk_u^2r}{p} + \frac{r^2k_u^3m}{p} + k_u^3r^2p + k_u^4r^2 + k_ur\mathrm{C}_{\mathrm{ADMM}}\left(\frac{m+n}{p}, k_u\right)\right)$.

5.2 Communication cost

We have three occurrences of an all_reduce MPI operation, and use the cost model of [14], where an all_reduce operation on a vector of length n and p processors is $O(\log p + n)$. In line 6 of Algorithm 3, the length of the vector is k^2r and all_reduce is executed n_iter times, so the communication cost is $O(n\_iter(\log p + k^2r))$.

In Algorithm 5, we have all_reduce in lines 5 and 15, but the latter cost dominates due to the fact that the operation is executed k times. The cost for all k executions is $O(k \log p + k^2r^2)$.

Adding these communication cost bounds, we get a bound the three all_reduce calls as

$$O(n\_iter(k_u \log p + k_u^3r) + k_u^2 \log p + k_u^3r^2).$$

Finally, the communication cost for one call to Algorithm *DnMFk*, from [32], is $O(\sqrt{mnk^2/p})$, assuming $p \geq m/n$, and the algorithm is called $r(k_u - k_l + 1)$ times.

Assuming as before that $n\_iter \leq r$ and ignoring the logarithmic factors, which are of lower complexity, we get a final communication cost bound of

$$O(k_u^3r^2 + k_u^2r\sqrt{mn/p}) = O(k_u^2r(k_ur + \sqrt{mn/p})).$$

5.3 Memory analysis

We next analyze the memory needed to store the (local) dense matrices, $\boldsymbol{A}$, $\widetilde{\boldsymbol{W}}$, $\widetilde{\boldsymbol{H}}$, and dense cubes, $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{H}}$, at each processor. For $\boldsymbol{A}$, the space bound is mn/p words. Although we perturb $\boldsymbol{A}$ r times, we only store one perturbation ($\boldsymbol{\mathcal{A}}_q$) at a time, therefore the above memory bound for $\boldsymbol{A}$ does not depend on the number r of perturbations. For the output factor matrices ($\widetilde{\boldsymbol{W}}$, $\widetilde{\boldsymbol{H}}$), the memory requirement is $\frac{m+n}{p}k$ words. We store these factor matrices for r perturbations therefore the cubes ($\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{H}}$) need $\frac{m+n}{p}kr$ words. The memory required to store the temporary matrices is $2r^2k$ for $\boldsymbol{\mathcal{G}}$ and $\boldsymbol{\mathcal{Z}}$ (in Algorithm 3), and $2rk$ for $\boldsymbol{I}$ and $\boldsymbol{J}$ (in Algorithm 5). Note that the temporary matrices have the same size across all processors, therefore, we do not divide by p. Putting everything together, we get a bound on the memory per processor

$$\frac{mn + (m+n)kr}{p} + (r+1)2rk.$$

For small values of k and p compared to m and n, which is typical for most real-application matrices and machines we have dealt with, the memory needed per processor is approximately $\frac{mn+(m+n)kr}{p}$ words.

6 Experiments

Our experiments are two types: first, we validate the correctness of *DnMFk* on 100 synthetic matrices with predetermined k, and on the well-know *Swimmer* dataset; second, we evaluate the time performance of *DnMFk*. Our scaling experiments are conducted on two dense synthetic matrices. We then perform a scaling study on custom clustering and Silhouette statistics.

6.1 Parameter settings

We run the experiments on the HPC cluster *Grizzly*, located at Los Alamos National Laboratory (LANL). *Grizzly* has Intel Xeon Broadwell (E5-2695v4) processors with a total of 1490 compute nodes, where each node has 18-core dual socket Ivy Bridge processor. Each of the 36 processors has a clock speed of 2.1 GHz with a private $L1$ and $L2$ caches of sizes 64KB and 256KB. Both the sockets share an $L3$ cache of size 45MB, where each node contains 128GB of memory. Grizzly uses Tri-Lab Operating System Stack (TOSS) version 3, while the interconnect is Intel *OmniPath* that uses a fat tree topology.

Our source code is in C++, where we extended the open-source distributed NMF library, Planc [31]. Planc depends on Armadillo [50] (a C++ linear algebra library), LAPACK, and BLAS. While Planc supports both dense and sparse matrices, in this paper, our *DnMFk* implementation supports dense matrices, and we leave the sparse implementations for future releases. We use

the default GNU C++ (v7.4.0) compiler and the OpenMPI (v2.1.2) library available on Grizzly.

We use AO-ADMM for all the experiments in this paper, however, DnMFk works for any of the distributed implementations of MU, BPP, and HALS, e.g., ones contained in the Planc library. In our experiments, we randomly initialize both $\boldsymbol{W}$ and $\boldsymbol{H}$ using a different seed at each perturbation. Every processor uses a unique prime seed to generate the random matrix and the noise for perturbation. In scheduling the MPI jobs, we follow the constraint $p_r \geq p_c$ where the matrices are spread across a virtual $2D$ grid topology. To the best of our knowledge, *DnMFk* is the first large scale NMF algorithm that determines the number of latent features (k) in the dense input data, therefore, we can not compare the performance with other similar methods.

6.2 Model determination

6.2.1 Synthetic data

We generate synthetic data, say $\boldsymbol{A}_{m\times n}$, with known number of latent features k, and selected dimensions m and n. The columns of $W_{m\times k}$ are feature vectors, where each vector is sampled from a normal distribution with random mean and variance. The elements of the mixing matrix, $H_{k\times n}$ are sampled from exponential distribution with scale 1. The matrix $\boldsymbol{A}$ is generated as a product of $\boldsymbol{W}$ and $\boldsymbol{H}$ with added uniform noise and a standard deviation of 10%.

