informatics *mathematics*

Ínría

Partitioning and Communication Strategies for Sparse Non-negative Matrix Factorization

Oguz Kaya Ramakrishnan Kannan Grey Ballard

Partitioning and Communication Strategies
for Sparse Non-negative Matrix Factorization [*]

Oguz Kaya [†] Ramakrishnan Kannan [‡] Grey Ballard [§]

Project-Team INRIA Bordeaux HiePACS

Abstract: Non-negative matrix factorization (NMF), the problem of finding two non-negative low-rank factors whose product approximates an input matrix, is a useful tool for many data mining and scientific applications such as topic modeling in text mining and blind source separation in microscopy. In this paper, we focus on scaling algorithms for NMF to very large sparse datasets and massively parallel machines by employing effective algorithms, communication patterns, and partitioning schemes that leverage the sparsity of the input matrix. In the case of machine learning workflow, the computations after SpMM must deal with dense matrices, as Sparse-Dense matrix multiplication will result in a dense matrix. Hence, the partitioning strategy considering only SpMM will result in a huge imbalance in the overall workflow especially on computations after SpMM and in this specific case of NMF on non-negative least squares computations. Towards this, we consider two previous works developed for related problems, one that uses a fine-grained partitioning strategy using a point-to-point communication pattern and on that uses a checkerboard partitioning strategy using a collective-based communication pattern. We show that a combination of the previous approaches balances the demands of the various computations within NMF algorithms and achieves high efficiency and scalability. From the experiments, we could see that our proposed algorithm communicates atleast 4x less than the collective and achieves upto 100x speed up over the baseline FAUN on real world datasets. Our algorithm was experimented in two different super computing platforms and we could scale up to 32000 processors on Bluegene/Q.

Key-words: sparse non-negative matrix factorization, hypergraph partitioning, parallel algorithms

---

Les Stratégies de Partitionnement et de Communication pour Factorisation des Matrices Non-négatives Creuses

Résumé :   La factorisation de matrice non-négative (NMF), le problème de trouver deux facteurs de rang faible non négatifs dont le produit se rapproche d'une matrice d'entrée, est un outil utile pour de nombreuses applications scientifiques et d'exploration de données telles que la modélisation de textes et la séparation de signaux en microscopie. Dans cet article, nous etudions les algorithmes passant à l'échelle pour NMF à de très grands ensembles de données creuses et des machines massivement parallèles en utilisant des algorithmes efficaces, des modèles de communication et des schémas de partitionnement qui exploitent la structure creuse de la matrice. Dans le cadre de cet algorithme, les calculs après SpMM doivent traiter des matrices denses, car la multiplication SpMM produira une matrice dense. Par conséquent, la stratégie de partitionnement ne prenant en compte que SpMM entraînera un déséquilibre énorme dans l'algorithme global, en particulier sur les calculs après SpMM et dans ce cas spécifique de NMF sur les calculs de moindres carrés non négatifs. À cet égard, nous considérons deux travaux antérieurs développés pour des problèmes connexes, l'un utilisant une stratégie de partitionnement de granularité ffine utilisant un modèle de communication "point-to-point" et utilisant une stratégie de partitionnement en damier utilisant un modèle de communication collectif. Nous montrons qu'une combinaison des approches précédentes permet d'équilibrer les exigences des divers calculs au sein des algorithmes NMF et permet d'obtenir une efficacité et une évolutivité élevées. À partir des expériences, nous avons constaté que notre algorithme proposé communique au moins 4x moins que le collectif et atteint jusqu'à 100 fois la vitesse de base sur les jeux de données réels. Notre algorithme a été expérimenté sur deux plates-formes superinformatiques différentes et nous avons pu passer à 32 000 processeurs sur Bluegene / Q.

Mots-clés : factorisation de matrice non-negative creuse, partitionnement d'hypergraphes, algorithmique parallèle

1   Introduction

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

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

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

NMF is widely used in data mining and machine learning as a dimension reduction and factor analysis method. It is a natural fit for many real world problems as the non-negativity is inherent in many representations of real-world data and the resulting low rank factors are expected to have a natural interpretation. The applications of NMF range from text mining [24], computer vision [10], and bioinformatics [15] to blind source separation [5], unsupervised clustering [17, 18] and many other areas. In most real-world applications m and n are on the order of millions or more while k is much smaller, on the order of tens to thousands. Furthermore, data sets from these applications are often quite sparse and have highly irregular nonzero patterns. We would like to highlight that "Non-negative" Matrix Factorization is NOT matrix factorization in collaborative filtering for recommender systems for which many implementations exist. The collaborative filtering problem is different than NMF – the focus of our paper – because it interprets the "*zero*" entries of the input matrix as missing data, while the NMF problem is defined for a completely known input matrix and does not handle missing values. This leads to different optimization problems, algorithms, and computational steps.

The most common method for solving Eq. (1) is to use an alternating optimization approach, iteratively updating $\mathbf{W}$ with $\mathbf{H}$ fixed and then updating $\mathbf{H}$ with $\mathbf{W}$ fixed. Specifically, updating a factor matrix involves three main operations; computing of $\mathbf{W}^T\mathbf{A}$ or $\mathbf{A}\mathbf{H}^T$, computing the Gram matrix $\mathbf{W}^T\mathbf{W}$ or $\mathbf{H}\mathbf{H}^T$, and solving a non-negative linear least squares (NLS) problem using these two resulting matrices to update the factor matrix. When the rank k is small, the typical bottleneck is the computation that involves the input matrix, $\mathbf{W}^T\mathbf{A}$ and $\mathbf{A}\mathbf{H}^T$, which are sparse-dense matrix multiplications (SpMMs) when the input data is sparse. However, for large k, the Gram and NLS computations can become the bottleneck, as they grow more quickly with k than the SpMMs. Efficient parallel algorithms for NMF must load balance the computation and avoid communication overheads. The distribution of the input $\mathbf{A}$ across processors affects the load balance and communication of the SpMMs, and the distribution of the factor matrices $\mathbf{W}$ and $\mathbf{H}$ affects the load balance of the Gram and NLS computations.

There is a lack of highly scalable parallel algorithms and software for computing sparse NMF. MPI-FAUN [13] software framework enables computing the NMF for dense and sparse data, yet the algorithm is optimized for dense input matrices and employs the same communication scheme to carry out sparse NMF computations. It uses a regular, Cartesian partitioning of the input matrix $\mathbf{A}$ across processors that is oblivious to the nonzero pattern. The advantages of this approach are that the resulting factor matrix communication (to compute the SpMMs) is also regular and cast as low-latency MPI collective operations and that the rest of the computation (including NLS updates) is perfectly load-balanced. The disadvantage is that because the partition ignores the sparsity of $\mathbf{A}$, the approach will often communicate more data than necessary, as some processors will receive data that they do not need for local computation. For sparse tensor factorization, there are scalable algorithms and software resembling NMF kernels with sparse irregular computational patterns [14], yet they require an expensive partitioning step and do not necessarily yield the optimal performance for NMF. The main advantage of this approach is that it minimizes the communication cost of the SpMM step through hypergraph partitioning, and that it employs point-to-point communication scheme which communicates elements of $\mathbf{W}$ and $\mathbf{H}$ only to processors in need. The disadvantages of this approach are an upfront hypergraph partitioning cost, possible load imbalance (particularly in the NLS computations) and communication

$\mathbf{A}$	Input matrix
$\mathbf{W}$	Left low rank factor
$\mathbf{H}$	Right low rank factor
m	Number of rows of input matrix
n	Number of columns of input matrix
k	Low rank
P	Number of parallel processes
P_r	Number of rows in processor grid
P_c	Number of columns in processor grid
$\mathcal{I}_p, \mathcal{J}_p$	Set of rows/columns of of $\mathbf{W}/\mathbf{H}$ owned by process p
$\mathcal{F}_p, \mathcal{G}_p$	Set of unique row and column indices of $\mathbf{A}_p$
$\mathbf{A}_p$	Submatrix of $\mathbf{A}$ owned by process p
$\mathbf{W}(\mathcal{I}_p,:)$	Owned rows of initial $\mathbf{W}$ by process p
$\mathbf{H}(:,\mathcal{J}_p)$	Owned columns of initial $\mathbf{H}$ by process p

Table 1: Notation

imbalance. We would like to iterate that in the case of machine learning workflow, the computations after SpMM must deal with dense matrices, as Sparse-Dense matrix multiplication will result in a dense matrix. Hence, the partitioning strategy considering only SpMM will result in a huge imbalance in the overall workflow especially on computations after SpMM and in this specific case of NMF on NLS computations. Our goal is in this paper is to fill in this gap between two approaches by comparing and evaluating them in the context of NMF, then proposing a synthesis involving parallel algorithms as well as communication and partitioning schemes enabling scalability to thousands of processes.

