DMS: Distributed Sparse Tensor Factorization with Alternating Least Squares

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

***Abstract*—Tensors are data structures indexed along three or more dimensions. Tensors have found increasing use in domains such as data mining and recommender systems where dimensions can have enormous length and are resultingly very sparse. 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). Factoring large, sparse tensors is a computationally challenging task which can no longer be done in the memory of a typical workstation. State of the art methods for distributed memory systems have focused on distributing the tensor in a one-dimensional (1D) fashion that prohibitively requires the dense matrix factors to be fully replicated on each node. To that effect, we present DMS, a novel distributed CPD-ALS algorithm. DMS uses a 3D decomposition that avoids complete factor replication and communication. DMS has a hybrid MPI+OpenMP implementation that exploits multi-core architectures with a low memory footprint. We theoretically evaluate DMS against leading CPD-ALS methods and experimentally compare them across a variety of datasets. Our 3D decomposition reduces communication volume by 74% on average and is over 35x faster than state of the art MPI code on a tensor with 1.7 billion nonzeros.**

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*). An example is a tensor of Amazon product reviews modeled as *user-item-word* triplets [1]. Similarly, the Never Ending Language Learning (NELL) project represents its dataset as *subject-verb-object* triplets [2]. These tensors have very long modes and are consequently very sparse (e.g., NELL has a density of $9{\times}10^{-13}$).

The recent popularity of tensors has led to the increased use of tensor factorization, a useful tool for discovering the latent features in multi-dimensional data. The most popular factorization is the canonical polyadic decomposition (CPD), a rank decomposition that is a higher-dimensional interpretation of the singular value decomposition. The CPD outputs a low-rank representation of the tensor via a matrix of latent features for each mode. The columns of the factors often represent some real-world interpretation of the dataset such as film genre or word category. Observing factor entries with large values reveals items with some importance to the latent features. This technique has been used with great success to perform tasks such as identifying word synonyms [3], webpage queries [4], and top-N recommendation [5].

Finding the CPD is a non-convex optimization problem that has only recently been studied in the context of high performance computing. The most common method of computing the CPD is using the method of alternating least squares (CPD-ALS), which approximates the problem by turning each iteration into a sequence of convex least squares solutions.

Large tensors cannot easily be factored in the memory of a typical workstation. We must turn to distributed computing to factor the tensors of today and the future. Two recent systems for distributed tensor factorization are DFacTo [6], and SALS [7]. A drawback to both methods is their *memory scalability*. While they are able to partition the input tensor across a distributed system, they still prohibitively require the dense matrix factors to be present on each node. Without scalability in both the input tensor and the output factors, larger tensors cannot be factored by simply increasing computing nodes. The factors can consume more memory than the original sparse tensor and each node will still need enough memory to hold the entire problem output.

To address these limitations, we present a novel CPD-ALS algorithm which allows memory and communication to scale with process count. This is achieved through a 3D decomposition detailed in Section IV. Our algorithm is realized by a distributed memory extension of SPLATT [8], named DMS (Distributed Memory SPLATT). Our contributions are:

1) DMS, a memory-scalable CPD-ALS algorithm for distributed memory systems that uses a 3D decomposition on the tensor.
2) We theoretically compare the parallel complexity of DMS to the state of the art and show that it reduces communication overhead from $O(I)$ to $O\left((I/\sqrt[3]{p})\log p\right)$, where I is the longest dimension and p is the number of processes.
3) We experimentally compare DMS to the state of the art. Our 3D decomposition reduces communication volume by 74% on average and DMS is over 35x faster than state of the art MPI code on a tensor with 1.7 billion nonzeros.

II. Tensor Background

In this section we provide a brief background on tensors and the CPD. For more information on tensors and their factorizations, we direct the reader to the excellent survey by Kolda and Bader [9].

A. Tensor Notation

In this work we focus on third-order tensors. However, we stress that all of our methods are easily extended to work with higher-order tensors. We discuss the extension of our methods to higher modes in Section V-G.