With this procedure, we generate 100 matrices, which belong to four sets of dimensions: a) 576×384 b) 1024×256 c) 2160×1080 d) 1440×720. Each of the 100 matrices have a randomly chosen k from 2 to 25. DnMFk is evaluated on how accurately it finds the correct number of hidden features of these 100 matrices. The columns of $\boldsymbol{W}$ span the feature space and the quality of their reconstruction is important when estimating the accuracy of the algorithm. In order to compare the predicted $\boldsymbol{W}$ with that of the original one, we use Pearson correlation coefficient [10], which determines the correctness of the prediction.

We analyze the correctness of *DnMFk* model determination on two data matrices (*Data 1* and *Data 2*) selected randomly from the 100 matrices. *Data 1* is of size 2160×1080 with predetermined $k = 7$ and *Data 2* is (1024×256) with $k = 17$. Fig. 6(**a**) and 6(**b**) shows the identification of hidden features through Silhouette statistics. To determine the correct number of features, *DnMFk* needs to compute the accuracy of reconstruction (relative error), and the minimum and the average Silhouette width at a given k. For k clusters, we have k Silhouettes. The mean and minimum width are used to quantify the stability of the clusters. At the correct k, the Silhouettes are expected to be close to 1 and the corresponding *relative error* to be low. In the case of $k > k_{opt}$, NMF actually over-fits the original data with noise, thereby results in less stable clusters, hence the drop in average Silhouette width. We can see the sudden drop in *Data 1* at $k = 8$ and for *Data 2* at $k = 18$. At the correct

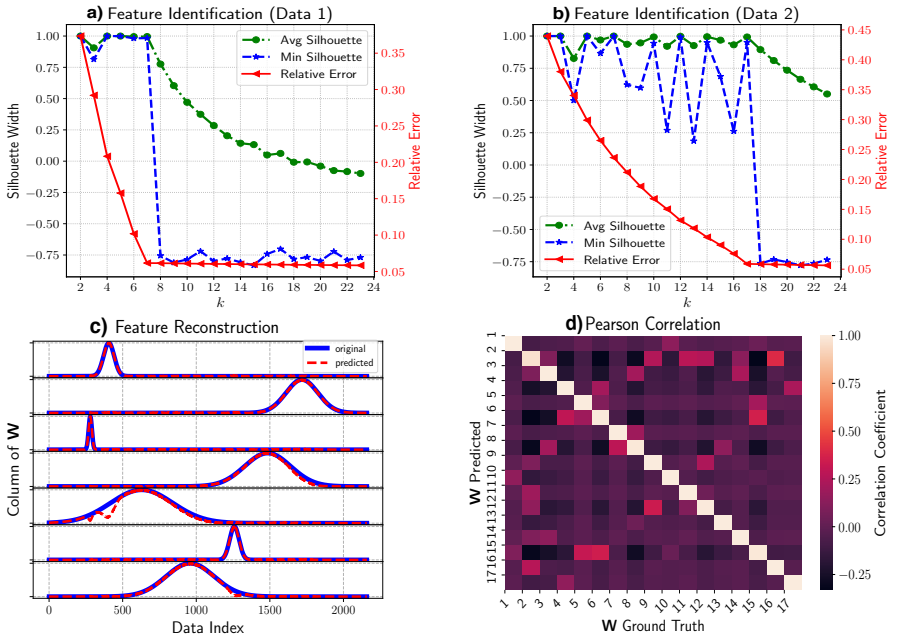

Fig. 6: Estimation of number of hidden features for sets **a**) *Data 1* (k = 7) and **b**) *Data 2* (k = 17), through Silhouette analysis. The bottom two plots demonstrate the $\boldsymbol{W}$ reconstruction. **c**) columns of original $\boldsymbol{W}$ (blue line) and predicted $\boldsymbol{W}$ (red dashed) for *Data 1*. **d**) Pearson correlation between the columns of ground truth $\boldsymbol{W}$ with that of reconstructed one for *Data 2*.

value of k, the distance between average Silhouette width and the relative error is maximum. Thus the analysis helps to find the number of hidden features in the two data sets to be 7 and 17, respectively, which agrees with the ground truth.

The significance of NMF is in its physically interpretable features after factorization. Hence, reconstructing the features is as important as finding how many hidden components are present in the data. For lower actual k, we can visualize the components of the factorized data with that of groundtruth, which is shown in Fig. 6(**c**). Seven columns of W obtained from *DnMFk* (dashed red curves) are plotted with Ground truth W (solid blue lines). For the *Data 2* with larger ground truth k(=17), as shown in 6(**d**) we demonstrated the better reconstruction of latent features by *DnMFk* using Pearson correlation coefficient. The results indicate a close match. We found that the average Pearson coefficient for the 100 matrices is 0.995 with a standard deviation of 0.003, which demonstrates the ability of DnMFk to accurately recognize the number of hidden features.

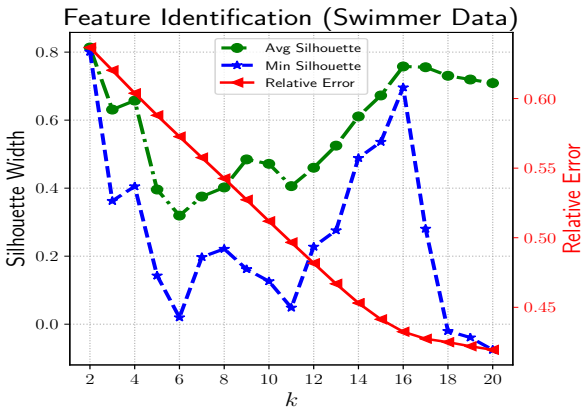

Fig. 7: Estimating the number of hidden features in *Swimmer* data by relative error and Silhouette statistics

6.2.2 Latent feature identification in Swimmer dataset

We applied *DnMFk* on *Swimmer* dataset [57] to find the number of hidden features. *Swimmer* is a well-known dataset for which the standard rank and the non-negative rank are unequal, because of which, finding the number of hidden components is non trivial. The data is an image library, which consists of 256 images of 32 x 32 pixels, each image contains torso at the center with different limbs positions, is converted to a column vector. Size of this dataset is 1024×256. DnMFk correctly identifies 16 latent features in the *Swimmer* dataset that corresponds to the limbs' positions, which is same as the number of latent features found with the Bayesian approaches [57]. Identifying the latent features in large scale real-world datasets is one of the future directions to explore. *DnMFk* takes 220 seconds on a Grizzly node with 16×16 processor grid while exploring through a range k values from 2 to 20.