2 Sparse Non-negative Matrix Factorization

2.1 Notation

Table 1 summarizes the notation we use throughout this paper. We use bold uppercase letters for matrices and lowercase letters for vectors. For matrix rows and columns, we employ MATLAB notation, i.e., $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ refer to the ith row and the jth column of $\mathbf{A}$. We use subscripts to refer to sub-blocks of matrices. For example, $\mathbf{A}_i j$ refers to the sub-block (i,j) of $\mathbf{A}$ in a 2D partition. We use m and n to denote the numbers of rows and columns of $\mathbf{A}$, respectively, and assume without loss of generality $m \geqslant n$ throughout.

2.2 Alternating-Updating NMF

The Alternating-Updating NMF algorithms are those that alternate between updating one of $\mathbf{W}$ and $\mathbf{H}$ using the given input matrix $\mathbf{A}$ and other 'fixed' factor - $\mathbf{H}$ for updating $\mathbf{W}$ or $\mathbf{W}$ for updating $\mathbf{H}$. This update is performed using the Gram matrix associated with the fixed factor matrix, and the product of the input matrix $\mathbf{A}$ with the fixed factor matrix. We show the structure of the framework in Algorithm 1.

After computing the Gram matrix and the multiplication of $\mathbf{A}$ with the fixed factor matrix, the specifics of the update at Lines 3 and 4 depend on the NMF algorithm, and we refer to the computation associated with these lines as the Local Update Computations (**LUC**).

We note that AU-NMF is very similar to a two-block, block coordinate descent (BCD) framework as explained by Bertsekas [2]. The BCD framework expresses solving optimization variables in complex

---

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

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

---

non-linear optimization problem as one block at a time, while keeping the others fixed. In NMF, the two blocks are the unknown factors $\mathbf{W}$ and $\mathbf{H}$, and we *solve* the following subproblems, which have a unique solution for a full rank $\mathbf{H}$ and $\mathbf{W}$:

$$\mathbf{W} \leftarrow \underset{\tilde{\mathbf{W}} \geqslant 0}{\arg\min} \left\| \mathbf{A} - \tilde{\mathbf{W}}\mathbf{H} \right\|_F + \phi(\tilde{\mathbf{W}}) + \psi(\mathbf{H}),$$

$$\mathbf{H} \leftarrow \underset{\tilde{\mathbf{H}} \geqslant 0}{\arg\min} \left\| \mathbf{A} - \mathbf{W}\tilde{\mathbf{H}} \right\|_F + \phi(\mathbf{W}) + \psi(\tilde{\mathbf{H}}). \tag{2}$$

Since each subproblem involves non-negative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [16]. Block Principal Pivoting (**ABPP**) is one algorithm that solves these NLS subproblems. In the context of the AU-NMF algorithm, an ANLS method *maximally* reduces the overall NMF objective function value by finding the optimal solution for given $\mathbf{H}$ and $\mathbf{W}$ in Lines 3 and 4, respectively.

From time to time, these updates do not necessarily solve each of the subproblems (2) to optimality but simply improve the overall objective function (3). such as Multiplicative Update (**MU**) [27] and Hierarchical Alternating Least Squares (**HALS**) [5].

The convergence properties of these different NMF algorithms are discussed in detail by Kim, He and Park [16]. While we focus only on the **MU** algorithms in this paper, we highlight that our algorithm is not restricted to this, and is seamlessly extensible to other NMF algorithms as well, including **HALS**, **ABPP**, Alternating Direction Method of Multipliers (**ADMM**) [30], and Nesterov-based methods [9].

2.3 Multiplicative Update (MU)

In this paper, we are considering ℓ_2 regularization on the $\mathbf{W}$ matrix and ℓ_1 regularization on the $\mathbf{H}$ matrix to address the sparsity of the input matrix. Hence, the NMF problem becomes

$$\min_{\mathbf{W} \geqslant 0, \mathbf{H} \geqslant 0} \|\mathbf{A} - \mathbf{WH}\|_F + \alpha\|\mathbf{W}\|_F^2 + \beta\sum_{i=1}^{n}\|\mathbf{h}_i\|_1^2. \tag{3}$$

The values α and β were fixed for the experiments. In the case of **MU** [27], individual entries of $\mathbf{W}$ and $\mathbf{H}$ are updated with all other entries fixed. In this case, the update rules are

$$w_{ij} \leftarrow w_{ij}\frac{(\mathbf{AH}^T)_{ij}}{(\mathbf{W}(\mathbf{HH}^T + 2\beta\mathbf{1}_k))_{ij}}, \text{ and}$$

$$h_{ij} \leftarrow h_{ij}\frac{(\mathbf{W}^T\mathbf{A})_{ij}}{((\mathbf{W}^T\mathbf{W} + 2\alpha\mathbf{I}_k)\mathbf{H})_{ij}}. \tag{4}$$

where $\mathbf{1}_k$ is a matrix of $k \times k$ with all one's and $\mathbf{I}_k$ is an identity matrix of size $k \times k$.

After computing the Gram matrices $\mathbf{HH}^T$ and $\mathbf{W}^T\mathbf{W}$, adding the appropriate regularizers and the products $\mathbf{AH}^T$ and $\mathbf{W}^T\mathbf{A}$, the extra cost of computing $\mathbf{W}(\mathbf{HH}^T + 2\beta\mathbf{1}_k)$ and $(\mathbf{W}^T\mathbf{W} + 2\alpha\mathbf{I}_k)$ is

$F(m,n,k)=2(m+n)k^2$ flops to perform updates for all entries of $\mathbf{W}$ and $\mathbf{H}$, as the other elementwise operations affect only lower-order terms. The details about using AU-NMF in Algorithm 1 for other algorithms **HALS** and **ABPP** are explained in [11, 12].

The above update equation Eq. (5) can be easily parallelized using the Algorithm 2.

It is important to observe that the update function of $\mathbf{W}$ is element-wise normalization of the sparse matrix-dense matrix multiplication $\mathbf{AH}^T$ with the denominator. Given that if all the processes owns the $k \times k$ gram of the factor matrix $\mathbf{H}$, computing the entire denominator $(\mathbf{W}(\mathbf{HH}^T+2\beta\mathbf{1}_k))$ is does not require any communication, hence can be done locally. One can argue that the same holds for updating $\mathbf{H}$ as well. Therefore, for row and column index sets $\mathcal{I}_p$ and $\mathcal{J}_p$, one can update these rows of the factor matrices as follows:

$$\mathbf{W}(\mathcal{I}_p,:) \leftarrow \mathbf{W}(\mathcal{I}_p,:) \circledast (\tilde{\mathbf{W}}(\mathcal{I}_p,:)) \oslash (\mathbf{W}(\mathcal{I}_p,:)(\mathbf{G}_H+2\beta\mathbf{1}_k)), \text{ and}$$
$$\mathbf{H}(\mathcal{J}_p,:) \leftarrow \mathbf{H}(\mathcal{J}_p,:) \circledast (\tilde{\mathbf{H}}(\mathcal{J}_p,:) \oslash (\mathbf{G}_W+2\alpha\mathbf{I}_k)\mathbf{H}(\mathcal{J}_p,:)).$$

(5)

where $\circledast$ and $\oslash$ correspond to element-wise multiplication and division of matrices or vectors. This scheme provides row-wise parallelism in NNLS computation. **HALS** and **ABPP** can similarly be expressed in this row-parallel form.

3   survey

In the data mining and machine learning literature, there is an overlap between low rank approximations and matrix factorizations due to the nature of applications. Despite its name, non-negative matrix "factorization" is in fact a low rank approximation. Recently, there has been a growing interest in collaborative filtering based recommender systems. One of the popular techniques for collaborative filtering is matrix factorization, often with nonnegativity constraints, and its implementation is widely available in many off-the-shelf distributed machine learning libraries such as GraphLab [21], MLLib [23], and many others [26, 33]. However, we would like to emphasize that collaborative filtering using matrix factorization is a different problem than NMF: In the case of collaborative filtering, nonzeros in the matrix are considered to be observed ratings and zeros to be missing entries, while in the case of NMF, there is no missing entries and even zero is an observed entry.

