Partitioning and Communication Strategies for Sparse Non-negative Matrix Factorization

Oguz Kaya
Inria Bordeaux
Talence, France
oguz.kaya@inria.fr

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

Grey Ballard
Wake Forest University
Winston-Salem, NC
ballard@wfu.edu

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 unmixing 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. 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 one that uses a Cartesian, or 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 see that our proposed strategy runs up to 10x faster than the state of the art on real-world datasets.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**; **Massively parallel algorithms**;

KEYWORDS

sparse/dense matrix multiplication (SpMM), hypergraph partitioning, distributed-memory parallel computing

ACM Reference Format:
Oguz Kaya, Ramakrishnan Kannan, and Grey Ballard. 2018. Partitioning and Communication Strategies for Sparse Non-negative Matrix Factorization. In *Proceedings of 47th International Conference on Parallel Processing (ICPP 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225127

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225127

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 $\mathbf{A} \approx \mathbf{WH}$. 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

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

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 [26], computer vision [11], and bioinformatics [16] to blind source separation [5], unsupervised clustering [19, 20], 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 with highly irregular nonzero patterns.

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 $\mathbf{W}^T\mathbf{A}$ or $\mathbf{AH}^T$, computing the Gram matrix $\mathbf{W}^T\mathbf{W}$ or $\mathbf{HH}^T$, 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 $\mathbf{AH}^T$, 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 their costs 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.

To solve this problem, the current work builds on and combines approaches from two existing libraries. The Mpi-Faun [13] software framework enables computing 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 communicate more data than necessary, as some processors will receive data that they do not need for local computation.

The HyperTensor [15] library is designed for sparse tensor factorization, which involves computation resembling NMF kernels with sparse irregular computational patterns, and uses hypergraph partitioning to distribute the data and computation across processors. The main advantage of this approach is that it minimizes the communication cost of the SpMM step and that it employs a point-to-point communication scheme that 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 imbalance. Hence, the partitioning strategy considering only SpMM will result in an imbalance in the overall workflow, especially 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 and proposing a synthesis of the communication and partitioning strategies that enables scalability to thousands of processors. In particular, our proposed approach

- uses point-to-point communication within SpMM, moving data to only those processors that need it,
- uses randomized Cartesian (or checkerboard) partitioning of the input matrix, load balancing the local SpMM computations and avoiding upfront partitioning cost,
- load balances the row distribution of output matrices, maintaining efficient NLS updates, and
- achieves up to a 10x decrease in run time compared to the Mpi-Faun library on a tagged image (Delicious) dataset with 1536 processors.

2 SPARSE NMF

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, $\mathbf{A}_{ij}$ refers to the sub-block (i,j) of $\mathbf{A}$ in a 2D Cartesian 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 (AU-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. Kannan, Park and Ballard [13] have discussed this framework as in Algorithm 1 and different NMF algorithms that can be expressed under this framework.

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**) and in our algorithm referred as UpdateW and UpdateH. For consistency, we borrow these acronyms from previous work [12, 13].

$\mathbf{A}$	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

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 $\mathbf{H}\mathbf{H}^T$ and $\mathbf{A}\mathbf{H}^T$
4: Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$

We note that AU-NMF is an instance of a two-block, block coordinate descent (BCD) framework as explained by Bertsekas [2]. The BCD framework expresses solving optimization variables in complex 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 $\mathbf{W}$:

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

Since each subproblem involves non-negative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [17]. From the computational perspective as in Algorithm 1, each of these subproblems will require a Gram computation, an SpMM, and an **LUC**. The Multiplicative Update (**MU**) algorithm proposed by Lee and Seung [29] is the most common NMF algorithm as it is easier to implement and available through many standard packages. But there are more recent algorithms that perform better than **MU** such as Hierarchical Alternating Least Squares (**HALS**) [5] and Block Principal Pivoting (**ABPP**) to solves these NLS subproblems.

These updates differ in the choice of blocks to update, considering either the entire factor matrix, a single column vector, or a single element as a block. The convergence properties of these different NMF algorithms are discussed in detail by Kim, He and Park [17]. While

we focus only on the most common **MU** algorithm in this paper, we highlight that our algorithm is not restricted to this choice. The parallel framework is seamlessly extensible to other NMF algorithms as well, including **HALS**, **ABPP**, Alternating Direction Method of Multipliers (**ADMM**) [33], and Nesterov-based methods [10].