6.3 Scalability

In the second set of experiments, we evaluate the scalability of *DnMFk*, using both strong and weak scaling. We analyze the scaling behavior with respect to the latent dimension, k. For both the scaling experiments, we keep the number of processors/cores to be in {36, 144, 324, 576, 900}, which corresponds to {1, 4, 9, 16 and 25} nodes. Since we study the scaling behavior, we fix the number of perturbations to 10, the number of AO-ADMM iterations to 5 for both $\boldsymbol{W}$ and $\boldsymbol{H}$ updates, while we execute NMF for 100 iterations.

We report the performance on compute and communication costs of *DnMFk*. The computation costs are: **gram** – to calculate the local computations of gram matrix ($\mathbf{W}^T\mathbf{W}$ or $\mathbf{H}\mathbf{H}^T$), which is of size $k \times k$; **mm** – matrix-matrix multiplications with the local (MPI rank specific) input matrix and factor ma-

trices; and **nnls** – solving the non-negative least squares. The communication cost include: **all_gather** – time taken for global matrix-matrix multiplications while distributing the results across all processors; **all_reduce** time required to compute global *gram* matrices; and **reduce_scatter** – to compute global matrix-matrix multiplications and scatter them across multiple ranks.

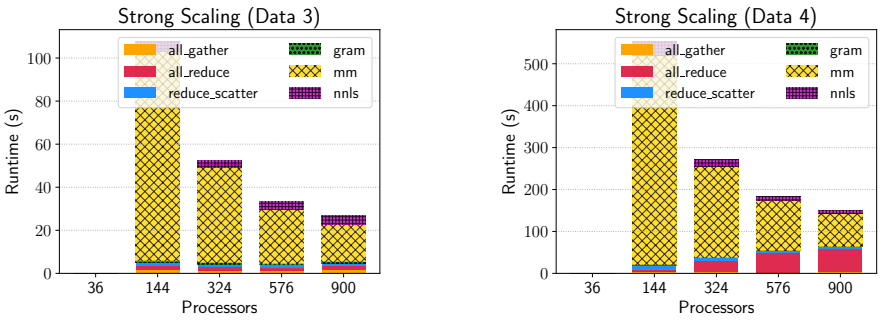

Fig. 8: Strong Scaling – The runtimes across 10 perturbations of *DnMFk* for *Data 3* (left) and *Data 4* (right). The number of latent features in both the matrices are fixed at $k = 10$ and $k = 50$.

6.3.1 Strong scaling

We conduct the strong scaling experiments on two random matrices of different sizes. Both the matrices are synthetically generated (using the procedure in section 6.2.1), and dense. *Data 3* matrix is of size 57600×38400, *Data 4* dimensions are 129600×51840, and both matrices are generated from a uniform random distribution while adding Gaussian noise. For both datasets, we fix the number of hidden features (k) to be 10 and 50 respectively. Each of the data matrices is too large to fit in the memory of a single node, therefore we report the strong scaling results on {4, 9, 16 and 25} nodes.

Table 2: The run times (in seconds) per perturbation (average) and across all 10 perturbations (total) with 100 NMF iterations of *DnMFk* for both the matrices in strong scaling.

Cores	**Data 3**		**Data 4**	
	Perturb	**Total**	**Perturb**	**Total**
144	12.58	125.84	63.16	631.69
324	06.19	061.93	30.98	309.87
576	03.94	039.42	21.08	210.86
900	03.31	033.16	16.93	169.40

Fig. 8 shows the strong scaling results for both matrices. Table 2 shows the average per perturbation runtimes at different number of processors. The parallel speedups of 25 nodes over 4 nodes are 3.79 times for *Data 3* and 3.73 for *Data 4*. The results indicate that the scaling in both the matrices is almost linear. There exists unnecessary communication overheads (clearly evident in *Data 4*), which refrain *DnMFk* from reaching ideal scaling.

6.3.2 Weak scaling

The number of processors for the weak scaling experiments are the same as for the strong scaling experiments, 36, 144, 324, 576, 900, which correspond to 1, 4, 9, 16 and 25 nodes with 36 cores per node, respectively. In the weak scaling study, we run experiments on two different sets of data, with matrices of dimensions as follows: for *Data 5* the matrices are 7200 × 7200 on 36 cores up to 36000 × 36000 on 900 cores with a fixed $k = 10$, and for *Data 6* the matrices vary from 10,800 × 10,800 up to 54,000 × 54,000 with a fixed $k = 50$, respectively. For both the matrices, the local matrices per core always have block dimensions of 1200×1200 (*Data 5*) and 1800×1800 (*Data 6*), respectively.

