2016 IEEE International Parallel and Distributed Processing Symposium

A Medium-Grained Algorithm for Distributed Sparse Tensor Factorization

Shaden Smith, George Karypis
Department of Computer Science and Engineering, University of Minnesota
{*shaden, karypis*}*@cs.umn.edu*

***Abstract*—Modeling multi-way data can be accomplished using tensors, which are data structures indexed along three or more dimensions. Tensors are increasingly used to analyze extremely large and sparse multi-way datasets in life sciences, engineering, and business. The canonical polyadic decomposition (CPD) is a popular tensor factorization for discovering latent features and is most commonly found via the method of alternating least squares (CPD-ALS). The computational time and memory required to compute CPD limits the size and dimensionality of the tensors that can be solved on a typical workstation, making distributed solution approaches the only viable option. Most methods for distributed-memory systems have focused on distributing the tensor in a coarse-grained, one-dimensional fashion that prohibitively requires the dense matrix factors to be fully replicated on each node. Recent work overcomes this limitation by using a fine-grained decomposition of the tensor nonzeros, at the cost of computationally expensive hypergraph partitioning. To that effect, we present a medium-grained decomposition that avoids complete factor replication and communication, while eliminating the need for expensive pre-processing steps. We use a hybrid MPI+OpenMP implementation that exploits multi-core architectures with a low memory footprint. We theoretically analyze the scalability of the coarse-, medium-, and fine-grained decompositions and experimentally compare them across a variety of datasets. Experiments show that the medium-grained decomposition reduces communication volume by 36-90% compared to the coarse-grained decomposition, is 41-76x faster than a state-of-the-art MPI code, and is 1.5-5.0x faster than the fine-grained decomposition with 1024 cores.**

***Keywords*-Sparse tensor, distributed, PARAFAC, CPD, parallel, medium-grained**

I. Introduction

Multi-way data arises in many of today's applications. A natural representation of this data is via a *tensor*, which is the extension of a matrix to three or more dimensions (called *modes*). For example, we can model product reviews as *user-item-word* triplets [1], the Never-Ending Language Learning (NELL) knowledge database as *noun-verb-noun* triplets [2], or electronic health records as *patient-procedure-diagnosis* triplets [3]. These tensors have very long modes and are very sparse (e.g., NELL has a density of 9×10^{-13}).

The recent popularity of tensors has led to an increased use of tensor factorization, a powerful tool for discovering the latent features in multi-way data. The most popular factorization is the canonical polyadic decomposition (CPD), a rank decomposition that can be seen as a higher-dimensional generalization of the singular value decomposition. The CPD represents the tensor via a matrix of latent features for each mode. We refer to these matrices as *factors*. The columns of the factors often represent some real-world interpretation of the dataset, such as film genre, word category, or phenotype. The CPD has been used with great success to perform tasks such as identifying word synonyms [4], performing webpage queries [5], and generating a list of recommendations [6].

Computing the CPD is a non-convex optimization problem. The most common method is using the method of alternating least squares (CPD-ALS), which solves the non-convex problem by turning each iteration into a sequence of convex least squares solutions. The computational time and memory required to compute the CPD limits the size and dimensionality of the tensors that can be solved on a typical workstation, making distributed solution approaches the only viable option.

Two systems for distributed tensor factorization are DFacTo [7] and SALS [8]. They partition the input tensor in a coarse-grained fashion, and require the dense matrix factors to be present on each node. A drawback to both methods is that they are not memory scalable because the matrix factors can consume more memory than the original sparse tensor and each node must communicate those factors in their entirety, each iteration. HyperTensor [9] is a recent work that overcomes this limitation by using a fine-grained decomposition of the tensor's nonzeros. The fine-grained decomposition reduces communication volume by using hypergraph partitioning, which often takes significantly more time than actually computing the factorization.

To address these limitations, we present a distributed CPD-ALS algorithm that is scalable in terms of computation and memory. Scalability is achieved by performing an m-dimensional decomposition of the tensor, where m is the number of modes, and one-dimensional decompositions of the factor matrices. Our distributed-memory CPD-ALS implementation, called DMS, is has two levels of parallelism: MPI provides node-level parallelism and OpenMP provides multi-core parallelism within each computing node. Our contributions are:

1) A CPD-ALS algorithm for distributed-memory systems that uses an m-dimensional decomposition of the tensor and one-dimensional decompositions of the factors to achieve computational and memory scalability.
2) A theoretical analysis of the medium-grained decomposition, which shows that it reduces the communi-

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.113

cation overhead from $O(IF)$ to $O\left(IF/\sqrt[n]{p}\right)$, where IF is the size of the output and p is the number of cores.

3) An extensive set of experiments across various datasets on up to 512 cores. DMS reduces communication volume by 36% to 90%, is 20× to 60× faster than DFACTO, and is 1.7× to 5.0× faster than our own fine-grained implementation.

The rest of this paper is organized as follows. Section II introduces notation and provides a background on the CPD and ALS. Section III highlights existing approaches for distributed tensor factorization. We describe our tensor decomposition and distributed CPD-ALS algorithm in Section IV, and detail efficient algorithms for finding a decomposition in Section V. Section VI details our experimental setup and provide a discussion of the results. Finally, Section VII provides some concluding remarks.

II. TENSOR BACKGROUND

A. Tensor Notation

We denote vectors using bold lowercase letters ($\boldsymbol{\lambda}$), matrices using bold capital letters ($\mathbf{A}$), and tensors using bold capital calligraphic letters ($\boldsymbol{\mathcal{X}}$). The element in coordinate (i, j, k) of $\boldsymbol{\mathcal{X}}$ is $\boldsymbol{\mathcal{X}}(i, j, k)$. Unless specified, the sparse tensor $\boldsymbol{\mathcal{X}}$ is of dimension $I \times J \times K$ and has $\text{nnz}(\boldsymbol{\mathcal{X}})$ nonzero elements. A colon in the place of an index represents all members of that mode. For example, $\mathbf{A}(:, f)$ is column f of the matrix $\mathbf{A}$. *Fibers* are the generalization of matrix rows and columns and are the result of holding two indices constant. A *slice* of a tensor is the result of holding one index constant and the result is a matrix.

A tensor can be unfolded, or *matricized*, into a matrix along any of its modes. In the mode-n matricization, the mode-n fibers form the columns of the resulting matrix. The mode-n unfolding of $\boldsymbol{\mathcal{X}}$ is denoted as $\mathbf{X}_{(n)}$. If $\boldsymbol{\mathcal{X}}$ is of dimension $I \times J \times K$, then $\mathbf{X}_{(1)}$ is of dimension $I \times JK$.