2.3 Multiplicative Update (MU)

We support ℓ_1 and ℓ_2 regularization on both $\mathbf{W}$ and $\mathbf{H}$. ℓ_2 tames the growth of values and ℓ_1 makes it insensitive to smaller values. Typically, $\mathbf{W}$ is a dense basis matrix and $\mathbf{H}$ is the sparse projection of samples on this basis. In real world, there will be fewer components that participate on a sample [14] and hence $\mathbf{H}$ is sparse with ℓ_1 regularization. Consider, if the value of j^{th} component of an i^{th} sample $h_i j = 0.001$, we can safely assume, the contribution of the j^{th} component to i^{th} sample is negligible. 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. It is beyond the scope of the paper to compare and contrast sparse NMF with and without regularization. Kim and Park discuss details on the interpretability of solutions and qualitative advantages of using ℓ_1 regularization for clustering sparse text data [18].

With these choices of regularization, the NMF problem becomes

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

The values α and β were fixed for the experiments, but in practice they need to be tuned for each application. In the case of **MU** [29], individual entries of $\mathbf{W}$ and $\mathbf{H}$ are updated with all other entries fixed. In this case, the update rules are

$$\begin{aligned} w_{ij} &\leftarrow w_{ij}\frac{(\mathbf{A}\mathbf{H}^T)_{ij}}{(\mathbf{W}(\mathbf{H}\mathbf{H}^T+2\beta\mathbf{1}_k))_{ij}}, \text{ and} \\ h_{ij} &\leftarrow h_{ij}\frac{(\mathbf{W}^T\mathbf{A})_{ij}}{((\mathbf{W}^T\mathbf{W}+2\alpha\mathbf{I}_k)\mathbf{H})_{ij}}. \end{aligned} \tag{4}$$

where $\mathbf{1}_k$ is a $k\times k$ matrix of all ones and $\mathbf{I}_k$ is a $k\times k$ identity matrix. These update equations are obtained by setting the first order partial derivatives of the objective function in Eq. (3) with respect to each matrix entry to zero. For more detailed analysis of these update equations, see [8, 17, 29].

After computing the Gram matrices $\mathbf{H}\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{W}$, adding the appropriate regularizers, and computing the products $\mathbf{A}\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{A}$, the extra cost of computing $\mathbf{W}(\mathbf{H}\mathbf{H}^T+2\beta\mathbf{1}_k)$ and $(\mathbf{W}^T\mathbf{W}+2\alpha\mathbf{I}_k)$ is $2(m+n)k^2$ flops to perform updates for all entries of $\mathbf{W}$ and $\mathbf{H}$, as the other element-wise operations affect only lower-order terms. The details about using AU-NMF in Algorithm 1 for **HALS** and **ABPP** are explained in [12, 13].

3 SURVEY

Matrix factorization is the problem of determining two smaller matrices called factors whose product approximates a given input matrix. In the case of NMF the factors are non-negative (all the entries are nonnegative). Recently, there has been a growing interest in Collaborative Filtering (CF) based recommender systems. One of the popular techniques for collaborative filtering is matrix factorization, and many open source implementations are available on off-the-shelf distributed machine learning libraries such as GraphLab [23], MLLib [25], and many others [28, 36]. As discussed by Kannan, Park and Ballard [13], CF using matrix factorization is a different problem than NMF: CF considers nonzeros in the matrix to be observed entries and zeros to be missing entries, while in the case of NMF, there are no missing entries (zeros signify observed entries).

There are several recent distributed NMF algorithms in the literature [6, 21, 35] for different objective losses such as KL divergence, squared loss, and "exponential" loss functions [22]. The building blocks of the **MU** algorithm are matrix multiplication, element-wise multiplication, and element-wise division. Liao et al. implement an open-source Hadoop-based **MU** algorithm and study its scalability on large-scale biological data sets [21] by performing distributed matrix operations. Similarly, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [35]. All of these works use Hadoop framework to implement their algorithms, hence are not very efficient. Spark is an in-memory MapReduce framework that also has a CF implementation in its open source project MLlib [25] using matrix factorization and that can impose non-negativity constraints as well. We do not compare our approach against this work because CF is different from NMF.

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 [12, 13], 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 (**ABPP**). 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 on 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. They claim that native MPI implementations on HPC platforms outperform Spark implementation by a factor of up to 44×. Similar observations have been made by Sukumar, Kannan, Matheson and Lim [31, 32] on supercomputers at Oak Ridge Leadership Computing Facility. Finally, there are implementations of the **MU** algorithm in a distributed memory setting using X10 [9] and on a GPU [24].