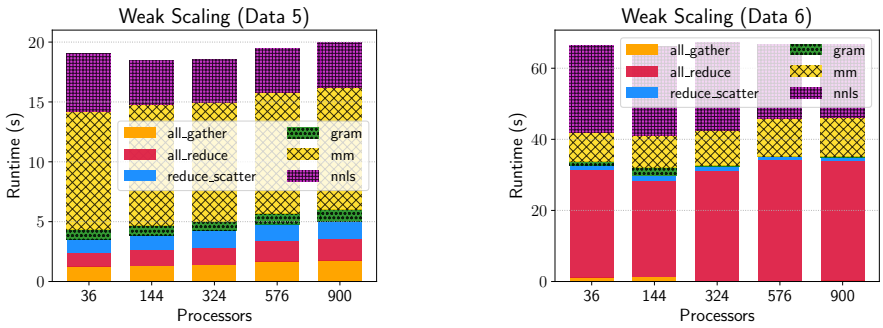

Fig. 9: Weak Scaling – The runtimess of *DnMFk* on *Data 5* (left) and *Data 6* (right). The latent feature are fixed for both the matrices as $k = 10$ and $k = 50$. *Data 5* dimensions are ranging from 7200 × 7200 on 36 cores up to 36000 × 36000 on 900 cores, that of *Data 6* are 10800 × 10800 to 54000 × 54000.

Fig. 9 shows the weak scaling results for both input matrices. The local input matrix dimensions are fixed as in the case of [32] so that the number of matrix multiplications per processor is fixed. Fig. 9 suggests that, in both matrices, most of the time is spent in matrix multiplications, while the runtime remains almost the same in all the processor configurations. The weak scaling results suggest that the scaling is almost linear with some negligible effects in communication times. In *Data 6*, all_reduce and **nnls** contribute significantly to the runtime, while the latter decreases for higher number of processors. This behavior is due to the fact that we are fixing the number of multiplications

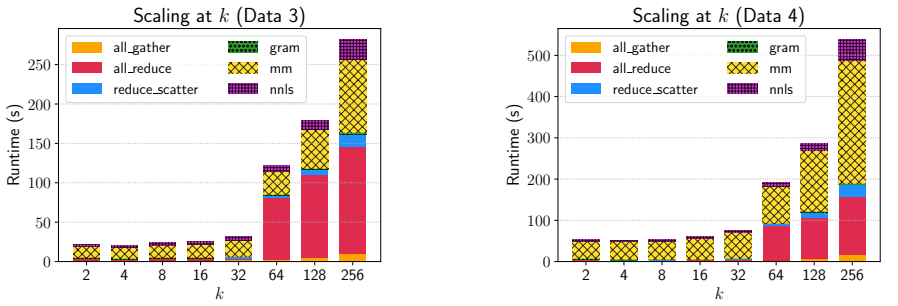

Fig. 10: k scaling at *900* processors using the strong scaling (*Data 3* and *Data 4*) matrices. For both the matrices the k values are changed as {2, 4, 8, 16, 32, 64, 128, 256} while the reported runtimes are for 10 perturbations of *DnMFk*.

only, rather than the size of the factor matrices, which helps in better scaling of **nnls** at higher number of processors.

6.3.3 Scaling at k

For both the datasets in strong scaling, we fix the number of processors to be 900 and vary k in order to study the effect of k on scaling. Fig. 10 shows the scaling results, where we vary k in {2, 4, 8, 16, 32, 64, 128, 256}. The results indicate similar trends in the execution times for both datasets. For k between 32 and 64, there is a sudden jump in the overall runtime, which is due to the fact that communication overheads increased significantly. These overheads are predominately resulting from all_reduce. However, if we neglect the smaller k values ({2, 4, 8, 16}), the scaling with k is nearly linear in terms of computation time.

In all the scaling experiments, the effect of communication at higher number of processors can be minimized with a reduced precision. Currently, we are using *double* procession making that *single floating* point can reduce the communication effects.

6.4 Performance of distributed clustering and silhouette

We conduct a different study on the performance of distributed clustering and Silhouette algorithms. Fig. 11 shows the speedups of both the clustering and silhouette calculation on 144, 324, 576 and 900 processors for both the datasets that are used in the strong scaling experiment. The reported speedups are to cluster the *DnMFk* factorized matrices from 10 perturbations. Therefore, the two input cubes for both clustering and Silhouettes (from *Data 3* and *Data 4*) are $57600 \times 10 \times 10$ and $129600 \times 50 \times 10$. The speedups are measured against the runtimes on single core. The maximum speedup for clustering *Data 3* factors is 115.98x on 900 processors while that of *Data 4* is 150.82x.

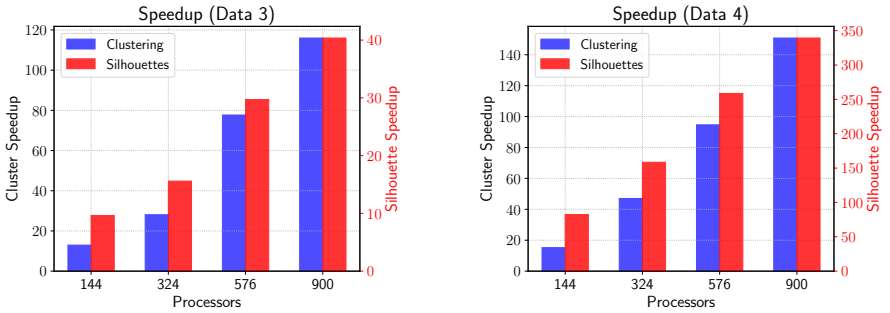

Fig. 11: Speedup – The speedup of *clustering* and *Silhouette* calculation across 10 perturbations of *DnMFk* for *Data 3* (left) and *Data 4* (right). The number of latent features in both the matrices are fixed at $k = 10$ and $k = 50$.