There are several recent distributed NMF algorithms in the literature [19, 6, 32, 20]. Liu et al. propose running Multiplicative Update (MU) for KL divergence, squared loss, and "exponential" loss functions [20]. Matrix multiplication, element-wise multiplication, and element-wise division are the building blocks of the MU algorithm. The authors discuss performing these matrix operations effectively in Hadoop for sparse matrices. Using similar approaches, Liao et al. implement an open source Hadoop-based MU algorithm and study its scalability on large-scale biological data sets [19]. Also, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [32]. Similarly Faloutsos et al. propose a distributed, scalable method for decomposing matrices, tensors, and coupled data sets through stochastic gradient descent on a variety of objective functions [6]. The authors also provide an implementation that can enforce non-negative constraints on the factor matrices. All of these works use Hadoop framework to implement their algorithms, hence are not very efficient.

Spark [34] is a popular big-data processing infrastructure that is generally more efficient for iterative algorithms such as NMF than Hadoop, as it maintains data in memory and avoids file system I/O. Even with a Spark implementation of previously proposed Hadoop-based NMF algorithms, the performance still suffers from expensive communication of input matrix entries, and Spark does not have innate mechanisms to overcome this shortcoming. Spark has collaborative filtering libraries such as MLlib [23] which use matrix factorization and can impose non-negativity constraints. Nevertheless, as mentioned,

the problem of collaborative filtering is different from NMF, hence we do not compare our approach against these.

In parallel with the Hadoop and Spark implementations, there have been growing interest in the HPC community towards efficiently computing these algorithms with tuned high performance implementations. Kannan, Ballard and Park [11, 12], have proposed MPI-FAUN framework to implement various NMF algorithms such as multiplicative update (MU), Hierarchical Alternating Least Squares (HALS) and Alternating Non-negative Least Squares using Block Principal Pivoting (ANLS-BPP). We choose this work as a baseline, as it is the only available high performance implementation of NMF, and it performs significantly faster than Hadoop and Spark-based approaches. To elaborate this, Gittens et.al., [7] recently benchmarked the implementations of different matrix factorization algorithms, such as NMF and Principal Component Analysis (PCA), in Spark and in C and MPI, which is tuned for HPC platforms. They claim native MPI implementations on HPC platforms out perform Spark implementation by a speedup factor of 44x. Similar observations have been made by Sukumar, Kannan, Matheson and Lim [28, 29] on super computers at Oak Ridge Leadership Computing Facility. Finally, there are implementations of the MU algorithm in a distributed memory setting using X10 [8] and on a GPU [22].

4 Distributed Sparse NMF

Here, we first introduce our parallel NMF algorithm that operates on a partition of the matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. For a given partition, we describe how parallel computations and communications take place within the algorithm, and illustrate computational and communication costs associated with a partition. We then discuss efficient partitioning strategies to better establish computational load balance and reduce communication in NMF. In doing so, we also explain how existing methods compare to this scheme with their advantages and disadvantages in terms of partitioning.

4.1 Distributed Sparse NMF Algorithm

Algorithm 2 DIST-SPNMF: Distributed sparse NMF algorithm

Input: $\mathbf{A}_p$: An $m \times n$ sparse matrix
$\quad \mathcal{I}_p, \mathcal{J}_p$: Set of rows/columns of $\mathbf{W}$ and $\mathbf{H}$ owned by process p
$\quad \mathcal{F}_p, \mathcal{G}_p$: Footprints of process p on $\mathbf{W}$ and $\mathbf{H}$
$\quad \mathbf{W}(\mathcal{I}_p,:), \mathbf{H}(:,\mathcal{J}_p)$: Owned rows/columns of $\mathbf{W}$ and $\mathbf{H}$
$\quad k$: The NMF rank
Output: Process p gets final values of $\mathbf{W}(\mathcal{I}_p,:)$ and $\mathbf{H}(:,\mathcal{J}_p)$
1: **repeat**
2: $\quad$ COMM-EXPAND($\mathbf{H}(:,\mathcal{G}_p)$)
3: $\quad \tilde{\mathbf{W}}(\mathcal{F}_p,:) \leftarrow \mathbf{A}_p \mathbf{H}(:,\mathcal{G}_p)$
4: $\quad$ COMM-FOLD($\tilde{\mathbf{W}}(\mathcal{F}_p,:)$)
5: $\quad \mathbf{G}_H \leftarrow$ ALL-REDUCE($\mathbf{H}(:,\mathcal{J}_p)\mathbf{H}(:,\mathcal{J}_p)^T$)
6: $\quad \mathbf{W}(\mathcal{I}_p,:) \leftarrow$ NNLS($\mathbf{G}_H, \tilde{\mathbf{W}}(\mathcal{I}_p,:)$)
7: $\quad$ COMM-EXPAND($\mathbf{W}(\mathcal{F}_p,:)$)
8: $\quad \tilde{\mathbf{H}}(:,\mathcal{G}_p) \leftarrow \mathbf{W}(\mathcal{F}_p,:)^T \mathbf{A}_p$
9: $\quad$ COMM-FOLD($\tilde{\mathbf{H}}(:,\mathcal{G}_p)$)
10: $\quad \mathbf{G}_W \leftarrow$ ALL-REDUCE($\mathbf{W}(\mathcal{I}_p,:)^T \mathbf{W}(\mathcal{I}_p,:)$)
11: $\quad \mathbf{H}(\mathcal{J}_p,:) \leftarrow$ NNLS($\mathbf{G}_W, \tilde{\mathbf{H}}(\mathcal{J}_p,:)$)
12: **until** convergence or maximum number of iterations

Parallelizing sparse NMF involves partition the sparse matrix $\mathbf{A}$ as well as the factor matrices $\mathbf{W}$ and $\mathbf{H}$, where the former partitioning distributes the computational load of sparse matrix-dense matrix multiplications $\mathbf{AH}$ and $\mathbf{W}^T\mathbf{A}$, whereas the latter divides the workload of NNLS computations to processes. We provide the execution of our parallel algorithm for computing a rank-k NMF of a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ in Algorithm 2, which is executed by each process p for $1 \le p \le P$. The algorithm starts with an arbitrary partition of the input matrix and the factor matrices; process p owns the submatrices $\mathbf{W}(\mathcal{I}_p,:)$ and $\mathbf{H}(:,\mathcal{J}_p)$ as well as the nonzero elements of the sparse matrix $\mathbf{A}_p$ where $\mathbf{A} = \bigcup_{i=1}^{P} \mathbf{A}_i$, i.e., $\mathbf{A}_1,...,\mathbf{A}_P$ partitions the nonzeros of $\mathbf{A}$. The sets $\mathcal{F}_p$ and $\mathcal{G}_p$ denote the "footprints" of the process p on the rows and the columns of matrices $\mathbf{W}$ and $\mathbf{H}$, respectively; hence, these rows need to be stored by this process. Specifically, we have $i \in \mathcal{F}_p$ or $j \in \mathcal{G}_p$ if only if $i \in \mathcal{I}_p$ or $j \in \mathcal{J}_p$ (row/column is owned), or there is a nonzero element $a_{i,j} \in \mathbf{A}_p$ (row/column is used in local computations). At each iteration, the process p is responsible for gathering the new value of submatrices $\mathbf{W}(\mathcal{I}_p,:)$ and $\mathbf{H}(:,\mathcal{J}_p)$, and sending them to processes in need.