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
$\mathcal{I}_p, \mathcal{J}_p$: Set of rows/columns of $\mathbf{W}/\mathbf{H}$ owned by process p
$\mathcal{F}_p, \mathcal{G}_p$: Footprints of process p on $\mathbf{W}$ and $\mathbf{H}$
$\mathbf{W}(\mathcal{I}_p,:), \mathbf{H}(:,\mathcal{J}_p)$: Owned rows/columns of $\mathbf{W}/\mathbf{H}$
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: Comm-Expand($\mathbf{H}(:,\mathcal{G}_p)$)
3: $\tilde{\mathbf{W}}(\mathcal{F}_p,:) \leftarrow \mathbf{A}_p\mathbf{H}(:,\mathcal{G}_p)$
4: Comm-Fold($\tilde{\mathbf{W}}(\mathcal{F}_p,:)$)
5: $\mathbf{G}_H \leftarrow$ All-Reduce($\mathbf{H}(:,\mathcal{J}_p)\mathbf{H}(:,\mathcal{J}_p)^T$)
6: $\mathbf{W}(\mathcal{I}_p,:) \leftarrow$ UpdateW($\mathbf{G}_H, \tilde{\mathbf{W}}(\mathcal{I}_p,:)$)
7: Comm-Expand($\mathbf{W}(\mathcal{F}_p,:)$)
8: $\tilde{\mathbf{H}}(:,\mathcal{G}_p) \leftarrow \mathbf{W}(\mathcal{F}_p,:)^T\mathbf{A}_p$
9: Comm-Fold($\tilde{\mathbf{H}}(:,\mathcal{G}_p)$)
10: $\mathbf{G}_W \leftarrow$ All-Reduce($\mathbf{W}(\mathcal{I}_p,:)^T\mathbf{W}(\mathcal{I}_p,:)$)
11: $\mathbf{H}(\mathcal{J}_p,:) \leftarrow$ UpdateH($\mathbf{G}_W, \tilde{\mathbf{H}}(\mathcal{J}_p,:)$)
12: **until** convergence or maximum number of iterations

Parallelizing sparse NMF involves partitioning 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 **LUC** 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., $\mathbf{A}_1, \ldots, \mathbf{A}_P$ 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 an *expand* communication step where each process p gets the submatrices $\mathbf{H}(:,\mathcal{G}_p \setminus \mathcal{J}_p)$ and $\mathbf{W}(\mathcal{F}_p \setminus \mathcal{I}_p,:)$ that are accessed by entries of $\mathbf{A}_p$, and these steps are 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 are highlighted at Lines 3 and 8. Indeed, partial results for the submatrices $\tilde{\mathbf{W}}(\mathcal{F}_p \setminus \mathcal{I}_p,:)$ and $\tilde{\mathbf{H}}(:,\mathcal{G}_p \setminus \mathcal{J}_p)$ 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. This is realized in a *fold* communication step at Lines 4 and 9.

The second task is to compute the Gram matrices $\mathbf{G}_H = \mathbf{H}\mathbf{H}^T$ 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 UpdateW and UpdateH 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 $\mathbf{H}(\mathcal{I}_p,:)$.

In the case of **MU**, the update rules are given by Eq. (5). Therefore, for row and column index sets $\mathcal{I}_p$ and $\mathcal{J}_p$, then UpdateW and UpdateH computations are as follows:

$$\begin{aligned} \mathbf{W}(\mathcal{I}_p,:) &\leftarrow \mathbf{W}(\mathcal{I}_p,:) \circledast (\tilde{\mathbf{W}}(\mathcal{I}_p,:)) \oslash (\mathbf{W}(\mathcal{I}_p,:)(\mathbf{G}_H + 2\beta\mathbf{1}_k)), \\ \mathbf{H}(\mathcal{J}_p,:) &\leftarrow \mathbf{H}(\mathcal{J}_p,:) \circledast (\tilde{\mathbf{H}}(\mathcal{J}_p,:) \oslash (\mathbf{G}_W + 2\alpha\mathbf{I}_k)\mathbf{H}(\mathcal{J}_p,:)) \end{aligned} \tag{5}$$

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

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 (**COLL**).* 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 $1, \ldots, m$ and $1, \ldots, n$ are divided into P_r and P_c ($P = P_rP_c$) sets $I_1, \ldots, I_{P_r}$ and $J_1, \ldots, J_{P_c}$ 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 $c = (P \mod P_c) + 1$, as well as m/P rows of $\mathbf{W}(I_r,:)$ and n/P columns of $\mathbf{H}(:,J_c,)$. 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 columns $\mathbf{H}(:,J_c)$ belonging to processes in the same row and column

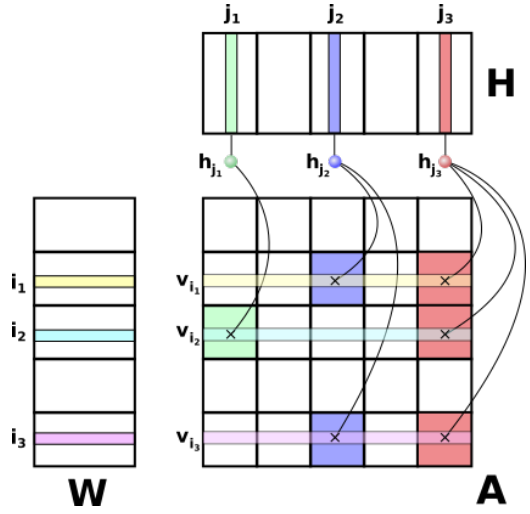


Figure 1: A 5x5 checkerboard partition of a sparse matrix.

of the process grid. Despite being favorable due to a 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 the 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* $\mathbf{A}$.