For Silhouettes, *Data 3* has 40.35x and *Data 4* contains 339.32x speedup. Table 3 (in Appendix) shows the corresponding runtimes of both clustering and Silhouette for these matrices. The performance results indicate significant gains as the number of cores increase. Although the performance gains are promising, the gains are far from the ideal speedups on these two matrices. The main reason for such performance trends is due to the fact that the input data $57600 \times 10 \times 10$ and $129600 \times 50 \times 10$ for the distributed clustering and silhouette algorithms is small compared to the number of processors. Note that NMF is a dimensionality reduction method, thus the resultant factor matrices $\mathbf{W}$ and $\mathbf{H}$ are much smaller than the size of input matrix, $\mathbf{A}$. In order to clearly study the performance of distributed clustering and silhouette, we conducted a different set of experiments with different data sizes, presented in Fig. 14 (in Appendix) and Table 4 (in Appendix). In summary, the clustering speedups on these datasets ($2097152 \times 10 \times 10$ and $1048576 \times 50 \times 10$) are 663.84x and 284.93x. The Silhouette speedups are 1187.98x and 594.34x, which shows that *DnMFk* clustering and Silhouettes scale better for large data.

We further study the effect of k on the performance of clustering and Silhouettes. Fig. 12 shows the performance trends for the above two matrices at 900 processors. We observe that the execution time increases with the increase in the k value. Similar experiments on different large datasets are in Fig. 15 (Appendix), that shows similar patterns. The results indicate that the execution time increases linearly with k for a fixed number of processors. Overall, our distributed implementations of custom K-medians clustering and Silhouette calculations scale for large data, and increasing k on higher number of MPI processors. Finally, our implementation of *DnMFk* scales nearly linearly at higher number of processors.

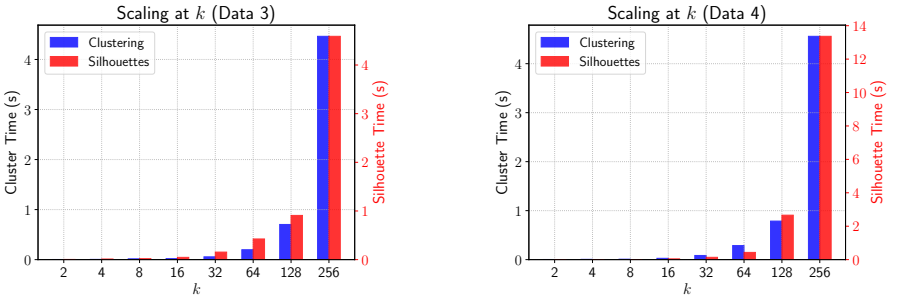

Fig. 12: k scaling for *clustering* and *Silhouette* – The runtimes of *clustering* and *silhouette* across 10 perturbations of *DnMFk* for *Data 3* (left) of size $57600 \times 10 \times 10$ and *Data 4* (right) of size $129600 \times 50 \times 10$. The number of latent features in both the matrices vary as $\{2, 4, 8, 16, 32, 64, 128, 256\}$.

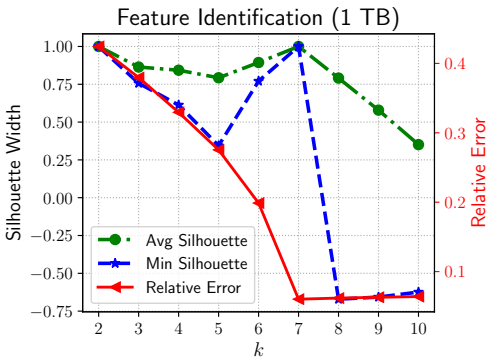

Fig. 13: Estimating the number of hidden features in $1TB$ data, *DnMFk* finds k as 7, which is same as the ground truth.

6.5 Model determination in large data

We determine the number of latent features in a single tera byte (TB) matrix with dimensions 16777216×8192 with $k = 7$ known latent features. We synthetically generate the $1\,TB$ matrix using the data generation algorithm from section 6.2.1. Fig. 13 shows the silhouette analysis used to determine the number of hidden feature, which shows that it finds that $k = 7$. Fig. 16 (in Appendix) identifies $k = 5$ in a dataset of 0.5 TB size. *DnMFk* takes 14 hours on 4096 processors to determine the number of hidden features in the $1TB$ matrix, while, for the 0.5 TB matrix, *DnMFk* takes 13.6 hours on 2048 processors. The average error in reconstruction for $1TB$ is 6.1% while that of half a TB is 5.8%. To the best of our knowledge, *DnMFk* is the first approach that determines the number of latent features in $1\,TB$ data.

7 Conclusions

We introduced *DnMFk*, a distributed non-negative matrix factorization with determination of the latent dimensionality of the data. The method for determining the number of latent features is based on resampling the initial data and comparing the accuracy and stability of a set of NMF solutions at each explored number of latent features, k. *DnMFk* requires building of a group of uniformly (or Poissonian, or half-Gaussian) distributed datasets with a mean same as the initial dataset. Our method needs custom k-means/medians clustering with same size clusters [1] and Silhouette statistics [49], both of which we developed in a distributed manner and integrated with the parallel NMF library introduced in [31].

DnMFk can determine the unknown number of latent variables in large datasets and extract them, which has not been addressed yet in previous parallel implementations of NMF. We provided the cost analysis for computation, communication time, and memory usage, as well as analysis of the accuracy and scalability of *DnMFk*. We demonstrated that our method is able to recover the latent dimensionality of synthetic and real datasets (which are dense) in a distributed and scalable manner. Our algorithm scales nearly linearly for large datasets, while the communication overheads, at higher number of processors, can be reduced through the reduction of double precision to single floating-point, one of the desired future endeavors. We found the number of latent features in a TB matrix. Moreover, in future, we propose to extend *DnMFk* to sparse matrices. We further apply *DnMFk* to non-negative tensor factorization (similar to [2]) to find the latent features in large tensors.

Acknowledgements This research used resources provided by the Los Alamos National Laboratory Institutional Computing Program, which is supported by the U.S. Department of Energy National Nuclear Security Administration under Contract No. 89233218CNA000001.