Two essential matrix operations used in the CPD are the *Hadamard product* and the *Khatri-Rao product*. The Hadamard product, denoted $\mathbf{A} * \mathbf{B}$, is the element-wise multiplication of $\mathbf{A}$ and $\mathbf{B}$. The Khatri-Rao product, denoted $\mathbf{A} \odot \mathbf{B}$, is the column-wise Kronecker product. If $\mathbf{A}$ is $I \times J$ and $\mathbf{B}$ is $M \times J$, then $\mathbf{A} \odot \mathbf{B}$ is $IM \times J$.

B. Canonical Polyadic Decomposition

The CPD is a generalization of the singular value decomposition (SVD) to tensors. In the SVD, a matrix $\mathbf{M}$ is factored into the summation of F rank-one matrices, where F can either be the rank of $\mathbf{M}$ or some smaller integer if a low-rank approximation is desired. CPD extends this concept to factor a tensor into the summation of F rank-one tensors. We are almost always interested in $F \ll \max\{I, J, K\}$ for sparse tensors. In this work we treat F as a small constant on the order of 10 or 100. A rank-F CPD produces factors $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are typically dense regardless of the sparsity of $\boldsymbol{\mathcal{X}}$. Unlike the SVD, the CPD does not require orthogonality in the columns of the factors. We output the factors with normalized columns and $\boldsymbol{\lambda} \in \mathbb{R}^F$, a vector for weighting the rank-one components. Using this form we can reconstruct $\boldsymbol{\mathcal{X}}$ via

$$\boldsymbol{\mathcal{X}}(i, j, k) \approx \sum_{f=1}^{F} \boldsymbol{\lambda}(f)\mathbf{A}(i, f)\mathbf{B}(j, f)\mathbf{C}(k, f).$$

Besides CPD there are other ways to compute factorizations of tensors such as the Tucker Decomposition [10]. However, the work in this paper focuses only on CPD and any reference to tensor factorization will indicate a CPD tensor factorization.

Algorithm 1 CPD-ALS

1: **while** not converged **do**
2: $\quad\mathbf{A}^\intercal = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}\left(\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})\right)^\intercal$
3: $\quad$Normalize columns of $\mathbf{A}$
4: $\quad\mathbf{B}^\intercal = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{A}^\intercal\mathbf{A})^{-1}\left(\mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})\right)^\intercal$
5: $\quad$Normalize columns of $\mathbf{B}$
6: $\quad\mathbf{C}^\intercal = (\mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A})^{-1}\left(\mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})\right)^\intercal$
7: $\quad$Normalize columns of $\mathbf{C}$ and store in $\boldsymbol{\lambda}$
8: **end while**

C. CPD with Alternating Least Squares

CPD-ALS is the most common algorithm for computing the CPD. The non-convex problem is transformed into a convex one for each factor and iterate until convergence. During each iteration, $\mathbf{B}$ and $\mathbf{C}$ are fixed and we solve the unconstrained least squares optimization problem

$$\underset{\mathbf{A}}{\text{minimize}} \qquad \frac{1}{2}||\mathbf{X}_{(1)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\intercal||_F^2$$

with solution

$$\mathbf{A}^\intercal = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}\left(\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})\right)^\intercal$$

We first find $\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$, followed by the Gram matrix $\mathbf{M} = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})$. $\mathbf{M}$ is an $F \times F$ positive semi-definite matrix and so we use its Cholesky factorization instead of explicitly computing its inverse. $\mathbf{B}$ and $\mathbf{C}$ are then solved for similarly. The factors are normalized each iteration and $\boldsymbol{\lambda}$ stores the F column norms. The full CPD-ALS steps are shown in Algorithm 1.

We denote $\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ as the *matricized tensor times Khatri-Rao product* (MTTKRP). Explicitly forming $\mathbf{C} \odot \mathbf{B}$ and performing the matrix multiplication requires orders of magnitude more memory than the original sparse tensor. Instead, we exploit the block structure of the Khatri-Rao product to perform the multiplication *in place*. The fastest MTTKRP algorithms can execute an MTTKRP operation in $O\left(F \cdot \text{nnz}(\boldsymbol{\mathcal{X}})\right)$ floating-point operations (FLOPs), with

a leading constant dependent on the sparsity pattern of the tensor [7], [11], [12]. Entry $\hat{\mathbf{A}}(i,f)$ is given by

$$\hat{\mathbf{A}}(i,f) = \sum_{\boldsymbol{\mathcal{X}}(i,:,:)} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{B}(j,f)\mathbf{C}(k,f). \quad (1)$$

Equation (1) shows us two important properties of the MTTKRP operation. First, nonzeros in slice $\boldsymbol{\mathcal{X}}(i,:,:)$ will only contribute to row $\hat{\mathbf{A}}(i,:)$. Second, the j and k indices in slice $\boldsymbol{\mathcal{X}}(i,:,:)$ determine which rows of $\mathbf{B}$ and $\mathbf{C}$ must be accessed during the multiplication.

CPD-ALS iterates until convergence. The residual of a tensor $\boldsymbol{\mathcal{X}}$ and its CPD approximation $\boldsymbol{\mathcal{Z}}$ is

$$\sqrt{\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}} \rangle + \langle \boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Z}} \rangle - 2\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle}.$$

$\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}} \rangle = ||\boldsymbol{\mathcal{X}}||_F^2$ is a direct extension of the matrix Frobenius norm, i.e., the sum-of-squares of all nonzero elements. $\boldsymbol{\mathcal{X}}$ is also a constant input and thus its norm can be pre-computed. The norm of a factored tensor is

$$||\boldsymbol{\mathcal{Z}}||_F^2 = \boldsymbol{\lambda}^\intercal \left(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A}\right)\boldsymbol{\lambda}.$$

Fortunately, each $\mathbf{A}^\intercal\mathbf{A}$ product is computed during the CPD-ALS iteration and the results can be cached and reused in just $O(F^2)$ space. The complexity of computing the residual is bounded by the inner product $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle$ which is given by

$$\sum_{f=1}^{F} \boldsymbol{\lambda}(f) \left(\sum_{\text{nnz}(\boldsymbol{\mathcal{X}})} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \right). \quad (2)$$

The cost of Equation (2) is $4F \cdot \text{nnz}(\boldsymbol{\mathcal{X}})$ FLOPs, which is more expensive than an entire MTTKRP operation. In Section IV-B6 we present a method of reusing MTTKRP operation results to reduce the cost to $2FI$.

All of the above discussion can be generalized to tensors with more than three modes. For more information on tensors and their factorizations, we direct the reader to the excellent survey by Kolda and Bader [13].