*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 [1, 3]. 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 partitioning 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 UpdateW and UpdateH 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**) successfully grasps the computation (both SpMM and **LUC**) 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 (**CUN**).* 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 [12, 13]. It provides perfect balance in UpdateW and UpdateH step yet may incur high communication cost. We also use a randomized variant (**CUR**) of this scheme in which the rows and columns of $\mathbf{A}$ are permuted randomly to balance its nonzeros among parts.

*Fine-grain hypergraph partitioning (**FHP**).* This is the partitioning strategy employed by HyperTensor. 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 $i = 1,\dots,m$ and $j = 1,\dots,n$. The resulting hypergraph is typically very large and is costly to partition, and unlike checkerboard variants the footprints of processes are not restricted to a row/column stride.

4.3.2 Partitioning $\mathbf{W}$ *and* $\mathbf{H}$. Once $\mathbf{A}$ 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 $\mathbf{W}$ and $\mathbf{H}$ 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 Section 4.2. 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 UpdateW and UpdateH 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 involve 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 $28M \times 1.6M$ and $17M \times 2.5M$, and have $112M$ and $72M$ nonzero elements, respectively. These matrices are obtained by pruning the third and the fourth dimensions of 4-dimensional Delicious and Flickr tensors available from [30]. 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.

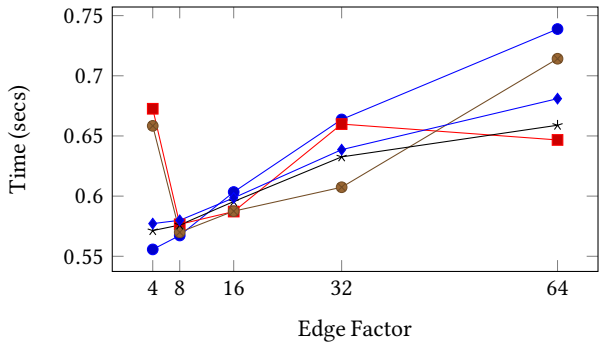
5.1.2 Implementation Platform. We conducted our experiments on two different parallel computing platforms. The first platform is the "Rhea" cluster at the Oak Ridge Leadership Computing Facility (OLCF), which is a commodity-type Linux cluster with a total of 512 nodes and a 4X FDR Infiniband interconnect. 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 [27]. We use BLAS and LAPACK for dense matrix operations by linking Armadillo with Intel MKL, OpenBLAS [34], 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.

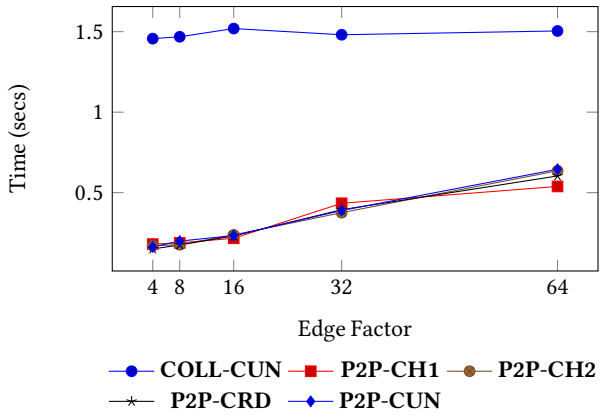
5.2 Effect of communication scheme

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}. As the edge factor increases, the sparse matrix becomes denser. We ran the baseline Mpi-Faun algorithm and the proposed algorithm with different partitioning schemes 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 partitioning and communication schemes using 4096 processors and a 64×64 processor grid. **COLL-CUN** is the implementation of Mpi-Faun and is the only instance that uses collective communication. In comparison, **P2P-CUN** also uses uniform partitioning but with point-to-point communication. **P2P-CRD** decides the checkerboard topology randomly and uses point-to-point scheme as well. **P2P-CH1** and **P2P-CH2** correspond to 1D-like and 2D checkerboard hypergraph partitioning models with point-to-point communication. Finally, **P2P-FHP** is the fine-graph hypergraph partitioning with point-to-point scheme, for which we



(a) Per-iteration computation cost for k=48 using 4096 processors



(b) Per-iteration communication cost for k=48 using 4096 processors

Figure 2: Per-iteration communication and computation costs of different partitioning strategies on Kronecker graphs using 4096 processors Rhea for $k = 48$, averaged over 30 iterations

only provide partition statistics for comparison and not the actual run time.