Funding

This study was funded by U.S. Department of Energy National Nuclear Security Administration under Contract No. DE-AC52-06NA25396 through LANL Laboratory Directed Research and Development (LDRD) grant 20190020DR.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Alexandrov, B.S., Alexandrov, L.B., Iliev, F.L., Stanev, V.G., Vesselinov, V.V.: Source identification by non-negative matrix factorization combined with semi-supervised clustering (2018). US Patent App. 15/690,176

2. Alexandrov, B.S., Stanev, V.G., Vesselinov, V.V., Rasmussen, K.Ø.: Nonnegative tensor decomposition with custom clustering for microphase separation of block copolymers. Statistical Analysis and Data Mining: The ASA Data Science Journal (2019)
3. Alexandrov, B.S., Vesselinov, V.V.: Blind source separation for groundwater pressure analysis based on nonnegative matrix factorization. Water Resources Research **50**(9), 7332–7347 (2014)
4. Alexandrov, L.B., Nik-Zainal, S., Wedge, D.C., Aparicio, S.A., Behjati, S., Biankin, A.V., Bignell, G.R., Bolli, N., Borg, A., Børresen-Dale, A.L., et al.: Signatures of mutational processes in human cancer. Nature **500**(7463), 415 (2013)
5. Alexandrov, L.B., Nik-Zainal, S., Wedge, D.C., Campbell, P.J., Stratton, M.R.: Deciphering signatures of mutational processes operative in human cancer. Cell reports **3**(1), 246–259 (2013)
6. Amari, S.i., Cichocki, A., Yang, H.H.: A new learning algorithm for blind signal separation. In: Advances in neural information processing systems, pp. 757–763 (1996)
7. Barlow, H.: Unsupervised learning. Neural Computation **1**(3), 295–311 (1989). DOI 10.1162/neco.1989.1.3.295. URL `https://doi.org/10.1162/neco.1989.1.3.295`
8. Battenberg, E., Wessel, D.: Accelerating non-negative matrix factorization for audio source separation on multi-core and many-core architectures. ISMIR pp. 501–506 (2009)
9. Baum, L.E., Petrie, T.: Statistical inference for probabilistic functions of finite state markov chains. The annals of mathematical statistics **37**(6), 1554–1563 (1966)
10. Benesty, J., Chen, J., Huang, Y., Cohen, I.: Pearson correlation coefficient. In: Noise reduction in speech processing, pp. 1–4. Springer (2009)
11. Beutel, A., Talukdar, P.P., Kumar, A., Faloutsos, C., Papalexakis, E.E., Xing, E.P.: Flexifact: Scalable flexible factorization of coupled tensors on hadoop. In: Proceedings of the 2014 SIAM International Conference on Data Mining, pp. 109–117. SIAM (2014)
12. Bishop, C.M.: Bayesian pca. In: Advances in neural information processing systems, pp. 382–388 (1999)
13. Brunet, J.P., Tamayo, P., Golub, T.R., Mesirov, J.P.: Metagenes and molecular pattern discovery using matrix factorization. Proceedings of the national academy of sciences **101**(12), 4164–4169 (2004)
14. Chan, E., Heimlich, M., Purkayastha, A., Van De Geijn, R.: Collective communication: theory, practice, and experience. Concurrency and Computation: Practice and Experience **19**(13), 1749–1783 (2007)
15. Cichocki, A., Phan, A.H., Zhao, Q., Lee, N., Oseledets, I., Sugiyama, M., Mandic, D.P., et al.: Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. Foundations and Trends® in Machine Learning **9**(6), 431–673 (2017)
16. Cichocki, A., Zdunek, R., Amari, S.i.: Hierarchical als algorithms for nonnegative matrix and 3d tensor factorization. In: International Conference on Independent Component Analysis and Signal Separation, pp. 169–176. Springer (2007)
17. Cichocki, A., Zdunek, R., Phan, A.H., Amari, S.i.: Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-way Data Analysis and Blind Source Separation. Wiley Publishing (2009)
18. Deerwester, S.C., Dumais, S.T., Furnas, G.W., Harshman, R.A., Landauer, T.K., Lochbaum, K.E., Streeter, L.A.: Computer information retrieval using latent semantic structure (1989). US Patent 4,839,853
19. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society: Series B (Methodological) **39**(1), 1–22 (1977)
20. Dong, C., Zhao, H., Wang, W.: Parallel nonnegative matrix factorization algorithm on the distributed memory platform. International journal of parallel programming **38**(2), 117–137 (2010)
21. Donoho, D., Stodden, V.: When does non-negative matrix factorization give a correct decomposition into parts? In: Advances in neural information processing systems, pp. 1141–1148 (2004)
22. Fairbanks, J.P., Kannan, R., Park, H., Bader, D.A.: Behavioral clusters in dynamic graphs. Parallel Computing **47**, 38–50 (2015)