III. Related Work

Distributed CPD algorithms such as DFacTo [7] and SALS [8] use *coarse-grained* decompositions in which independent one-dimensional (1D) decompositions are used for each tensor mode. Processes own a set of contiguous slices for each mode and are responsible for the corresponding factor rows. Figure 1 is an illustration of this decomposition scheme. An advantage of this scheme is the simplicity of performing MTTKRP operations. Each process owns all of the nonzeros that contribute to its owned output and thus the only communication required is exchanging updated factor rows after each iteration. Independent 1D decompositions can be interpreted as a task decomposition on the problem output, often called the *owner-computes rule*.

A limitation of these coarse-grained methods is that by owning slices in each mode of the tensor, processes own nonzeros that can span the complete modes of $\boldsymbol{\mathcal{X}}$. As a result, from Equation (1) we can see that processes will require access to the factors in their entirety over the course of the MTTKRP operations during an ALS iteration. The memory footprint of all factors can rival that of the entire tensor when the input is very sparse. Thus, memory consumption is not scalable and since updated factors must be communicated, communication is also not scalable.

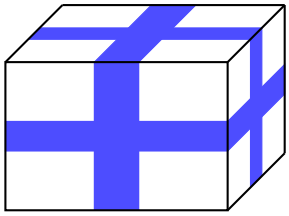


Figure 1: A coarse-grained decomposition of $\boldsymbol{\mathcal{X}}$. Slices owned by a single process are shaded.

Adding constraints such as non-negativity or sparsity in the latent factors is also an interest to the tensor community. A distributed non-negative CPD algorithm for dense tensors was introduced in [14]. A coarse-grained decomposition was used on the tensor and factors. A generalized framework for constrained CPD that uses the Alternating Direction Method of Multipliers (ADMM) was presented in [15]. Parallelism is extracted by performing a 2D decomposition on the matricized tensor and a row distribution of the factors. Neither of these two methods for parallel constrained tensor factorization were explicitly designed for sparse tensors and thus the storage and communication of full factors is not considered a limitation.

Recently, a new CPD-ALS algorithm named HyperTensor was presented in [9]. HyperTensor uses a *fine-grained* decomposition over $\boldsymbol{\mathcal{X}}$ in which nonzeros are individually assigned to processes. Several methods of computing such a decomposition are presented, with the most successful relying on hypergraph partitioning. HyperTensor maps $\boldsymbol{\mathcal{X}}$ to a hypergraph with $\text{nnz}(\boldsymbol{\mathcal{X}})$ vertices and $I+J+K$ hyperedges. The vertex representing nonzero $\boldsymbol{\mathcal{X}}(i,j,k)$ is connected to hyperedges i, j, and k. The experiments presented in [9] show that a balanced partitioning of the hypergraph leads to a load-balanced computation with low communication volume.

SPLATT [11] is a software toolkit for parallel sparse tensor factorization on shared-memory systems. It uses a compressed, fiber-centric data structure for the tensor called *compressed sparse fiber* (CSF). The CSF data structure allows SPLATT to perform operation-efficient, multi-threaded MTTKRP operations using a single tensor representation [16]. The MTTKRP algorithm used in SPLATT computes whole rows of $\hat{\mathbf{A}}$ at a time by processing all of

the fibers in a single slice:

$$\hat{\mathbf{A}}(i,:) \leftarrow \sum_{j=1}^{J} \left(\mathbf{B}(j,:) * \sum_{k=1}^{K} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{C}(k,:) \right).$$

Factoring out the contributions of $\mathbf{B}$ allows SPLATT to use fewer FLOPs than other algorithms that operate on individual nonzeros.

IV. Medium-Grained CPD-ALS

In order to address the high memory and communication requirements of the coarse-grained decomposition while at the same time eliminate the need to perform the expensive pre-processing step associated with hypergraph partitioning, we developed an approach that uses a medium-grained decomposition. Like coarse- and fine-grained methods, medium-grained have roots in the sparse matrix community [17]–[19]. The medium-grained decomposition uses an m-mode decomposition over the tensor and related 1D decompositions on the factor matrices. The medium-grained CPD-ALS algorithm is parallelized at the node-level using a message passing model and exploits multi-core architectures as well with thread-level parallelism on each node.

In order to simplify the presentation, this section considers only three mode tensors and the generalization of the algorithms to higher-order tensors is discussed in Section IV-C.

A. Data Distribution Scheme

Assume that there are $p = q \times r \times s$ processing elements available. We form a 3D decomposition of $\boldsymbol{\mathcal{X}}$ by partitioning its three modes into q, r, and s chunks, respectively. The intersections of these partitions form a total of p partitions arranged in a $q \times r \times s$ grid. We denote $\boldsymbol{\mathcal{X}}_{(x,y,z)}$ as the partition of $\boldsymbol{\mathcal{X}}$ with coordinate (x, y, z), and $p_{(x,y,z)}$ as the process that owns $\boldsymbol{\mathcal{X}}_{(x,y,z)}$. We refer to a group processes which share a coordinate as a *layer*. For example, $p_{(i,:,:)}$ is a layer of $r \times s$ processes along the first mode and $p_{(:,j,:)}$ is layer of $q \times s$ processes along the second mode.

In our implementation, each process stores its subtensor in the CSF data structure. This allows us to use the operation-efficient MTTKRP algorithm included in SPLATT to extract parallelism on shared-memory architectures.

We use the 3D decomposition of $\boldsymbol{\mathcal{X}}$ to induce partitionings of the rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. The rows of $\mathbf{A}$ are divided into chunks $\mathbf{A}_1, \ldots, \mathbf{A}_q$ which have boundaries aligned with the q partitions of the first mode of $\boldsymbol{\mathcal{X}}$. The rows in $\mathbf{A}_i$ are collectively owned by all processes in layer $p_{(i,:,:)}$. The rows of $\mathbf{B}$ and $\mathbf{C}$ are similarly divided into $\mathbf{B}_1, \ldots, \mathbf{B}_r$ and $\mathbf{C}_1, \ldots, \mathbf{C}_s$, respectively. This decomposition is illustrated in Figure 2a.

We further partition the rows of each chunk of $\mathbf{A}$ into $r \times s$ groups such that each process in layer $p_{(i,:,:)}$ owns a subset of the rows of $\mathbf{A}_i$. We note that the partitioning need not assign a contiguous set of rows to a process and a process is not required to be assigned any rows. The output of the MTTKRP operation, $\hat{\mathbf{A}}$, has the same distribution as $\mathbf{A}$. Process p_i owns the same rows of $\hat{\mathbf{A}}_i$ as it does $\mathbf{A}_i$. The process is repeated for $\mathbf{B}$ and $\mathbf{C}$ similarly. We relabel the slices of $\boldsymbol{\mathcal{X}}$ in order to make the rows owned by each process contiguous. This is illustrated in Figure 2c.