We notice in Fig. 2 that **COLL-CUN** always incurs significantly higher communication cost. As the edge factor increases and matrix becomes denser, the communication cost of **COLL-CUN** does not change as expected since it does not depend on the matrix sparsity. The communication cost of **P2P-CUN** progressively increases with the edge factor, but even with an edge factor of 64 **COLL-CUN** is about two times more costly. Therefore, we conclude that even though there is a converging trend between the cost of collective and point-to-point communication, it is a rather slow convergence, and the point-to-point scheme should be the method of choice unless the matrix gets very dense.

5.3 Effect of partitioning strategies

In Fig. 2b, our first observation is that all partitionings except **COLL-CUN** yield comparable results in terms of communication, while there is some variation in the computation times provided in Fig. 2a. **P2P-CUN** gives similar computation time to **P2P-CRD** since the matrix is randomly permuted. **P2P-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. **P2P-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.

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
P2P-CUN	1.30987	**1**	**1.00392**	**15366**	11876
P2P-CH1	1.21477	1.27734	1.05098	20800	11038
P2P-CH2	1.06479	1.30469	1.53725	23484	**10853**
P2P-CRD	1.38681	1.04688	1.05098	15475	11341
P2P-FHP	**1.00115**	1.05078	1.0549	47925	13802

We conclude that even though **P2P-CH2** provides a better model of 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. **P2P-CH1** yields a smaller hypergraph with reasonable partitioning cost and provides some benefits in terms of communication reduction. **P2P-CRD** 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 that justify these observations. We observe that **P2P-FHP** 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 [15].

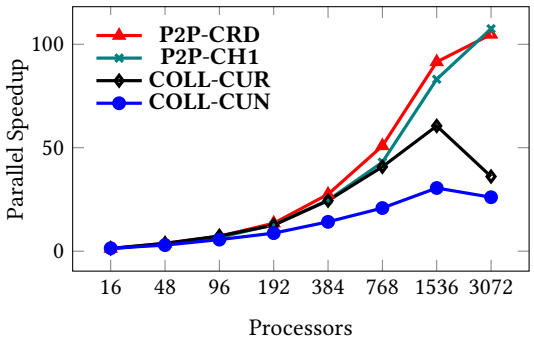
5.4 Strong scaling

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 as randomized and hypergraph checkerboard (1D-like) partitioning.

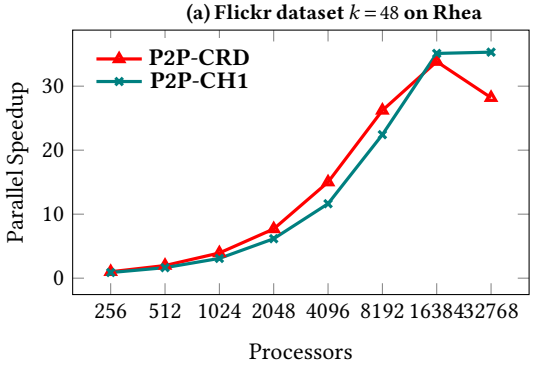
In this experiment, we considered the following algorithms and partitionings:

- **COLL-CUN**: Mpi-Faun algorithm [12, 13] with uniform partitioning (**COLL-CUN**) where each process holds an input matrix of size $m/P_r \times n/P_c$.
- **COLL-CUR**: The partitioning strategy in **COLL-CUN** 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 **COLL−CUR**.
- **P2P-CH1**: Dist-SpNMF (Algorithm 2) with 1D-like checkerboard hypergraph partitioning explained in Section 4.3.1.
- **P2P-CRD**: Dist-SpNMF (Algorithm 2) with randomized checkerboard partitioning 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 communication scheme