23. Févotte, C., Cemgil, A.T.: Nonnegative matrix factorizations as probabilistic inference in composite models. In: 2009 17th European Signal Processing Conference, pp. 1913–1917. IEEE (2009)
24. Franke, B., Plante, J.F., Roscher, R., Lee, E.s.A., Smyth, C., Hatefi, A., Chen, F., Gil, E., Schwing, A., Selvitella, A., et al.: Statistical inference, learning and models in big data. International Statistical Review **84**(3), 371–389 (2016)
25. Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 69–77. ACM (2011)
26. Golub, G.H., Reinsch, C.: Singular value decomposition and least squares solutions. In: Linear Algebra, pp. 134–151. Springer (1971)
27. Hartigan, J.A., Wong, M.A.: Algorithm as 136: A k-means clustering algorithm. Journal of the Royal Statistical Society. Series C (Applied Statistics) **28**(1), 100–108 (1979)
28. Huang, K., Sidiropoulos, N.D., Liavas, A.P.: A flexible and efficient algorithmic framework for constrained matrix and tensor factorization. IEEE Transactions on Signal Processing **64**(19), 5052–5065 (2016)
29. Iliev, F.L., Stanev, V.G., Vesselinov, V.V., Alexandrov, B.S.: Nonnegative matrix factorization for identification of unknown number of sources emitting delayed signals. PloS one **13**(3), e0193974 (2018)
30. Jolliffe, I.: Principal component analysis. Springer (2011)
31. Kanna, R.: Parallel low-rank approximations with non-negativity constraints (PLANC). `https://github.com/ramkikannan/planc`. Accessed: 2019-09-03
32. Kannan, R., Ballard, G., Park, H.: A high-performance parallel algorithm for nonnegative matrix factorization. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, pp. 9:1–9:11. ACM (2016)
33. Kim, J., Park, H.: Toward faster nonnegative matrix factorization: A new algorithm and comparisons. In: 2008 Eighth IEEE International Conference on Data Mining, pp. 353–362. IEEE (2008)
34. Kim, J., Park, H.: Fast nonnegative matrix factorization: An active-set-like method and comparisons. SIAM Journal on Scientific Computing **33**(6), 3261–3281 (2011)
35. Koitka, S., Friedrich, C.M.: nmfgpu4r: Gpu-accelerated computation of the non-negative matrix factorization (nmf) using cuda capable hardware. The R Journal **8**(2), 382–392 (2016)
36. Korenius, T., Laurikkala, J., Juhola, M.: On principal component analysis, cosine and euclidean measures in information retrieval. Information Sciences **177**(22), 4893–4905 (2007)
37. Kubjas, K., Robeva, E., Sturmfels, B.: Fixed points EM algorithm and nonnegative rank boundaries. The Annals of Statistics pp. 422–461 (2015)
38. Kysenko, V., Rupp, K., Marchenko, O., Selberherr, S., Anisimov, A.: Gpu-accelerated non-negative matrix factorization for text mining. In: International Conference on Application of Natural Language to Information Systems, pp. 158–163. Springer (2012)
39. Laurberg, H., Christensen, M.G., Plumbley, M.D., Hansen, L.K., Jensen, S.H.: Theorems on positive data: On the uniqueness of nmf. Computational intelligence and neuroscience **2008** (2008)
40. Lee, D.D., Seung, H.S.: Learning the parts of objects by non-negative matrix factorization. Nature **401**(6755), 788–791 (1999)
41. Liao, R., Zhang, Y., Guan, J., Zhou, S.: Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets. Genomics, proteomics & bioinformatics **12**(1), 48–51 (2014)
42. Liu, C., Yang, H.c., Fan, J., He, L.W., Wang, Y.M.: Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In: Proceedings of the 19th international conference on World wide web, pp. 681–690. ACM (2010)
43. Lopes, N., Ribeiro, B.: Non-negative matrix factorization implementation using graphic processing units. In: International Conference on Intelligent Data Engineering and Automated Learning, pp. 275–283. Springer (2010)
44. MacKay, D.J., et al.: Bayesian nonlinear modeling for the prediction competition. ASHRAE transactions **100**(2), 1053–1062 (1994)

45. Mejía-Roa, E., García, C., Gómez, J.I., Prieto, M., Tirado, F., Nogales, R., Pascual-Montano, A.: Biclustering and classification analysis in gene expression using nonnegative matrix factorization on multi-gpu systems. In: 2011 11th International Conference on Intelligent Systems Design and Applications, pp. 882–887. IEEE (2011)
46. Mejía-Roa, E., Tabas-Madrid, D., Setoain, J., García, C., Tirado, F., Pascual-Montano, A.: NMF-mGPU: non-negative matrix factorization on multi-gpu systems. BMC bioinformatics **16**(1), 43 (2015)
47. Moon, G.E., Sukumaran-Rajam, A., Parthasarathy, S., Sadayappan, P.: PL-NMF: parallel locality-optimized non-negative matrix factorization. CoRR **abs/1904.07935** (2019)
48. Paatero, P., Tapper, U.: Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. Environmetrics **5**(2), 111–126 (1994)
49. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics **20**, 53–65 (1987)
50. Sanderson, C., Curtin, R.: Armadillo: a template-based c++ library for linear algebra. Journal of Open Source Software **1**(2), 26 (2016)
51. Spearman, C.: "General Intelligence" objectively determined and measured. The American Journal of Psychology **15**(2), 201–292 (1904)
52. Stanev, V., Vesselinov, V.V., Kusne, A.G., Antoszewski, G., Takeuchi, I., Alexandrov, B.S.: Unsupervised phase mapping of x-ray diffraction data by nonnegative matrix factorization integrated with custom clustering. npj Computational Materials **4**(1), 43 (2018)
53. Stanev, V.G., Iliev, F.L., Hansen, S., Vesselinov, V.V., Alexandrov, B.S.: Identification of release sources in advection–diffusion system by machine learning combined with green's function inverse method. Applied Mathematical Modelling **60**, 64–76 (2018)
54. Sun, D.L., Fevotte, C.: Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence. In: 2014 IEEE international conference on acoustics, speech and signal processing (ICASSP), pp. 6201–6205. IEEE (2014)
55. Sun, M., Zhang, X., et al.: A stable approach for model order selection in nonnegative matrix factorization. Pattern Recognition Letters **54**, 97–102 (2015)
56. Syed, A.M., Qazi, S., Gillis, N.: Improved svd-based initialization for nonnegative matrix factorization using low-rank correction. arXiv preprint arXiv:1807.04020 (2018)
57. Tan, V.Y.F., Févotte, C.: Automatic relevance determination in nonnegative matrix factorization. In: SPARS'09 - Signal Processing with Adaptive Sparse Structured Representations, Inria Rennes - Bretagne Atlantique (2009)
58. Vesselinov, V.V., Alexandrov, B.S., O'Malley, D.: Contaminant source identification using semi-supervised machine learning. Journal of contaminant hydrology **212**, 134–142 (2018)
59. Wold, S., Sjöström, M., Eriksson, L.: Pls-regression: a basic tool of chemometrics. Chemometrics and intelligent laboratory systems **58**(2), 109–130 (2001)
60. Wu, S., Joseph, A., Hammonds, A.S., Celniker, S.E., Yu, B., Frise, E.: Stability-driven nonnegative matrix factorization to interpret spatial gene expression and build local gene networks. Proceedings of the National Academy of Sciences **113**(16), 4290–4295 (2016)
61. Yin, J., Gao, L., Zhang, Z.M.: Scalable nonnegative matrix factorization with block-wise updates. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 337–352. Springer (2014)

Appendix

Table 3: Execution times for distributed clustering and Silhouette calculation using *DnMFk* on two data matrices: *Data 3* (57600×38400) and *Data 4* (129600×51840). These two matrices does not fit in the memory of a single node therefore the execution times for 36 cores is ignored.