In subsequent discussions we will refer to process-level partitions of $\mathbf{A}$ in two ways: $\mathbf{A}_{p_i}$ refers to the chunk of $\mathbf{A}$ owned by process p_i, and $\mathbf{A}_{(x,y,z)}$ refers to the chunk of $\mathbf{A}$ owned by process with coordinate (x, y, z). The coordinate form will simplify discussion during the MTTKRP operation that relies on the 3D decomposition.

B. Distributed CPD-ALS

We will now detail each step of a CPD-ALS iteration using our 3D decomposition. For brevity we only discuss the computations used for the first mode. The other tensor modes are computed identically.

1) Distributed MTTKRP Operations: Process $p_{(x,y,z)}$ performs an MTTKRP operation with $\boldsymbol{\mathcal{X}}_{(x,y,z)}$. Any nonzeros in $\boldsymbol{\mathcal{X}}_{(x,y,z)}$ whose mode-1 indices are non-local will produce partial products that must be sent to other processes in the layer $p_{(x,:,:)}$. Likewise, $p_{(x,y,z)}$ will receive partial products from any processes in layer $p_{(x,:,:)}$ which output to rows in $\hat{\mathbf{A}}_{(x,y,z)}$. The received partial products are then aggregated, resulting in the completed $\hat{\mathbf{A}}_{(x,y,z)}$.

2) Cholesky Factorization: $\mathbf{B}^\mathsf{T}\mathbf{B}$ and $\mathbf{C}^\mathsf{T}\mathbf{C}$ are $F \times F$ matrices that comfortably fit in the memory of each process. Assume $\mathbf{B}^\mathsf{T}\mathbf{B}$ and $\mathbf{C}^\mathsf{T}\mathbf{C}$ are already resident in each process' memory. All processes redundantly compute the Cholesky factorization of $\mathbf{M} = (\mathbf{C}^\mathsf{T}\mathbf{C} * \mathbf{B}^\mathsf{T}\mathbf{B})$ in $O(F^3)$ time, which is a negligible overhead for the low-rank problems that we are interested in. We perform the forward and backward substitutions in block form to exploit our row-wise distribution of $\hat{\mathbf{A}}$:

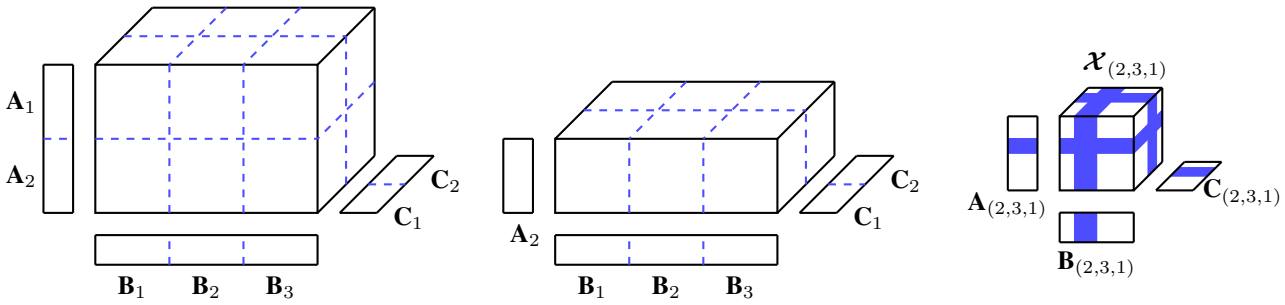
$$\mathbf{A}^\mathsf{T} = \mathbf{M}^{-1}\hat{\mathbf{A}}^\mathsf{T} = \begin{bmatrix} \mathbf{M}^{-1}\hat{\mathbf{A}}_{p_1}^\mathsf{T} & \mathbf{M}^{-1}\hat{\mathbf{A}}_{p_2}^\mathsf{T} & \ldots & \mathbf{M}^{-1}\hat{\mathbf{A}}_{p_p}^\mathsf{T} \end{bmatrix}$$

3) Column Normalization: After computing the new factor $\mathbf{A}$, we normalize its columns and store the norms in the $F \times 1$ vector $\boldsymbol{\lambda}$. Processes first compute the local column norms of $\mathbf{A}_{p_i}$ and collectively find the global $\boldsymbol{\lambda}$ with a parallel reduction. Finally, each process normalizes the columns of $\mathbf{A}_{p_i}$ with $\boldsymbol{\lambda}$.

Processes further parallelize the normalization process by using the 1D decomposition of the matrix rows and finding thread-local norms. The threads then use a reduction before the global $\boldsymbol{\lambda}$ is computed.

4) Forming the New Gram Matrix: Each process needs the updated $\mathbf{A}^\mathsf{T}\mathbf{A}$ factor in order to form $\mathbf{M}$ during the proceeding modes. We view the block matrix form of the computation to derive a distributed algorithm:

$$\mathbf{A}^\mathsf{T}\mathbf{A} = \begin{bmatrix} \mathbf{A}_{p_1}^\mathsf{T} & \mathbf{A}_{p_2}^\mathsf{T} & \ldots & \mathbf{A}_{p_p}^\mathsf{T} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{p_1} \\ \mathbf{A}_{p_2} \\ \ldots \\ \mathbf{A}_{p_p} \end{bmatrix} = \sum_{i=1}^{p} \mathbf{A}_{p_i}^\mathsf{T}\mathbf{A}_{p_i}.$$



(a) $\mathcal{X}$ is distributed over a 2×3×2 grid.

(b) Process layer $p_{(2,:,:)}$ collectively owns $\mathbf{A}_2$.

(c) Process p_7 owns $\mathcal{X}_{2,3,1}$ and the shaded factor rows. Shaded nonzeros do not result in communication.

Figure 2: A medium-grained decomposition for twelve processes.

Each process first computes its local $\mathbf{A}_{p_i}^\mathsf{T}\mathbf{A}_{p_i}$. The 1D decomposition on the rows of $\mathbf{A}_{p_i}$ is used again to extract thread-level parallelism. We then perform an All-to-All reduction to find the final matrix and distribute it among all processes.

5) Updating Non-Local Rows: Processes with non-local rows of $\mathbf{A}$ must receive updated values before the next MTTKRP operation. This communication is a dual of exchanging partial products during the distributed MTTKRP operation. Any processes that sent partial MTTKRP products to process p_i now receive the updated rows of $\mathbf{A}_{p_i}$.