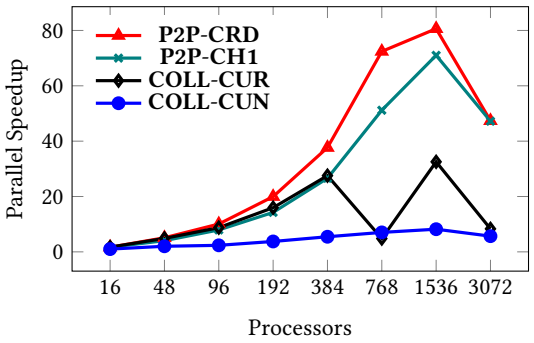


(a) Flickr dataset $k = 48$ on Rhea

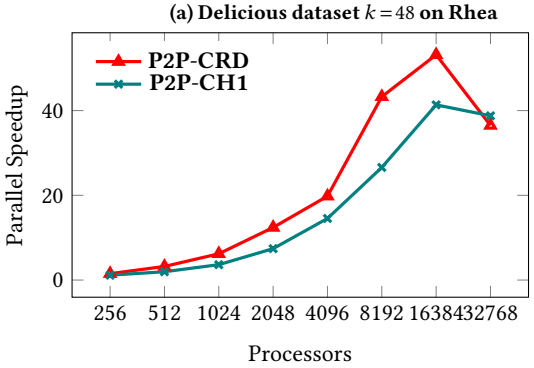


(b) Flickr dataset $k = 48$ on Bluegene/Q

Figure 3: Strong scaling on Flickr dataset



(a) Delicious dataset $k = 48$ on Rhea



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

Figure 4: Strong scaling on Delicious dataset

for both instances. We also realize that **COLL-CUR** significantly improves the runtime with respect to **COLL-CUN**, meaning that **COLL-CUN** indeed causes load imbalance in partitioning nonzeros of **A**. At 3072 processes, both **COLL-CUR** and **COLL-CUN** lose scalability and slow down, whereas **P2P-CH1** and **P2P-CRD** 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 **COLL-CUR** and **COLL-CUN** scale even worse in this case. Our algorithm also loses scalability after 1536 processes, and similarly to the previous case **P2P-CH1** starts slower than **P2P-CRD** 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 **P2P-CH1** manages to slightly improve the runtime using 32768 cores on Flickr, while all other slowing down using 32768 ranks. Again, **P2P-CH1** is slower than **P2P-CRD** using lower number of processors as the communication cost is negligible in these instances, and **P2P-CH1** introduces worse load balance than **P2P-CRD**. However, using 32768 processors **P2P-CH1** manages to outrun **P2P-CRD** by incurring less communication.

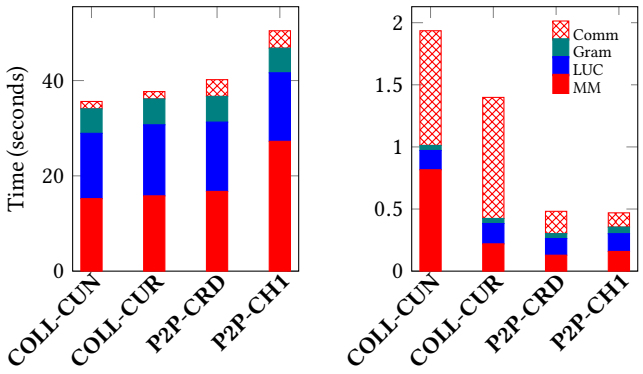
5.5 Time Breakdown Per Iteration

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 Update computation to compute the final value of the factor matrix using Gram and MM (time taken by UpdateW and UpdateH functions).
- **Comm**: Total expand and fold communication time for **P2P–CRD** and **P2P–CH1**, and the total time spent on All-Gather and Reduce-Scatter steps for **COLL–CUN** and **COLL–CUR**.

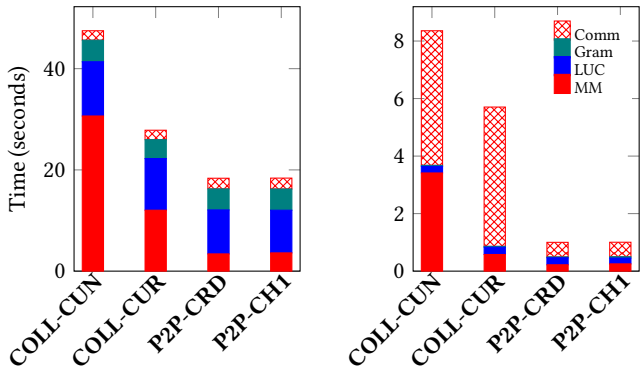
In our results, we do not distinguish the costs of these tasks for **W** and **H** separately; we instead report their sum.

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 by comparing the computational



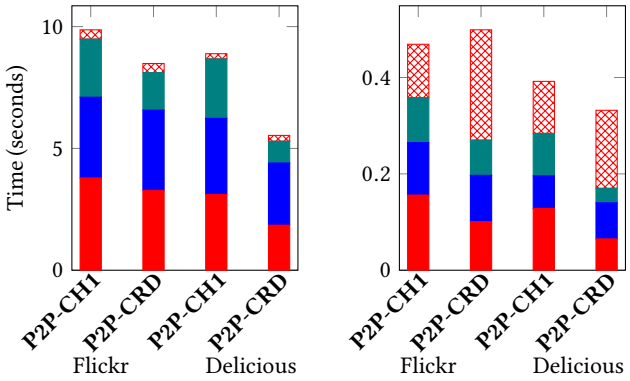
(a) Rhea $p = 16$ for $k = 48$ **(b) Rhea $p = 3072$ for $k = 48$**

Figure 5: Flickr dataset per-iteration time breakdown for k=48 on Rhea, averaged over 30 iterations



(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

and communication costs of different communication schemes and partitioning strategies.