Data	Processors			
	144	324	576	900
Clustering Time (s)				
Data 3	1.941	0.896	0.324	0.251
Data 4	3.674	1.190	0.591	0.371
Silhouette Time (s)				
Data 3	1.051	0.650	0.341	0.251
Data 4	1.980	1.030	0.631	0.481

Table 4: Execution times for distributed clustering and Silhouette calculation using *DnMFk* on two data matrices: *Data 1* ($2^{21} \times 10 \times 10$) and *Data 2* ($2^{20} \times 50 \times 10$). The reported runtimes are for a different number of processors with 10 perturbations and 100 NMF iterations in each perturbation. The runtimes are clearly decreasing as the number of processors increase.

Input	Processors						
	1	18	32	144	324	576	900
Clustering Times (s)							
Data $2^{21} \times 10 \times 10$	74.351	5.325	2.934	0.851	0.502	0.362	0.112
Data $2^{20} \times 50 \times 10$	190.046	13.440	7.349	1.985	1.060	0.710	0.667
Silhouette Times (s)							
Data $2^{21} \times 10 \times 10$	117.610	6.488	5.396	1.609	0.988	0.768	0.099
Data $2^{20} \times 50 \times 10$	1555.399	87.653	56.140	19.697	12.390	2.780	2.617

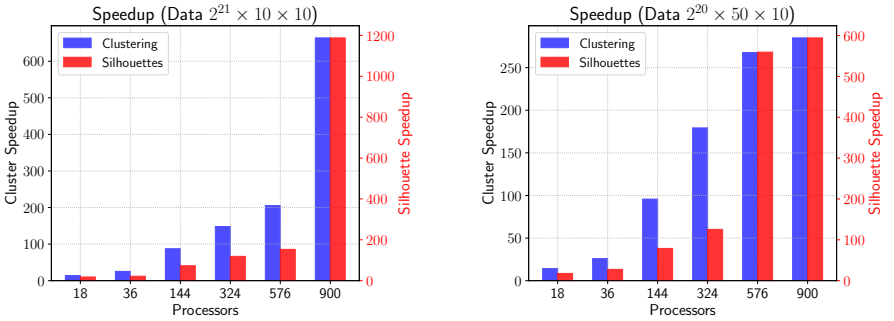

Fig. 14: Speedups for clustering and silhouette – The results are across 10 perturbations of *DnMFk* for *Data* ($2^{21} \times 10 \times 10$, left) and *Data* ($2^{20} \times 50 \times 10$, right). The speedups are with respect to the runtimes of *DnMFk* execution on 1 processor. We can clearly witness the performance scales linearly as the number of processors increase. Especially, *Data* $2^{21} \times 10 \times 10$, shows a speed of 663.84x for clustering while that of silhouettes is 1187.98x. For *Data* $2^{20} \times 50 \times 10$, the speedups are 594.34x and 284.93x for clustering and silhouettes.

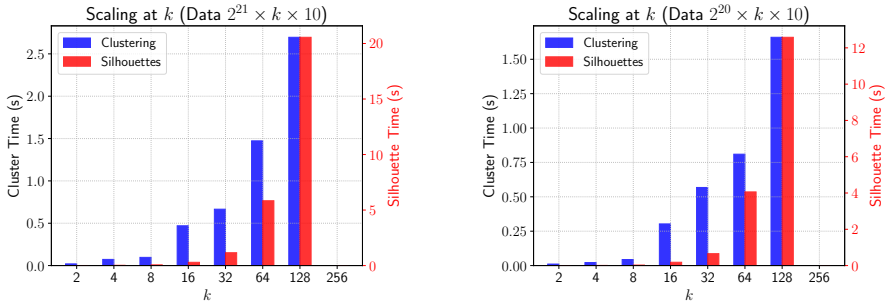

Fig. 15: k scaling for clustering and silhouette – The runtimes of *clustering* and *silhouette* across 10 perturbations of *DnMFk*. The matrices are $2^{21} \times k \times 10$ and $2^{20} \times k \times 10$, where the k vary as $\{2, 4, 8, 16, 32, 64, 128, 256\}$. The execution time increases linearly with k at a fixed processor count.

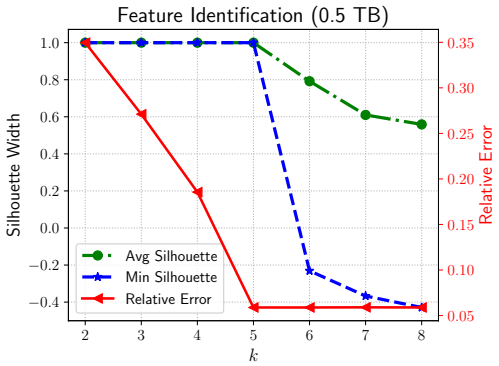

Fig. 16: Find the number of hidden features in 0.5 TB matrix of 8388608×8192. *DnMFk* find k as 5, which agrees with the ground truth. *DnMFk* takes 13.6 hours on 2048 processors with an average reconstruction error of 5.8%.