6) Residual Computation: Convergence is tested at the end of every iteration. In Section II-C we showed that residual computation cost is bounded by $\langle \mathcal{X}, \mathcal{Z} \rangle$, which uses $4F \cdot \text{nnz}(\mathcal{X})$ FLOPs. We observe that contributions from $\mathbf{B}$ and $\mathbf{C}$ with $\mathcal{X}$ are already computed during the MTTKRP operation. Thus, we can cache $\hat{\mathbf{A}}$ and rewrite Equation (2) as

$$\mathbf{1}^\mathsf{T} \begin{bmatrix} \mathbf{A}_{p_1} * \hat{\mathbf{A}}_{p_1} \\ \mathbf{A}_{p_2} * \hat{\mathbf{A}}_{p_2} \\ \cdots \\ \mathbf{A}_{p_p} * \hat{\mathbf{A}}_{p_p} \end{bmatrix} \boldsymbol{\lambda} = \sum_{i=1}^{p} \mathbf{1}^\mathsf{T} \left(\hat{\mathbf{A}}_{p_i} * \mathbf{A}_{p_i} \right) \boldsymbol{\lambda}, \qquad (3)$$

where $\mathbf{1}$ is the vector of all ones. This reduces the computation to $2IF$ FLOPs.

Each process computes its own local $\mathbf{1}^\mathsf{T} \left(\hat{\mathbf{A}}_{p_i} * \mathbf{A}_{p_i} \right) \boldsymbol{\lambda}$. Thread-level parallelism is achieved via 1D row decompositions on $\hat{\mathbf{A}}_{p_i}$ and $\mathbf{A}_{p_i}$. Finally, we use a parallel reduction on each node's local result and form $\langle \mathcal{X}, \mathcal{Z} \rangle$.

C. Extensions to Higher Modes

Extending our distributed CPD-ALS algorithm to tensors with an arbitrary number of modes is straightforward. Suppose $\mathcal{X}$ is a tensor with m modes and we wish to compute factors $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(m)}$.

Operation-efficient MTTKRP algorithms for a general number of modes are found in [11], [16]. Adding partial products from neighbor processes remains the same, with the only consideration being that a *layer* is no longer a 2D group of processes, but a group of dimension $m-1$.

Residual computation again is easily extended. Generalized MTTKRP computes

$$\hat{\mathbf{A}}^{(1)}(i_1, f) = \sum \mathcal{X}(i_1, \ldots, i_m)\mathbf{A}^{(2)}(i_2, f) \ldots \mathbf{A}^{(n)}(i_m, f),$$

and so we can directly use Equation (3) to complete the residual calculation. Assuming $\hat{\mathbf{A}}^{(1)}$ can be cached, our algorithm does not increase in cost as more modes are added.

D. Complexity Analysis

The cost of CPD-ALS is bounded by MTTKRP and its associated communication. Coarse-, medium-, and fine-grained distributed algorithms distribute work such that each process does $O(F \cdot \text{nnz}(\mathcal{X})/p)$ work. They differ, however, in the overheads associated with communication. In this section, we discuss the communication costs present in coarse- and medium-grained decompositions for a single mode. We define the communication cost as the number of words of $\mathbf{A}$ and $\hat{\mathbf{A}}$ that must be communicated. The flexibility of fine-grained decompositions makes analysis difficult; both coarse- and medium-grained communication patterns are possible if the nonzeros are distributed appropriately. In our discussion we will use the personalized all-to-all collective communication. Derivation of its complexity can be found in [20].

Assume that $\mathcal{X}$ has m modes, is of dimension $I \times \ldots \times I$, p processes are arranged in a $\sqrt[m]{p} \times \ldots \times \sqrt[m]{p}$ grid, and that messages require $O(1)$ time to transfer per word. A medium-grained decomposition has two communication steps to consider: aggregating non-local rows during an MTTKRP operation and sending updated rows of $\mathbf{A}_{p_i}$ after an iteration.

In the worst case, every process has nonzeros in all $(I/\sqrt[m]{p})$ slices of the layer. A process must send (I/p) unique rows of $\hat{\mathbf{A}}$ to each of its neighbors in the layer. Using a personalized all-to-all collective, this communication is accomplished in time

$$\frac{IF}{p}\left(p^{\frac{m-1}{m}} - 1\right) = \left(\frac{IF}{\sqrt[m]{p}} - \frac{IF}{p}\right) = O\left(\frac{IF}{\sqrt[m]{p}}\right). \qquad (4)$$

The worst case of the update stage is sending (I/p) rows to each of the $p^{\frac{m-1}{m}}$ neighbors in the layer. This operation

is the dual of Equation (4) and has the same cost.

In comparison, a coarse-grained decomposition will send up to (I/p) rows to all p processes. The communication overhead is thus

$$\frac{IF}{p}(p-1) = O\left(IF\right). \tag{5}$$

No partial results from an MTTKRP operation need to be communicated, however, so Equation (5) is the only communication associated with a coarse-grained decomposition. Comparing Equations (4) and (5) shows that only the medium-grained decomposition can reduce communication costs by increasing parallelism. We experimentally evaluate this observation in Section VI-D.

V. Computing the Data Decomposition

Our discussion so far has provided an overview of our medium-grained data decomposition and a distributed algorithm for CPD-ALS. There are two forms of overhead that an ideal data decomposition will minimize: load imbalance and communication volume. Graph and hypergraph partitioners co-optimize these objectives, but can require significant preprocessing. We chose to optimize the objectives separately. We load balance the computation during the tensor decomposition because computational load is mostly a function of the number of nonzeros assigned to a process. Communication volume is optimized during the decomposition of the factor matrices because the assignment of rows directly impacts communication.

A. Finding a Balanced Tensor Decomposition

Our objective is to derive a load balanced $q\times r\times s$ decomposition of the modes of $\boldsymbol{\mathcal{X}}$. We begin by randomly permuting the each mode of the tensor. The purpose of the random permutation is to remove any ordering present from the data collection process that could result in load imbalance. Each mode is then partitioned independently.

The decomposition of the first mode into q parts is determined as follows: We greedily assign partition boundaries by adding consecutive slices until a partition has at least $\mathrm{nnz}(\boldsymbol{\mathcal{X}})/q$ nonzeros. We call $\mathrm{nnz}(\boldsymbol{\mathcal{X}})/q$ the *target* size of a partition because it will result in a load balanced partitioning of the mode. Slices can vary in density and adding a slice with many nonzeros can push a partition significantly over the target size. Thus, after identifying the slice which pushes a partition over the target size we compare it to the slice immediately before and choose whichever leads to better balance.