Flickr on Rhea: We observe in Fig. 5 that in a one-node configuration with $p = 16$, the **COLL-CUN** and **COLL-CUR** performs similar to **P2P-CRD** and **P2P-CH1** 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 **P2P-CRD** and **P2P-CH1** stays reasonably low, whereas in the case of **COLL-CUN** and **COLL-CUR**, we clearly observe that the communication cost dominates the execution time. Randomization offers load balance to **COLL-CUR** which gives it a slight edge over **COLL-CUN**, yet both instances suffer from the high communication cost associated with the collective communication strategy, which explains the drop in the scalability results.

Delicious on Rhea: In Fig. 6, we see that **P2P-CRD** and **P2P-CH1** perform better than **COLL-CUN** even in single node configuration. Fig. 6 shows that **COLL-CUN** takes twice more than **P2P-CRD** and **P2P-CH1** 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, **P2P-CRD** and **P2P-CH1** perform significantly better than **COLL-CUN** and **COLL-CUR**, 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 that using 512 processors, communication cost is negligible, and **P2P-CRD** beats **P2P-CH1** thanks to better load balance. Using 16384 processors, however, on Flickr matrix **P2P-CH1** gets faster than **P2P-CRD** due to significant reduction in the communication volume. On Delicious matrix, **P2P-CH1** 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 the 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 32768 cores 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 to reduce the communication cost and employs effective partitioning to further enhance parallel scalability. Our immediate next steps for extending our work involve adding shared memory parallelism to obtain further speedup.

7 ACKNOWLEDGEMENTS

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This project was partially funded by the Laboratory Director's Research and Development fund. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy, and the resources of GENCI (Turing supercomputer) under the IDRIS Grant (i2016067501) .

Partial funding for this work was provided by National Science Foundation (NSF) grant ACI-1642385. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the USDOE or NSF.

REFERENCES

[1] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Ucar. 2008. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing* 68 (2008), 609–625. Issue 5. https://doi.org/10.1016/j.jpdc.2007.09.006

[2] Dimitri P Bertsekas. 1999. Nonlinear programming. (1999). https://doi.org/10.1057/palgrave.jors.2600425

[3] U. V. Catalyurek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (Jul 1999), 673–693. https://doi.org/10.1109/71.780863

[4] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446. https://doi.org/10.1137/1.9781611972740.43

[5] Andrzej Cichocki, Rafal Zdunek, Anh Huy Phan, and Shun-ichi Amari. 2009. *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation.* Wiley. https://doi.org/10.1002/9780470747278

[6] Christos Faloutsos, Alex Beutel, Eric P. Xing, Evangelos E. Papalexakis, Abhimanu Kumar, and Partha Pratim Talukdar. 2014. Flexi-FaCT: Scalable Flexible Factorization of Coupled Tensors on Hadoop. In *Proceedings of the SDM*. 109–117. https://doi.org/10.1137/1.9781611973440.13

[7] Alex Gittens, Aditya Devarakonda, Evan Racah, Michael F. Ringenburg, Lisa Gerhardt, Jey Kottalam, Jialin Liu, Kristyn J. Maschhoff, Shane Canon, Jatin Chhugani, Pramod Sharma, Jiyan Yang, James Demmel, Jim Harrell, Venkat Krishnamurthy, Michael W. Mahoney, and Prabhat. 2016. Matrix Factorization at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *CoRR* abs/1607.01335 (2016). http://arxiv.org/abs/1607.01335

[8] Edward F Gonzalez and Yin Zhang. 2005. *Accelerating the Lee-Seung algorithm for non-negative matrix factorization.* Technical Report. 1–13 pages. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.3585&rep=rep1&type=pdf

[9] David Grove, Josh Milthorpe, and Olivier Tardieu. 2014. Supporting Array Programming in X10. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. Article 38, 6 pages. https://doi.org/10.1145/2627373.2627380

[10] N. Guan, D. Tao, Z. Luo, and B. Yuan. 2012. NeNMF: An Optimal Gradient Method for Nonnegative Matrix Factorization. *IEEE Transactions on Signal Processing* 60, 6 (June 2012), 2882–2898. https://doi.org/10.1109/TSP.2012.2190406

[11] Patrik O Hoyer. 2004. Non-negative matrix factorization with sparseness constraints. *JMLR* 5 (2004), 1457–1469. www.jmlr.org/papers/volume5/hoyer04a/hoyer04a.pdf