In an iteration of Algorithm 2, each process p possesses three types of computational tasks as well as associated pre- and post-communication steps. The first task involves performing sparse matrix-dense matrix multiplications $\mathbf{A}_p\mathbf{H}(:,\mathcal{G}_p)$ and $\mathbf{W}(\mathcal{F}_p,:)^T\mathbf{A}_p$, whose results are stored in distributed matrices $\tilde{\mathbf{W}}$ and $\tilde{\mathbf{H}}$, which follow the same row-/column-wise data distribution as $\mathbf{W}$ and $\mathbf{H}$. Note that carrying out these multiplications must be preceded by a communication step where each process p gets the submatrices $\mathbf{H}(:,\mathcal{G}_p \backslash \mathcal{J}_p)$ and $\mathbf{W}(\mathcal{F}_p \backslash \mathcal{I}_p,:)$ that are accessed by entries of $\mathbf{A}_p$, and this step is performed at Lines 2 and 7. These multiplications performed by each process p generate partial results for the set $\mathcal{F}_p$ and $\mathcal{G}_p$ of rows of $\tilde{\mathbf{W}}$ and columns of $\tilde{\mathbf{H}}$, respectively, which is highlighted at Lines 3 and 8. Indeed, partial results for the submatrices $\tilde{\mathbf{W}}(\mathcal{F}_p \backslash \mathcal{I}_p,:)$ and $\tilde{\mathbf{H}}(:,\mathcal{G}_p \backslash \mathcal{J}_p)$ correspond to rows and columns owned by other processes; hence, they need to be communicated. The results for $\tilde{\mathbf{W}}(\mathcal{I}_p,:)$ and $\tilde{\mathbf{H}}(:,\mathcal{J}_p)$, however, should be kept locally, and all partial results for these matrix rows and columns generated by other processes should similarly be received and accumulated in order to obtain the final value for these owned portions. The second task is to compute the Gram matrices $\mathbf{G}_H = \mathbf{HH}^T$ and $\mathbf{G}_W = \mathbf{W}^T\mathbf{W}$ of size $k \times k$, and making these matrices available to all processes, which is performed at Lines 5 and 10. This is done in a row-parallel dense matrix multiplication step, in which the process p computes $\mathbf{H}(:,\mathcal{J}_p)\mathbf{H}^T(:,\mathcal{J}_p)$ and $\mathbf{W}^T(\mathcal{I}_p,:)\mathbf{W}(\mathcal{I}_p,:)$, followed by an ALL-REDUCE communication of these partial multiplications. The third task pertains to updating the factor matrices $\mathbf{W}$ and $\mathbf{H}$ using matrices $\tilde{\mathbf{W}}$ and $\mathbf{G}_H$, or $\tilde{\mathbf{H}}$ and $\mathbf{G}_W$, which takes place at Lines 6 and 11. This corresponds to Lines 3 and 4 of Algorithm 1, and can be computed locally at each process p by executing NNLS algorithm on dense matrices $\tilde{\mathbf{W}}(\mathcal{I}_p,:)$ and $\mathbf{G}_H$, or $\tilde{\mathbf{H}}(:,\mathcal{J}_p)$ and $\mathbf{G}_W$, to obtain new $\mathbf{W}(\mathcal{I}_p,:)$ or $\mathbf{H}(\mathcal{I}_p,:)$, respectively, as described in Section 2.3.

The first type of communication in Algorithm 2 pertains to an ALL-REDUCE of a dense matrix of fixed size $k \times k$ at Lines 5 and 10, and the cost of this step is typically negligible in compare to the rest. The other two communication types involve (i) transferring the partial row results of $\tilde{\mathbf{W}}$ and $\tilde{\mathbf{H}}$ to their owner processes at Lines 4 and 9 to accumulate at the owners, (ii) sending the updated rows of $\mathbf{W}$ and $\mathbf{H}$ to processes in need at Lines 2 and 7. We respectively call these steps *fold* and *expand* communications, following the convention used by the sparse matrix community. The way these two communications are carried out plays a vital role in obtaining parallel scalability as they dominate the communication cost of the algorithm.

4.2 Communication Scheme

Collective communication (COL): MPI-FAUN employs collective communication strategies for both expand and fold steps of Algorithm 2 for dense as well as sparse $\mathbf{A}$, and partitions $\mathbf{A}$ using a uniform checkerboard topology. In this strategy, the rows and the columns indices $1,...,m$ and $1,...,n$ are divided into P_r and P_c ($P = P_r P_c$) sets $I_1,...,I_{P_r}$ and $J_1,...,J_{P_c}$ of equal size and having contiguous indices. Here, process p owns the matrix subblock $\mathbf{A}_{(r,c)}$, where $r = \lfloor P/P_c \rfloor + 1$ and $c = (P \mod P_c) + 1$,

Figure 1: A 5x5 checkerboard partition of a sparse matrix.
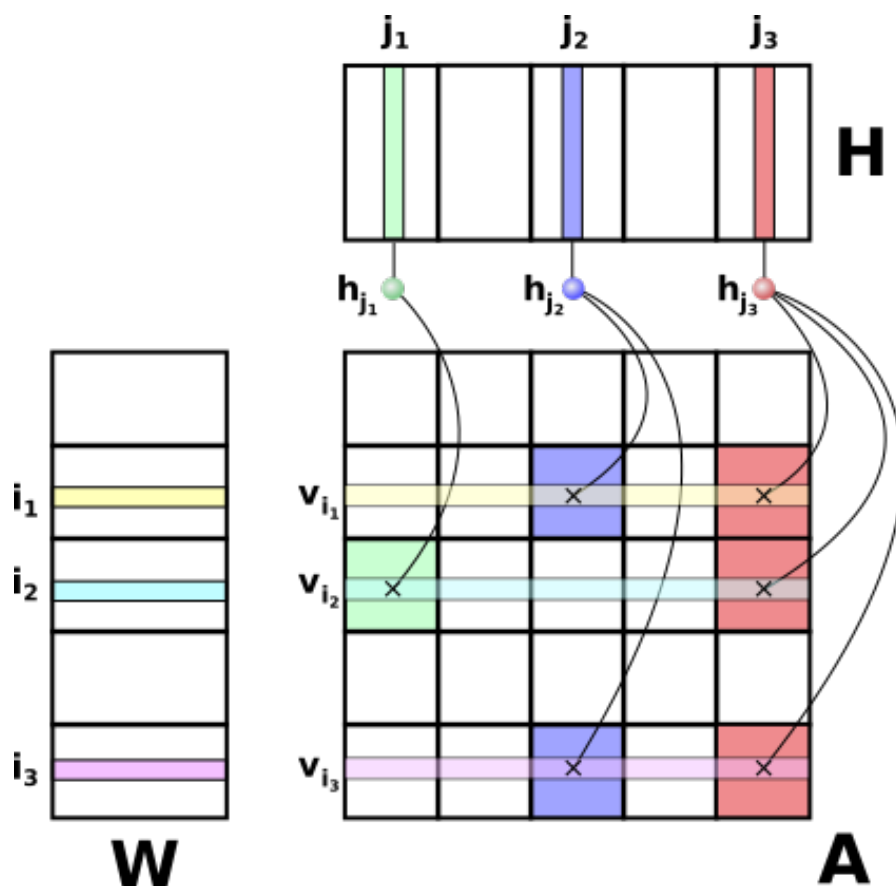
as well as m/P and n/P rows of $\mathbf{W}(I_r,:)$ and $\mathbf{H}(J_c,:)$. As $\mathbf{A}_{(r,c)}$ only touches the rows/columns of processes in the same row/column of the processor grid, the communication of $\mathbf{W}$ and $\mathbf{H}$ are performed within each process row and column using ALL-GATHER and REDUCE-SCATTER routines, in which the process p receives all matrix rows $\mathbf{W}(I_r,:)$ and $\mathbf{H}(J_c,:)$ belonging to processes in the same row and column of the process grid. Despite being favorable due to small number of exchanged messages in collective routines, in this strategy processes might receive rows that they do not need in their local sparse matrix dense matrix multiplication (particularly if $\mathbf{A}$ is very sparse), and this redundancy dramatically increases the communication volume, thus preventing scalability.

Point-to-point communication (P2P): HYPERTENSOR employs point-to-point communication for fold and expand steps by precomputing set of processes having a row/column in its footprint for each row/column of $\mathbf{W}/\mathbf{H}$. This reduces the communication volume at the cost of increased number of messages with respect to the strategy of MPI-FAUN.

4.3 Partitioning

Algorithm 2 requires a partitioning of the nonzeros of $\mathbf{A}$ as well as the rows and the columns of $\mathbf{W}$ and $\mathbf{H}$, and these three partitions completely determine its computational and communication costs. Here,

we compare different partitioning strategies, employed by Mpi-Faun, HyperTensor, and SpMV kernels, and argue how they relate to these two performance metrics.

4.3.1 Partitioning A

Checkerboard hypergraph partitioning (CH2) A hypergraph consists of vertices with associated weights and hyperedges that connect two or more vertices. In the literature, a hypergraph is typically formed by adding a vertex for each computational task with the associated execution cost, adding a hyperedge for each data element, and connecting the vertex to a hyperedge whenever the associated task and the data are dependent. Then, the vertices of the hypergraph is partitioned using a hypergraph partitioner to distribute vertex loads to parts equitably while reducing a metric called *cutsize*, which amounts to minimizing the total number of different parts each hyperedge connects. This corresponds in the actual computation to minimizing the data dependencies between tasks, hence the communication volume.