Each of the independent mode decompositions is an instance of the chains-on-chains partitioning problem, for which there are fast exact algorithms [21]. We found that in practice, using optimal partitionings led to higher load imbalance than greedily choosing sub-optimal partitionings. Since we ultimately work with the *intersection* of the 1D partitionings, having optimality in each dimension does not guarantee optimality in the final partitioning.

B. Partitioning the Factor Matrices

A process may have nonzeros whose indices correspond to factor rows which are not owned by the process itself. These non-local rows must be communicated. Thus, the partitioning of rows during the sub-division of $\mathbf{A}_i$ directly affects the number of partial results which are exchanged during the MTTKRP operation. Our objective is to minimize the total number of communicated rows, or the *communication volume*. We adapt a greedy method of assigning rows developed for two-dimensional sparse matrix-vector multiplication [19]. We again partition each mode independently.

The sub-division of $\mathbf{A}$ is determined as follows: The q chunks of of $\mathbf{A}$ are partitioned independently. For each row i_r in chunk $\mathbf{A}_i$, processes count the number of tensor partitions (and thus, processes) that contain a nonzero value in slice $\boldsymbol{\mathcal{X}}(i_r,:,:)$. Any row that is found in only a single partition is trivially assigned to the owner because it will not increase communication volume. Next, the master process in the layer $p_{(i,:,:)}$ coordinates the assignment of all remaining rows. At each step it selects the processes with the two smallest communication volumes, p_j and p_k, with p_j having the smaller volume. The master process sends a message to p_j instructing it to claim rows until its volume matches p_k. Processes first claim indices which are found in their local tensor and only claim non-local ones when options are exhausted. The assignment procedure sometimes reaches a situation in which all processes have equal volumes but not all rows have been assigned. To overcome this obstacle we instruct the next process to claim a $1/(r\times s)$ fraction of the remaining rows.

These steps are then performed on the second and third tensor modes to complete the decomposition.

C. Choosing the Shape of the Decomposition

Our decomposition does not require an equal number of processes along each mode. We select at runtime the number of processes that should be assigned to each mode. Most tensors will feature one or more modes that are significantly longer than the others. For example, the Netflix tensor described in Section VI has over $20\times$ more users than it does films. When choosing the dimensions for the decomposition, it is advantageous to assign more processes to the long modes than the short ones. The reasoning behind this decision is that short modes are likely to require storage and communication regardless of the decomposition and we should instead use more processes to further the decompose the modes which can benefit.

A constraint we impose when computing the decomposition is that the product of the dimensions must equal the number of processes, i.e., $q\times r\times s = p$. To achieve this, we

Algorithm 2 Deriving the decomposition shape

Input: dims, the dimensions of $\mathcal{X}$; m, the number of modes in $\mathcal{X}$; p, the number of processes.
Output: P, a vector storing the decomposition dimensions

$F \leftarrow$ the prime factors of p in non-increasing order
$P \leftarrow \mathbf{1}$, an m-dimensional vector of ones
▷ Find the optimal number of slices per process.
target $\leftarrow \left(\sum_{i=1}^{m} \text{dims}[i]\right)/p$
for all $f \in F$ **do** ▷ Assign a factor of p at a time
  distances $\leftarrow \mathbf{0}$, an m-dimensional vector of zeros
  ▷ Find the mode with the most work per process
  for $i \leftarrow 1$ to m **do**
    distances$[i] \leftarrow (\text{dims}[i]/P[i]) - \text{target}$
  end for
  furthest $\leftarrow \text{argmax}_i \text{distances}[i]$
  $P[\text{furthest}] \leftarrow P[\text{furthest}] \times f$ ▷ Give f processes
end for

break p into its prime factors and greedily assign them to modes. This process is detailed in Algorithm 2.

VI. Experimental Methodology & Results

A. Experimental Setup

We used SPLATT to implement three versions of distributed CPD-ALS. We refer to the collection of our implementations as DMS (distributed-memory SPLATT). The first version, DMS-CG, uses a coarse-grained decomposition and is a direct implementation of the algorithm used in SPLATT and adapted to distributed-memory systems. The second method uses a medium-grained decomposition described in Section IV and is denoted DMS-MG. Our final implementation is DMS-FG, which follows the fine-grained tensor decomposition used in the evaluation of HyperTensor [9]. All three algorithms use the same computational kernels and only differ in decomposition and the resulting communications. DMS-CG and DMS-MG are implemented with personalized all-to-all collective operations, while DMS-FG uses point-to-point communications.

Zoltan [22] with PHG was used for hypergraph partitioning with LB_APPROACH set to "PARTITION". All hypergraphs were partitioned offline using 512 cores. Partitioning required between 1400 seconds on Netflix and 6400 seconds on Delicious.

DMS is implemented in C with double-precision floating-point numbers and 64-bit integers. DMS uses MPI for distributed memory parallelism and OpenMP for shared-memory parallelism. All source code is available for download[1]. Source code was compiled with GCC 4.9.2 using optimization level three.

We compare against DFacTo, which to our knowledge is the fastest publicly available tensor factorization software.

[1] http://cs.umn.edu/~splatt/

Table I: Summary of datasets.

Dataset	I	J	K	nnz	storage (GiB)
Netflix	480K	18K	2K	100M	3.0
Delicious	532K	17M	3M	140M	4.2
NELL	3M	2M	25M	143M	4.3
Amazon	5M	18M	2M	1.7B	51.9
Random1	20M	20M	20M	1.0B	29.8
Random2	50M	5M	5M	1.0B	29.8

nnz is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively. **storage** is the amount of memory required to represent the tensor as $(i, j, k) = v$ tuples using 64-bit integers and 64-bit floating-point values.

DFacTo is implemented in C++ and uses MPI for distributed memory parallelism.

We used $F = 16$ for all experiments. Experiments were carried out on HP ProLiant BL280c G6 blade servers on a 40-gigabit InfiniBand interconnect. Each server had dual-socket, quad-core Xeon X5560 processors running at 2.8 GHz with 8MB last-level cache and 22 gigabytes of available memory.

B. Datasets

Table I is a summary of the datasets we used for evaluation. The Netflix dataset is taken from the Netflix Prize competition [23] and forms a *user-item-time* ratings tensor. NELL [2] is comprised of *noun-verb-noun* triplets. Amazon [1] is a *user-item-word* tensor parsed from product reviews. We used Porter stemming [24] on review text and removed all users, items, and words that appeared less than five times. Delicious is a *user-item-tag* dataset originally crawled by Görlitz et al. [25] and is also available for download. Random1 and Random2 are both synthetic datasets with nonzeros uniformly distributed. They have the same number of nonzeros and total mode length (i.e., output size), but differ in the length of individual modes.