[12] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. 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)*. ACM, New York, NY, USA, 9:1–9:11. https://doi.org/10.1145/2851141.2851152

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

[14] R. Kannan, A. V. Ievlev, N. Laanait, M. A. Ziatdinov, R. K. Vasudevan, S. Jesse, and S. V. Kalinin. 2018. Deep data analysis via physically constrained linear unmixing: universal framework, domain examples, and a community-wide platform. *Advanced Structural and Chemical Imaging* 4, 1 (30 Apr 2018), 6. https://doi.org/10.1186/s40679-018-0055-8

[15] Oguz Kaya and Bora Uçar. 2015. Scalable Sparse Tensor Decompositions in Distributed Memory Systems. In *Proceedings of SC*. ACM, Article 77, 11 pages. https://doi.org/10.1145/2807591.2807624

[16] Hyunsoo Kim and Haesun Park. 2007. Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis. *Bioinformatics* 23, 12 (2007), 1495–1502. http://dx.doi.org/10.1093/bioinformatics/btm134

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

[18] Jingu Kim and Haesun Park. 2008. *Sparse nonnegative matrix factorization for clustering.* Technical Report. Georgia Institute of Technology. https://smartech.gatech.edu/bitstream/handle/1853/20058/GT-CSE-08-01.pdf?sequence=1&isAllowed=y

[19] Da Kuang, Chris Ding, and Haesun Park. 2012. Symmetric nonnegative matrix factorization for graph clustering. In *Proceedings of SDM*. 106–117. http://epubs.siam.org/doi/pdf/10.1137/1.9781611972825.10

[20] Da Kuang, Sangwoon Yun, and Haesun Park. 2013. SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering. *Journal of Global Optimization* (2013), 1–30. http://dx.doi.org/10.1007/s10898-014-0247-2

[21] Ruiqi Liao, Yifan Zhang, Jihong Guan, and Shuigeng Zhou. 2014. CloudNMF: A MapReduce Implementation of Nonnegative Matrix Factorization for Large-scale Biological Datasets. *Genomics, proteomics & bioinformatics* 12, 1 (2014), 48–51. http://dx.doi.org/10.1016/j.gpb.2013.06.001

[22] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. 2010. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce. In *Proceedings of the WWW*. ACM, 681–690. http://dx.doi.org/10.1145/1772690.1772760

[23] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. http://dx.doi.org/10.14778/2212351.2212354

[24] Edgardo Mejía-Roa, Daniel Tabas-Madrid, Javier Setoain, Carlos García, Francisco Tirado, and Alberto Pascual-Montano. 2015. NMF-mGPU: non-negative matrix factorization on multi-GPU systems. *BMC bioinformatics* 16, 1 (2015), 43. http://dx.doi.org/10.1186/s12859-015-0485-4

[25] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. arXiv:1505.06807 http://arxiv.org/abs/1505.06807

[26] V Paul Pauca, Farial Shahnaz, Michael W Berry, and Robert J Plemmons. 2004. Text mining using nonnegative matrix factorizations. In *Proceedings of SDM*. https://doi.org/10.1137/1.9781611972740.45

[27] Conrad Sanderson. 2010. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report. NICTA. http://arma.sourceforge.net/armadillo_nicta_2010.pdf

[28] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. 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, 979–990. https://doi.org/10.1145/2588555.2610518

[29] D. Seung and L. Lee. 2001. Algorithms for non-negative matrix factorization. *NIPS* 13 (2001), 556–562. http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

[30] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. http://frostt.io/

[31] Sreenivas R. Sukumar, Ramakrishnan Kannan, Seung-Hwan Lim, and Michael A. Matheson. 2016. 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*. 1026–1031. https://doi.org/10.1109/BigData.2016.7840703

[32] Sreenivas R. Sukumar, Michael A. Matheson, Ramakrishnan Kannan, and Seung-Hwan Lim. 2016. Mini-apps for high performance data analysis. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. 1483–1492. https://doi.org/10.1109/BigData.2016.7840756

[33] D. L. Sun and C. Févotte. 2014. 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)*. 6201–6205. https://doi.org/10.1109/ICASSP.2014.6854796

[34] Zhang Xianyi. Last Accessed 03-Dec-2015. OpenBLAS. http://www.openblas.net

[35] Jiangtao Yin, Lixin Gao, and Zhongfei(Mark) Zhang. 2014. Scalable Nonnegative Matrix Factorization with Block-wise Updates. In *Machine Learning and Knowledge Discovery in Databases (LNCS)*, Vol. 8726. 337–352. http://dx.doi.org/10.1007/978-3-662-44845-8_22

[36] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. 2014. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *Proceedings of the VLDB Endowment* 7, 11 (2014), 975–986. https://doi.org/10.14778/2732967.2732973