Traditional checkerboard hypergraph partitioning aims to partition the matrix $\mathbf{A}$ into P_r row slices first, and P_c column slices next to obtain an $P_r \times P_c$ checkerboard partition [3, 1]. The first partitioning phase is done using a column-net hypergraph model, in which for each row $\mathbf{A}(i,:)$, a vertex v_i with weight equaling to the number of nonzeros in $\mathbf{A}(i,:)$ is created. Each column j is represented with a hyperedge h_j and for each nonzero $(i,j) \in \mathbf{A}$, which implies a dependency to $\mathbf{H}(:,j)$ in computing $\tilde{\mathbf{W}}(i,:)$ at Line 3, we connect v_i to h_j. This hypergraph is partitioned into P_r parts giving the row partition of the checkerboard topology. The second partitioning phase uses a row-net hypergraph model induced by this row partition, where each column is represented with a vertex with P_r weights corresponding to the number of nonzeros in that column in all P_r row segments. Partitioning this hypergraph into P_c parts finalizes the $P_r \times P_c$ checkerboard partitionin by balancing the weights (number of nonzeros of $\mathbf{A}_{(r,c)}$) of each part while minimizing the communication volume. In the context of NMF, one issue arises when the matrix has some variance in the number of nonzeros in its rows/columns, which in turns yields unbalanced row/column strides. This in turn creates an imbalance in the NNLS computations as rows/cols of $\mathbf{W}/\mathbf{H}$ are partitioned to the processes in the same stride. To alleviate this issue, we modify this scheme slightly as follows. In both row and column partitioning phases, we add an additional constant weight to vertices. Balancing this additional constraint in hypergraph partitioning is expected to prevent such imbalanced strides. This partitioning model (which we call **CH2**) succesfully grasps the computation (both SpMM and NNLS) and communication requirements using checkerboard topology for sparse NMF, yet is costly to compute in practice due to high number of constraints ($P_r + 1$) in the row-net hypergraph.

1D-like checkerboard hypergraph partitioning (CH1) This variant partitions rows same as **CH2**, then partitions columns randomly to avoid multi-constraint partitioning. Random column partition provides load and communication balance yet increases the communication volume for the rows of $\mathbf{W}$.

Randomized checkerboard partitioning(CRD) This scheme corresponds to partitioning both the rows and the columns of $\mathbf{A}$ into P_r and P_c segments randomly. It is expected to provide good load and communication balance both in sparse and dense matrix operations, but it overlooks the communication volume.

Uniform checkerboard partitioning(CN) This partitioning variant forms an $P_R \times P_C$ partition of $\mathbf{A}$ by putting a contiguous set of m/P_R and n/P_C rows and columns in each slice. $\mathbf{W}$ and $\mathbf{H}$ are partitioned conformally with this topology; each process is assigned a contiguous set of $m/P_R P_C$ and $n/P_R P_C$ rows and columns of $\mathbf{W}$ and $\mathbf{H}$. This is the partitioning scheme employed by Mpi-Faun [11, 12]. It provides perfect balance in NNLS step yet may incur high communication

cost. We also use a randomized variant (**FAUNRP**) of this scheme in which the rows and columns of **A** are permuted randomly to balance its nonzeros among parts.

Fine-grain hypergraph partitioning (FH)   This is the partitioning strategy employed by HY-PERTENSOR. It forms a fine-grain hypergraph involving a vertex for each nonzero ($\mathbf{A}(i,j)$) and a hyperedge for each row and column index $i = 1,...,m$ and $j = 1,...,n$. The resulting hypergraph is typically very large and is costly to partition, and unlike checkerboard variants fingerprint of processes are not restricted to a row/column stride.

4.3.2   Partitioning W and H

Once **A** is partitioned, one has to partition rows and columns of factor matrices to form the sets $\mathcal{I}_p$ and $\mathcal{J}_p$ in Algorithm 2. In doing so, we are interested in assigning rows and columns to processes equitably. For this purpose, we specify imbalance parameters α that correspond to maximum imbalance we allow in this partitioning; i.e., $|\mathcal{I}_p| \leq \alpha m/P$ and $|\mathcal{J}_p| \leq \alpha n/P$ for each process p, and set $\alpha = 1.05$ in the experiments.

Next, for each row and column of **W** and **H** we create a list of processes that has a dependency to that row or column, which corresponds to processes owning the matrix blocks of same color in Fig. 1. Finally, we randomly assign each row and column to one of the processes satisfying the imbalance constraint in this list. If all processes in the list are overloaded, we assign it to the process that has the minimum number of rows/columns assigned. For a checkerboard partition, the minimum is always chosen from the same processor row/column so that 2D communication topology is not disturbed. Note that such an assignment increases the communication volume due to that row or column by 1; hence, in general smaller imbalance parameters yield larger communication volume due to increasing this type of assignment. On the other hand, we desire to keep α small as it pertains to the load imbalance in the NNLS step.

5   Experiments

In this section, we compare our algorithm DIST-SPNMF against MPI-FAUN, and compare its parallel performance on two big sparse matrices formed from real world datasets. We analyze and compare the computation and communication timings of these algorithms on a smaller cluster, then test the scalability limits of our method on a large supercomputing environment.

5.1   Experimental Setup

In this paper, we use both synthetic and realworld datasets. The synthetic datasets are used to evaluate the communication strategies with the increase in number of non-zeros of the sparse matrices. We used PaToH [3] for partitioning hypergraphs.

5.1.1   Datasets

Synthetic: For synthetic sparse matrices, we used the popular Kronecker generator from Graph500 benchmark (http://www.graph500.org/). The graph generator is a Kronecker generator similar to the Recursive MATrix (R-MAT) scale-free graph generation algorithm [4]. This model recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes edges within these partitions with unequal probabilities. Initially, the adjacency matrix is empty, and edges are added one at a time. The parameters to the generator are the number of vertices N and the Edge Factor (ef) that defines the ratio of the graph's edge count to its vertex count (i.e., half the average

degree of a vertex in the graph). Typically, the number of edges M of the scale free graph will be $ef \times N$. In our case, we consider $2^{20} (\approx 1.05$ million) vertices and set of edge factor $ef = \{4,8,16,32,64\}$.

Real World: We use two datasets from Flickr.com and Delicious.com that involves images tagged with different labels by users. The rows of the matrix correspond to different images, whereas the columns of the matrix represent different tags. The value of each nonzero $a_{i,j} \in \mathbf{A}$ indicates the number of unique users that tagged the image i with the tag j. Flickr and Delicious matrices are of size 28Mx1.6M and 17Mx2.5M, and have 112M and 72M nonzero elements, respectively. The current implementation of Mpi-Faun can only operate when P_R and P_C can divide m and n, hence we trimmed the matrices slightly.

5.1.2   Implementation Platform

We conducted our experiments on two different parallel computing platforms. The first platform is the "Rhea" cluster at the Oak Ridge Leadership Computing Facility (OLCF), which is a commodity-type Linux cluster with a total of 512 nodes and a 4X FDR Infiniband interconnect. Each node contains dual-socket 8-core Intel Sandy Bridge-EP processors operating at 2GHz clock frequency and 128 GB of memory. Each socket has a shared 20MB L3 cache, and each core has a private 256K L2 cache. There, we ran our experiments up to 3072 cores, which is the maximum allowed in the cluster. The second platform is an IBM BlueGene/Q supercomputer consisting of 6 racks each having 16384 cores. Each compute node has 16GB of memory and single socket 16-core PowerPC A2 processor at 1.6GHz clock frequency with 16KB of L1 cache per core, and 32MB shared L2 cache. We ran both algorithms using 16 MPI ranks per node, and set $P_c = 16$ in all partitionings.

Our code for local matrix operations is developed using the matrix library Armadillo [25]. We use BLAS and LAPACK for dense matrix operations by linking Armadillo with Intel MKL, Open-BLAS [31], or any other BLAS and LAPACK implementation. Both codes are compiled using the default GNU C++ Compiler (g++ (GCC) 5.3.0) and MPI library (Open MPI 1.8.4) on RHEA, and Clang compiler (3.5.0) with IBM MPI library on BlueGene/Q.