C. Effects of Distribution on Load Balance

Table II shows the load imbalance with 64 and 128 nodes. Load imbalance is defined as the ratio of the maximum amount of work (tensor nonzeros) assigned to a process to the average amount of work over all processes. DMS-CG suffers severe load imbalance on the Amazon tensor, with the imbalance growing from 2.17 with 64 nodes to 3.86 with 128 nodes. In contrast, DMS-MG has much lower imbalance, with its largest ratios being only 1.08 with 64 nodes on Amazon. DMS-FG is the most balanced, with Zoltan reaching 1.05 on Delicious with 128 nodes and 1.00 on all other that datasets we could partition.

D. Effects of Distribution on Communication Volume

Table III presents results for communication volume with 128 nodes. We only count communication that is a consequence of the tensor decomposition, i.e., the aggregation of partial products during MTTKRP operations and exchanging

Table II: Load imbalance with 64 and 128 nodes.

Dataset	DMS-CG		DMS-MG		DMS-FG	
	64	**128**	**64**	**128**	**64**	**128**
Netflix	1.03	1.18	1.00	1.00	1.00	1.00
Delicious	1.21	1.41	1.01	1.06	1.00	1.05
NELL	1.12	1.29	1.01	1.01	1.00	1.00
Amazon	2.17	3.86	1.08	1.08	part	part

Load imbalance is the ratio of the largest number of nonzeros assigned to a process to the average number of nonzeros per process. **part** indicates that we were unable to compute a hypergraph partitioning in the memory available on 64 nodes. Hypergraph partitioning was performed with the load imbalance parameter set to 1.10.

Table III: Communication volume with 128 nodes.

Dataset	DMS-CG		DMS-MG		DMS-FG	
	max	**avg**	**max**	**avg**	**max**	**avg**
Netflix	674.8K	616.9K	99.3K	56.8K	2.6M	210.5K
Delicious	2.8M	2.3M	2.5M	1.6M	4.2M	719.2K
NELL	3.8M	3.4M	2.5M	1.7M	6.0M	1.2M
Amazon	8.3M	7.3M	4.0M	2.5M	part	part
Random1	72.1M	72.1M	39.5M	39.3M	part	part
Random2	55.2M	55.2M	23.6M	23.5M	part	part

Table values are the communication volumes with 128 MPI processes. **max** is the maximum volume of any MPI process and **avg** is the average volume. **part** indicates that we were unable to compute a hypergraph partitioning in the memory available on 64 nodes.

updated rows. We report the average volume per MPI process as well as the maximum over all processes. We define the communication volume as the total number of rows sent and received per iteration, per MPI process. By measuring the total number of rows communicated, and not the number of *words*, our discussion is independent of the rank of the decomposition. When $F = 1$, the number of communicated rows is equal to the communicated words.

When the each process owns (I/p) rows of a factor, the worst case communication volume results from sending (I/p) rows to p processes and receiving $I - (I/p)$ rows for a total volume of $2I - (I/p)$. The maximum volume over all modes is

$$V_{max} = 2I + 2J + 2K - \frac{I + J + K}{p}.$$

DFACTO uses a pessimistic approach to communication and always has a communication volume of V_{max}. DMS-CG uses the same decomposition as DFACTO but instead utilizes an optimistic approach in which only the necessary factor rows are stored and communicated. Resultingly, DMS-CG has a smaller communication volume than V_{max} on all datasets that we were able to collect results for. Despite the added communication step of aggregating partial results during the MTTKRP operations, DMS-MG and DMS-FG exhibit lower average communication volumes than DMS-CG on all datasets.

DMS-FG has the lowest average volume on all datasets except Netflix. The discrepancy between mode lengths is largest on Netflix, resulting in DMS-MG using a $64\times2\times1$ decomposition of the tensor. By using most of processes to partition only the longest mode, the majority of the possible communication volume is constrained to the $p_{(i,:,:)}$ layers which have only two processes each. DMS-MG avoids partitioning the other tensor modes in exchange for greatly reducing the communication along the longest mode.

While the average communication volumes are lowest with DMS-FG, this method also sees the largest maximum volumes. Hypergraph partitioners optimize the total communication volume, not necessarily the maximum over any process. Additionally, with fine-grained decompositions a process may have to exchange rows with all other processes instead of being bounded by the size of a layer. Thus, some processes can exhibit very large communication volumes in exchange for a lower average.

E. Strong Scaling

Table IV shows the runtimes of our methods and DFACTO. We scale from 2 to 128 computing nodes and measure the time to perform one iteration of CPD-ALS averaged over 50 runs. Each node has eight processors available which we utilize. DMS is a hybrid MPI+OpenMP code and so we use one MPI process and eight OpenMP threads per node. DFACTO is a pure MPI code and so we use eight MPI processes per node.

The DMS methods are faster than DFACTO on all datasets. DMS-MG is $41\times$ faster on Amazon and $76\times$ faster on Delicious when both methods use 128 nodes (1024 cores). Our success is due to several key optimizations. The three DMS methods begin faster on small node counts due to an MTTKRP algorithm which on average is $5\times$ faster [11]. As we increase the number of nodes, DMS methods outscale DFACTO due to their ability to exploit parallelism in the dense matrix operations that take place after the MTTKRP operation. DMS methods also use significantly less memory than DFACTO, which is unable to factor some of our large datasets. This is due to a combination of our optimistic factor storage and our MPI+OpenMP hybrid code. DFACTO must replicate factors on every core to exploit multi-core architectures. Even in the worst case, the DMS methods only need one copy of each matrix factor (and in practice, almost always less than one copy).

DMS-FG was unable to partition the tensors with billions of nonzeros due to the overhead of hypergraph partitioning. It is important to note that fine-grained decompositions are not limited to only the hypergraph model, and nonzeros could instead be randomly assigned to processes. However, experimental results in [9] show that random assignment results in runtime performance that is comparable to a coarse-grained decomposition. On the tensors we were able to factor, its performance is very comparable to DMS-CG on Netflix and Delicious, but DMS-CG is $1.7\times$ faster on NELL.

Table IV: Strong scaling results.