5.2   Effect of communication scheme

To understand the effect of partitioning and its impact on communication and computation, we performed experiments on synthetic Kronecker graph and the results are presented in Fig. 2. For 2^{20} vertices, we chose the edge factor of the kronecker graph as $\{4,8,16,32,64\}$ and in this setting as the edge factor increases, higher the number of non-zeros making the sparse matrix denser. We ran the baseline FAUN algorithm and the proposed algorithm with different partitioning scheme as explained in Section 4.3 on 4096 processors using a 64×64 processor grid on Rhea for low rank $k = 48$.

In Fig. 2, we provide per-iteration computation and communication timings for different partitionings and communication schemes using 4096 processors and a $64x64$ processor grid. **COL** is the is the implementation of Mpi-Faun and is the only instance that uses collective communication. Comparing it with the **CN** that also uses uniform partitioning but with point-to-point communication. **CR** decides the checkerboard topology randomly and uses point-to-point scheme as well. **CH1** and **CH2** correspond to 1D-like and 2D checkerboard hypergraph partitioning models with point-to-point communication. Finally, **FH** is the fine-graph hypergraph partitioning with point-to-point scheme, for which we only provide partition statistics for comparison and not the actual run time.

We notice in Fig. 2 that **COL** always incurs significantly higher communication cost. As the edge factor increases and matrix becomes denser, the communication cost of **COL** does not change as expected since it does not depend on the matrix sparsity. The communication cost of **CN** progressively increases with the edge factor, but even with an edge factor of 64 **COL** is about two times more costly. Therefore, we conclude that even though there is a converging trend between the cost of collective and

Table 2: Load balance and communication statistics for 4096-way partitioning with edge factor 64.

Partitioning	nz-ib	row-ib	col-ib	com-max.	com-avg
CN	1.30987	1	1.00392	15366	11876
CH1	1.21477	1.27734	1.05098	20800	11038
CH2	1.06479	1.30469	1.53725	23484	10853
CR	1.38681	1.04688	1.05098	15475	11341
FH	1.00115	1.05078	1.0549	47925	13802

point-to-point communication, it is a rather slow convergence, and the point-to-point scheme should be the method of choice unless matrix gets very dense.

5.3 Effect of partitioning strategies

In Fig. 2b, our first observation is that all partitionings except **COL** yield comparable results in terms of communication, while there is some variation in the computation times provided in Fig. 2a. **CN** gives similar computation time to **CR** since matrix is randomly permuted. **CH1** yields slightly higher computation time and no notable improvement in the communication time, as it involves multi-constraint partitioning with too many constraints, which is a difficult partitioning problem that partitioners such as PaToH might not solve very effectively. **CH2** provides some advantage both in terms of computation and communication time, and this partitioning is feasible to compute as it involves hypergraph partitioning with only two constraints.

We conclude that even though **CH2** correctly models computation and communication costs of distributed sparse NMF, it does not yield better results in practice due to having too many constraints in the partitioning problem, which also renders it impractical due to costly precomputation for partitioning. **CH1** yields a smaller hypergraph with reasonable partitioning cost and provides some benefits in terms of communication reduction. **CR** seems to produce good results overall as it provides good computation and communication balance. We provide partition statistics with respect to these strategies in Table 2 which justify these observations. We observe that **FH** incurs a significant communication imbalance with respect to other methods preventing scalability and increasing memory consumption of the bottleneck processor, and these results confirm those reported in [14].

5.4 Strong scaling

In this section, we provide strong scalability experiments on real-world datasets in the next section with a detailed comparison of point-to-point and collective strategies as well has randomized and hypergraph checkerboard (1d-like) partitioning.

In this experiment, we considered the following algorithms and partitionings:

- **FAUN**: MPI-FAUN algorithm [11, 12] with uniform partitioning (**FAUN**) where each process holds an input matrix of size $m/P_r \times n/P_c$.

- **FAUNRP**: The partitioning strategy in **FAUN** could result in a significant computational load imbalance in with a skewed nonzero distribution of **A**. We alleviate this by randomly permuting the rows and columns of the matrix before executing MPI-FAUN, and call this scheme **FAUNRP**.

- **P2PHP**: DIST-SPNMF (Algorithm 2) with 1D-like checkerboard hypergraph partitioning explained in Section 4.3.1.

- **P2PRP**: DIST-SPNMF (Algorithm 2) with randomized checkerboard partioning explained in Section 4.3.1.

In Fig. 3a we show the speedup results of all four instances on the Rhea cluster using up to 3072 MPI ranks/cores on Flickr data. The speedup values are with respect to slowest runtime among all four instances using 16 cores (single node). We observe in Fig. 3a that all algorithms scale up to 1536 cores, yet MPI-FAUN instances achieve this with significantly lower parallel efficiency. This mostly is due to higher communication costs involved in the all-to-all communication scheme for both instances. We also realize that **FAUNRP** significantly improves the runtime with respect to **FAUN**, meaning that **FAUN** indeed causes load imbalance in partitioning nonzeros of **A**. At 3072 processes, both **FAUNRP** and **FAUN** lose scalability and slow down, whereas **P2PHP** and **P2PRP** scale to 3072 processors.

Similarly, in Fig. 4a we provide the same results for the Delicious matrix. We observe a similar trend in the comparisons of different methods, except that **FAUNRP** and **FAUN** scale even worse in this case. Our algorithm also loses scalability after 1536 processes, and similarly to the previous case **P2PHP** starts slower than **P2PRP** due to load imbalance, and catches up for $k=48$. These two test cases clearly show that employing a point-to-point communication with good partitionings is essential for obtaining high performance in NMF algorithm.

To better test the scalability limit of our algorithms, we ran them on an IBM BlueGene/Q supercomputer up to 32768 processors using the same two matrices. The results of these two experiments are provided in Figs. 3b and 4b. Our algorithm graciously scales up to 16384 cores in all four instances, and **P2PHP** manages to slightly improve the runtime using 32768 cores on Flickr, while all other slowing down using 32768 ranks. Again, **P2PHP** is slower than **P2PRP** using lower number of processors as the communication cost is negligible in these instances, and **P2PHP** introduces worse load balance than **P2PRP**. However, using 32768 processors **P2PHP** manages to outrun **P2PRP** by incurring less communication.

5.5   Time Breakdown Per Iteration

In this section we provide the time spent on each individual operation type and communication within an NMF iteration. We report the averages over 30 iterations on Rhea, and 10 iterations on BlueGene/Q for each of four runs. As provided in Algorithm 2 there are three types computations and two types of communications within an NMF iteration, and we present timings for these steps with the following labels:

- **Gram**: Computing the local contribution to the Gram matrix, and performing an ALL-REDUCE to gather the final result.

- **MM**: Computing the sparse matrix-dense-matrix multiplication using $\mathbf{A}_p$ and one of the factor matrices.

- **LUC** : Local NNLS computation to compute the final value of the factor matrix.

- **Comm**: Total expand and fold communication time in the case of **P2PRP** and **P2PHP**, and the total time spent on ALL-GATHER and REDUCE-SCATTER steps for **FAUN** and **FAUNRP**.

In our results, we do not distinguish the costs of these tasks for **W** and **H** separately; we instead report their sum.

We report the time breakdown for Flickr and Delicious datasets in Fig. 5, Fig. 6 for Rhea and Fig. 7 for BlueGene/Q. For each cluster and data set, we show the timings for the smallest and the largest number of processors used. Our objective in this experiment is to better analyze the speedup results and by comparing the computational and communication costs of different communication schemes and partitionings.
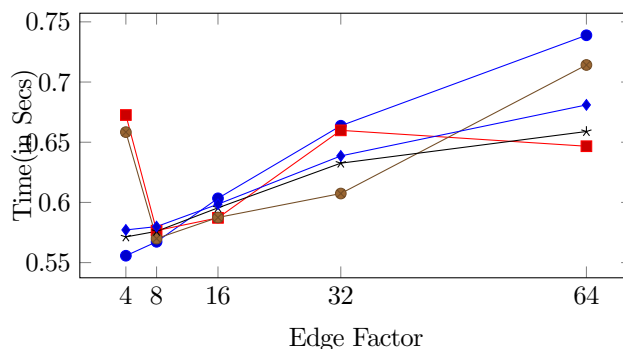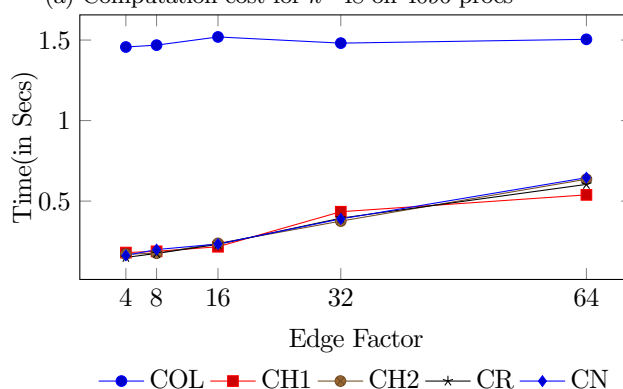
(a) Computation cost for $k=48$ on 4096 procs