	Netflix				Delicious				NELL			
Nodes	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**
1	11.34	**1.82**	**1.82**	**1.82**	mem	**7.90**	**7.90**	**7.90**	mem	**10.82**	**10.82**	**10.82**
2	6.07	1.16	**0.84**	1.03	mem	4.82	**4.11**	6.98	mem	6.66	**6.01**	9.14
4	3.24	0.64	**0.37**	0.56	mem	3.08	**2.23**	4.43	mem	4.06	**3.32**	5.24
8	1.90	0.39	**0.18**	0.31	28.01	1.88	**1.25**	2.16	mem	2.55	**2.02**	3.46
16	1.34	0.23	**0.09**	0.22	25.54	1.26	**1.04**	1.35	mem	1.64	**1.16**	2.33
32	0.95	0.20	**0.06**	0.20	24.93	0.86	**0.59**	0.96	mem	1.09	**0.82**	1.74
64	0.82	0.19	**0.04**	0.19	25.15	0.81	**0.37**	0.66	mem	0.76	**0.55**	1.16
128	1.33	0.14	**0.05**	0.24	24.34	0.42	**0.32**	0.48	mem	0.53	**0.35**	0.92

(a)

	Amazon				Random1				Random2			
Nodes	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**
8	mem	mem	**8.34**	part	mem	mem	**18.25**	part	mem	mem	**16.27**	part
16	64.12	13.07	**4.30**	part	mem	12.80	**11.42**	part	mem	mem	**9.61**	part
32	50.92	10.06	**2.19**	part	mem	9.98	**8.12**	part	mem	10.61	**6.25**	part
64	45.29	10.82	**1.80**	part	mem	8.02	**5.51**	part	mem	7.86	**4.06**	part
128	40.20	7.82	**0.97**	part	mem	6.85	**3.96**	part	mem	5.53	**2.81**	part

(b)

Table values are seconds per iteration of CPD-ALS, averaged over 50 iterations. **mem** indicates the configuration required more memory than available. **part** indicates that we were unable to compute a data partitioning. Each node has eight cores which are fully utilized.

DMS-MG is the fastest method among the DMS implementations. It ranges from $1.3\times$ to $8.0\times$ faster than DMS-CG and $1.5\times$ to $5.0\times$ faster than DMS-FG In many cases, DMS-MG is able to factor tensors when other methods cannot due to memory limitations or the hypergraph partitioning overhead. Figure 3 graphs the strong scaling results for the Netflix dataset. DMS-MG maintains near-perfect speedup through 512 cores. Between 16 and 128 cores, DMS-MG achieves speedups which are super-linear. We attribute this behavior to the decomposition shape that DMS-MG chooses. As discussed in Section VI-D, almost all processes are assigned to the first mode of the tensor. In addition to decreasing the communication volume, this has the added effect of decreasing the amount of $\mathbf{A}$ that is stored and accessed on each node. As a result, the memory hierarchy is better utilized during the computational kernels. Interestingly, both DMS-MG and DMS-FG slow down between 512 and 1024 cores. This is a result of communication imbalance. While the *average* communication volume per node continues to decrease as we scale, we find that the *maximum* communication increases after 64 nodes (512 cores). DMS-CG is able to decrease both the average and maximum communication volume due to it having a much larger amount of communication.

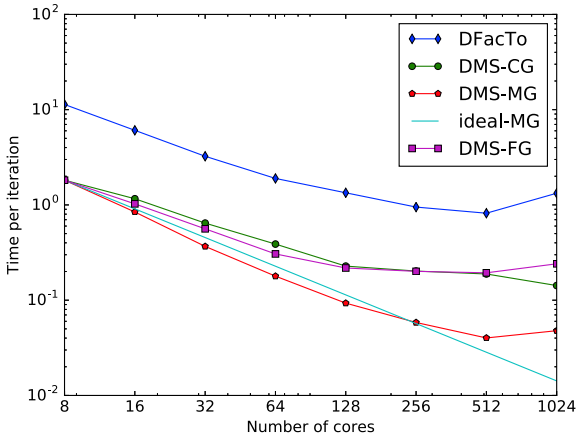


Figure 3: Average time per iteration in seconds on the Netflix dataset. **ideal-MG** indicates perfect scalability relative to **DMS-MG**.

VII. Conclusions and Future Work

We introduced a medium-grained decomposition for sparse tensor factorization. The decomposition addresses the limitations of coarse-grained methods by avoiding complete replication and communication of the factors. In addition, the medium-grained decomposition does not require computationally expensive pre-processing such as hypergraph partitioning to have a low communication volume.

Our implementation of medium-grained CPD-ALS algorithm, DMS-MG, is a lightweight MPI+OpenMP hybrid that further reduced memory footprint compared to pure MPI. We compared against DFacTo, a state of the art distributed CPD-ALS tool as well our own implementation of coarse- and fine-grained methods. We found DMS-MG to be $41\times$ to $76\times$ faster than DFacTo and $1.5\times$ to $5.0\times$ faster than a fine-grained implementation with 1024 cores. Using only eight computing nodes, DMS-MG is capable of factoring a real-world tensor with 1.7 billion nonzeros in less than ten minutes.

ACKNOWLEDGMENTS

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

REFERENCES

[1] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.

[2] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *In AAAI*, 2010.

[3] J. C. Ho, J. Ghosh, and J. Sun, "Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 115–124.

[4] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 316–324.

[5] T. G. Kolda and B. Bader, "The TOPHITS model for higher-order web link analysis," in *Proceedings of Link Analysis, Counterterrorism and Security 2006*, 2006.

[6] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver, "Tfmap: optimizing map for top-n context-aware recommendation," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 155–164.

[7] J. H. Choi and S. Vishwanathan, "DFacTo: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.

[8] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *Data Mining (ICDM), 2014 IEEE International Conference on*, Dec 2014, pp. 989–994.

[9] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[10] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[11] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *International Parallel & Distributed Processing Symposium (IPDPS'15)*, 2015.

[12] N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis, "Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2014.

[13] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[14] Q. Zhang, M. W. Berry, B. T. Lamb, and T. Samuel, "A parallel nonnegative tensor factorization algorithm for mining global climate data," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 405–415.

[15] A. P. Liavas and N. D. Sidiropoulos, "Parallel algorithms for constrained tensor factorization via the alternating direction method of multipliers," *arXiv preprint arXiv:1409.2383*, 2014.

[16] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 7.

[17] D. M. Pelt and R. H. Bisseling, "A medium-grain method for fast 2d bipartitioning of sparse matrices," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 529–539.

[18] U. V. Catalyurek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[19] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM review*, vol. 47, no. 1, pp. 67–95, 2005.

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[21] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 974–996, 2004.

[22] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing." IEEE, 2006.

[23] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[24] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[25] O. Görlitz, S. Sizov, and S. Staab, "Pints: peer-to-peer infrastructure for tagging systems." in *IPTPS*, 2008, p. 19.