COL — CH1 — CH2 — CR — CN

(b) Communication cost for $k=48$ on 4096 procs

Figure 2: Communication and computation costs of different partitioning strategies on Kronecker graphs on 4096 processors Rhea for $k=48$

Flickr on Rhea: We observe in Fig. 5 that in one node configuration with $p=16$, the **FAUN** and **FAUNRP** performs similar to **P2PRP** and **P2PHP** in terms of computation, and the communication time takes a small portion of the execution in all instances. As the number of processes increases to 3072, the communication time of **P2PRP** and **P2PHP** stays reasonably low, whereas in the case of **FAUN** and **FAUNRP**, we clearly observe that the communication cost dominates the execution time using both low rank values $k=48$. Randomization offers load balance to **FAUNRP** which gives it a slight edge over **FAUN**, yet both instances suffer from the high communication cost associated with the all-to-all communication strategy, which explains the drop in the scalability results.

Delicious on Rhea: In Fig. 6, we see that **P2PRP** and **P2PHP** perform better than **FAUN** even in single node configuration. Fig. 6 shows that **FAUN** takes twice more than **P2PRP** and **P2PHP** in the sparse matrix multiplication step, highlighting the skewed distribution of the matrix nonzeros, which is alleviated to a certain extend by randomly permuting the matrix. Similar to Flickr data, using 3072 processors, **P2PRP** and **P2PHP** perform significantly better than **FAUN** and **FAUNRP**, whose iteration times are dominated by the communication.

Flickr and Delicious on BlueGene/Q: In Fig. 7 we give the timings for computation and communication steps using our methods with two different partitionings of matrices on BlueGene/Q. We observe
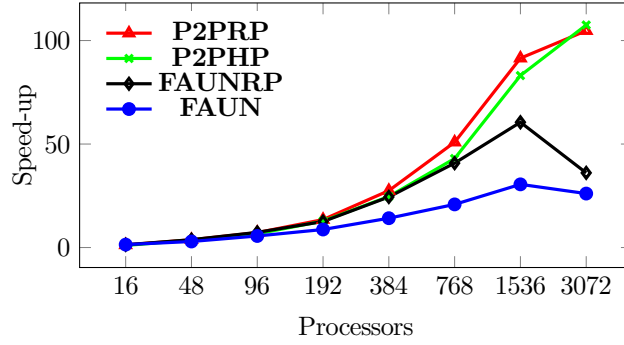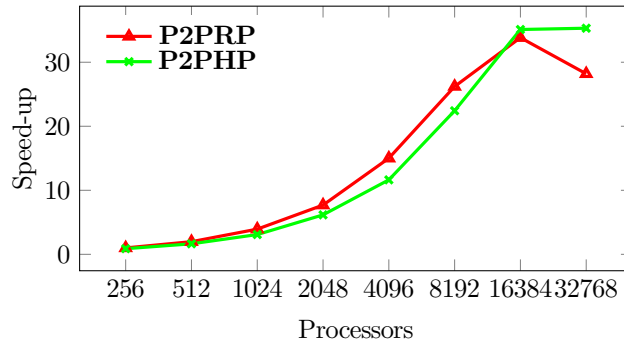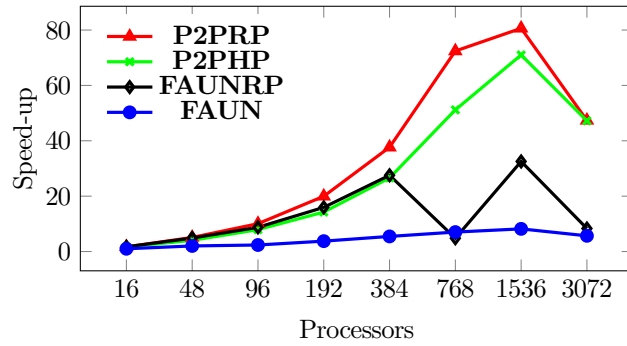
(a) Flickr dataset $k\!=\!48$ on Rhea

(b) Flickr dataset $k\!=\!48$ on Bluegene/Q
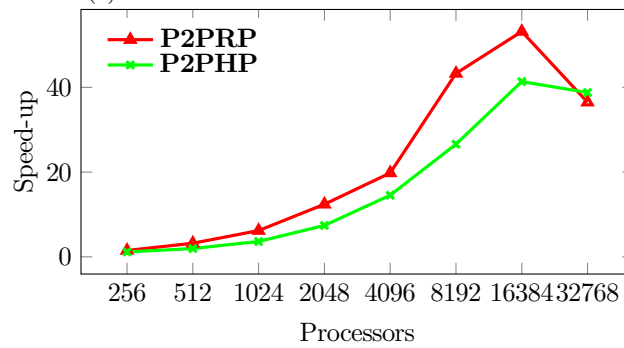
Figure 3: Strong scaling on Flickr dataset

that using 512 processors, communication cost is negligible, and **P2PRP** beats **P2PHP** thanks to better load balance. Using 16384 processors, however, on Flickr matrix **P2PHP** gets faster than **P2PRP** due to significant reduction in the communication volume. On Delicious matrix, **P2PHP** similarly better reduces the communication, yet this is outweighed by the load imbalance in matrix multiplications.

6   Conclusion

In this paper, we compared various partitioning and communication strategies used in the literature in the context of non-negative matrix factorization. We showed that an important difference in the parallel NMF algorithms is balancing matrix rows among processors, and this constraint renders state-of-the-art hypergraph partitioning methods less effective. We employed variations of the MPI-FAUN implementation with point-to-point communication, and concluded that unless matrix at hand is quite dense, point-to-point communication significantly improves the scalability by reducing the communication volume. With optimized implementations, we achieved scalability up to 32K nodes on a BG/Q supercomputer using partitioning schemes that are cheap to compute. To the best of our knowledge, our work is the first high performance implementation of distributed NMF that takes the sparsity of the input matrix into consideration in communication to reduce the communication cost, and employs effective partitionings to further enhance parallel scalability. Our immediate next steps for extending our work involve adding shared memory parallelism to obtain further speedup.
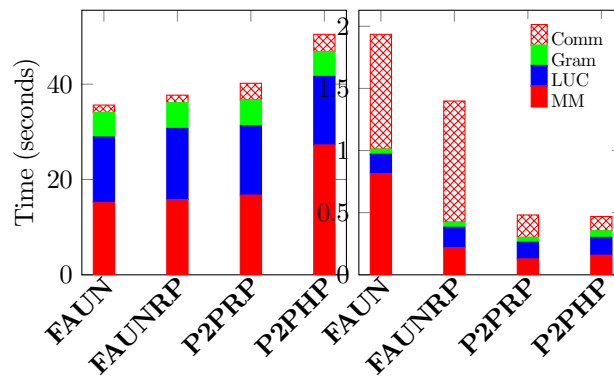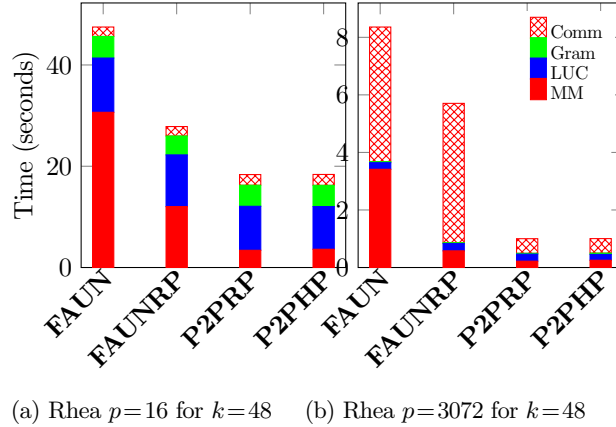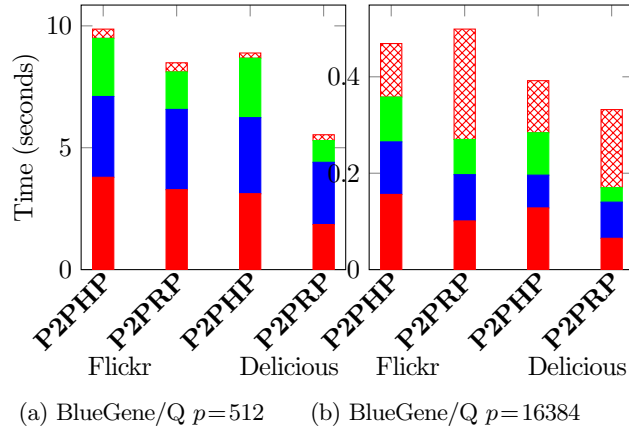
(a) Delicious dataset $k=48$ on Rhea

(b) Delicious dataset $k=48$ on Bluegene/Q

Figure 4: Strong scaling on Delicious dataset

(a) Rhea $p=16$ for $k=48$ (b) Rhea $p=3072$ for $k=48$

Figure 5: Flickr dataset Time Breakdown for $k=48$ on Rhea

(a) Rhea $p\!=\!16$ for $k\!=\!48$ (b) Rhea $p\!=\!3072$ for $k\!=\!48$

Figure 6: Delicious dataset Time Breakdown for $k\!=\!48$ on Rhea

(a) BlueGene/Q $p\!=\!512$ (b) BlueGene/Q $p\!=\!16384$

Figure 7: Time Breakdown of Flickr and Delicious for $k\!=\!48$ on Bluegene/Q. The left two bars are for the Flickr matrix, and the right two bars are for the Delicious matrix

References

[1] C. AYKANAT, B. B. CAMBAZOGLU, AND B. UCAR, *Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 609–625.

[2] D. P. BERTSEKAS, *Nonlinear programming*, (1999).

[3] U. V. CATALYUREK AND C. AYKANAT, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[4] D. CHAKRABARTI, Y. ZHAN, AND C. FALOUTSOS, *R-mat: A recursive model for graph mining*, in Proceedings of the 2004 SIAM International Conference on Data Mining, SIAM, 2004, pp. 442–446.

[5] A. CICHOCKI, R. ZDUNEK, A. H. PHAN, AND S.-I. AMARI, *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation*, Wiley, 2009.

[6] C. FALOUTSOS, A. BEUTEL, E. P. XING, E. E. PAPALEXAKIS, A. KUMAR, AND P. P. TALUKDAR, *Flexi-FaCT: Scalable flexible factorization of coupled tensors on Hadoop*, in Proceedings of the SDM, 2014, pp. 109–117.

[7] A. GITTENS, A. DEVARAKONDA, E. RACAH, M. F. RINGENBURG, L. GERHARDT, J. KOTTALAM, J. LIU, K. J. MASCHHOFF, S. CANON, J. CHHUGANI, P. SHARMA, J. YANG, J. DEMMEL, J. HARRELL, V. KRISHNAMURTHY, M. W. MAHONEY, AND PRABHAT, *Matrix factorization at scale: a comparison of scientific data analytics in spark and C+MPI using three case studies*, CoRR, abs/1607.01335 (2016).

[8] D. GROVE, J. MILTHORPE, AND O. TARDIEU, *Supporting array programming in X10*, in Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY'14, 2014, pp. 38:38–38:43.

[9] N. GUAN, D. TAO, Z. LUO, AND B. YUAN, *Nenmf: An optimal gradient method for nonnegative matrix factorization*, IEEE Transactions on Signal Processing, 60 (2012), pp. 2882–2898.

[10] P. O. HOYER, *Non-negative matrix factorization with sparseness constraints*, JMLR, 5 (2004), pp. 1457–1469.

[11] R. KANNAN, G. BALLARD, AND H. PARK, *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, New York, NY, USA, February 2016, ACM, pp. 9:1–9:11.

[12] R. KANNAN, G. BALLARD, AND H. PARK, *MPI-FAUN: an mpi-based framework for alternating-updating nonnegative matrix factorization*, CoRR, abs/1609.09154 (2016).

[13] R. KANNAN, G. B. BALLARD, AND H. PARK, *MPI-FAUN: An MPI-based framework for alternating-updating nonnegative matrix factorization*, IEEE Transactions on Knowledge and Data Engineering, 30 (2018), pp. 544–558.

[14] O. KAYA AND B. UÇAR, *Scalable sparse tensor decompositions in distributed memory systems*, in Proceedings of SC, ACM, 2015, pp. 77:1–77:11.

[15] H. KIM AND H. PARK, *Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis*, Bioinformatics, 23 (2007), pp. 1495–1502.

[16] J. KIM, Y. HE, AND H. PARK, *Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework*, Journal of Global Optimization, 58 (2014), pp. 285–319.

[17] D. KUANG, C. DING, AND H. PARK, *Symmetric nonnegative matrix factorization for graph clustering*, in Proceedings of SDM, 2012, pp. 106–117.

[18] D. KUANG, S. YUN, AND H. PARK, *SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering*, Journal of Global Optimization, (2013), pp. 1–30.

[19] R. LIAO, Y. ZHANG, J. GUAN, AND S. ZHOU, *CloudNMF: A MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets*, Genomics, proteomics & bioinformatics, 12 (2014), pp. 48–51.

[20] C. LIU, H.-C. YANG, J. FAN, L.-W. HE, AND Y.-M. WANG, *Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce*, in Proceedings of the WWW, ACM, 2010, pp. 681–690.

[21] Y. LOW, D. BICKSON, J. GONZALEZ, C. GUESTRIN, A. KYROLA, AND J. M. HELLERSTEIN, *Distributed GraphLab: A framework for machine learning and data mining in the cloud*, Proc. VLDB Endow., 5 (2012), pp. 716–727.

[22] E. MEJÍA-ROA, D. TABAS-MADRID, J. SETOAIN, C. GARCÍA, F. TIRADO, AND A. PASCUAL-MONTANO, *NMF-mGPU: non-negative matrix factorization on multi-GPU systems*, BMC bioinformatics, 16 (2015), p. 43.

[23] X. MENG, J. BRADLEY, B. YAVUZ, E. SPARKS, S. VENKATARAMAN, D. LIU, J. FREEMAN, D. B. TSAI, M. AMDE, S. OWEN, D. XIN, R. XIN, M. J. FRANKLIN, R. ZADEH, M. ZAHARIA, AND A. TALWALKAR, *MLlib: Machine Learning in Apache Spark*, May 2015.

[24] V. P. PAUCA, F. SHAHNAZ, M. W. BERRY, AND R. J. PLEMMONS, *Text mining using nonnegative matrix factorizations*, in Proceedings of SDM, 2004.

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

[26] N. SATISH, N. SUNDARAM, M. M. A. PATWARY, J. SEO, J. PARK, M. A. HASSAAN, S. SENGUPTA, Z. YIN, AND P. DUBEY, *Navigating the maze of graph analytics frameworks using massive graph datasets*, in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, 2014, pp. 979–990.

[27] D. SEUNG AND L. LEE, *Algorithms for non-negative matrix factorization*, NIPS, 13 (2001), pp. 556–562.

[28] S. R. SUKUMAR, R. KANNAN, S. LIM, AND M. A. MATHESON, *Kernels for scalable data analysis in science: Towards an architecture-portable future*, in 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016, 2016, pp. 1026–1031.

[29] S. R. SUKUMAR, M. A. MATHESON, R. KANNAN, AND S. LIM, *Mini-apps for high performance data analysis*, in 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016, 2016, pp. 1483–1492.

[30] D. L. SUN AND C. FÉVOTTE, *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), May 2014, pp. 6201–6205.

[31] Z. XIANYI, *Openblas*, Last Accessed 03-Dec-2015.

[32] J. YIN, L. GAO, AND Z. ZHANG, *Scalable nonnegative matrix factorization with block-wise updates*, in Machine Learning and Knowledge Discovery in Databases, vol. 8726 of LNCS, 2014, pp. 337–352.

[33] H. YUN, H.-F. YU, C.-J. HSIEH, S. VISHWANATHAN, AND I. DHILLON, *Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion*, Proceedings of the VLDB Endowment, 7 (2014), pp. 975–986.

[34] M. ZAHARIA, M. CHOWDHURY, M. J. FRANKLIN, S. SHENKER, AND I. STOICA, *Spark: Cluster computing with working sets*, in Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, USENIX Association, 2010, pp. 10–10.