We denote matrices as $\mathbf{A}$ and tensors as $\boldsymbol{\mathcal{X}}$. 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 $nnz(\boldsymbol{\mathcal{X}})$ 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 an extension of the Singular Value Decomposition (SVD) to tensors. In the SVD, a matrix $\mathbf{M}$ is decomposed 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, we do not require orthogonality in the columns of the factors. We output the factors with normalized columns and $\lambda \in \mathbb{R}^F$, a vector for scaling. Using this form we can reconstruct $\boldsymbol{\mathcal{X}}$ via

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

Algorithm 1 CPD-ALS

while not converged **do**
  $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}$
  Normalize columns of $\mathbf{A}$
  $\mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{C}^\intercal\mathbf{C} * \mathbf{A}^\intercal\mathbf{A})^{-1}$
  Normalize columns of $\mathbf{B}$
  $\mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A})^{-1}$
  Normalize columns of $\mathbf{C}$ and store in λ
  Check convergence
end while

C. CPD with Alternating Least Squares

CPD-ALS is the most common algorithm for computing the CPD. We transform the non-convex problem into a convex one for each factor and iterate until convergence. During each iteration we fix $\mathbf{B}$ and $\mathbf{C}$ and 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

$$\begin{aligned} \mathbf{A} &= \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \left[(\mathbf{C} \odot \mathbf{B})^\intercal(\mathbf{C} \odot \mathbf{B})\right]^{-1} \\ &= \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}. \end{aligned}$$

We first find $\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$, followed by $\mathbf{M} = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}$. $\mathbf{M}$ is an $F \times F$ symmetric positive definite matrix and so we use its Cholesky factorization to compute the inverse. $\mathbf{B}$ and $\mathbf{C}$ are then solved for similarly. The factors are normalized each iteration and λ stores the F column norms. CPD-ALS is summarized 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 MTTKRP in $c \cdot F \cdot nnz(\boldsymbol{\mathcal{X}})$ FLOPs, with c between 2 and 3 and dependent on the sparsity pattern of the tensor [6], [8], [10]. Entry $\hat{\mathbf{A}}(i, f)$ is equal to

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

Equation (1) shows us two important properties of MTTKRP. 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 $\boldsymbol{\mathcal{X}}(i, :, :)$ precisely determine which rows of $\mathbf{B}$ and $\mathbf{C}$ must be accessed during the multiplication. These properties are critical to designing scalable CPD-ALS algorithms.

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 Kruskal tensor is

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

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 $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle =$

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

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

III. Related Work

Distributed algorithms for CPD typically fall into two broad approaches. The first approach, and the one we explore in this work, exploits the naturally parallel computations in the traditional CPD-ALS algorithm. The second class of algorithms instead utilize the uniqueness of the CPD to compute separate factorizations on sub-tensors in parallel which are then joined to form some global factorization.

Within the first class, distributed CPD algorithms such as DFacTo [6] and SALS [7] use independent one-dimensional (1D) decompositions 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 MTTKRP. Each process owns all of the nonzeros that contribute to its owned output and thus no communication is required during the multiplication. Independent 1D decompositions can be interpreted as a task decomposition on the problem output, often called the *owner-computes rule*.

A limitation of these 1D methods is that by owning entire slices of the tensor, all processes own nonzeros that collectively can span the complete modes of the input. From Equation (1) we can see that every row of the factors will contribute to the MTTKRP output. 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.

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 [11]. A 1D decomposition was used on the tensor and factors. Recently, [12] presented a generalized framework for constrained CPD that uses the Alternating Direction Method of Multipliers (ADMM). Parallelism is extracted by performing a 2D decomposition on the matricized tensor and a row distribution of the factors. Although [12] supports sparsity, neither of the two works were explicitly designed for sparse tensors and thus the storage and communication of full factors is not considered a limitation.

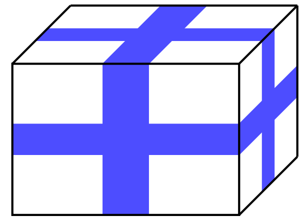


Figure 1: Independent 1D decompositions of $\boldsymbol{\mathcal{X}}$. Slices owned by process p_i are shaded.

Algorithms following the second approach [13], [14] extract parallelism by distributing small tensors to each process and doing independent factorizations in parallel. The resulting factorizations are then carefully joined, resulting in a factorization of the original input tensor. The convergence of these methods varies from CPD-ALS methods, and so we leave a comparison against these to future work.

IV. 3D Tensor Decomposition

Assume that we have $p = p_1 \times p_2 \times p_3$ processing elements available. We begin with a 1D decomposition on the output factors and divide the rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ into p_1, p_2, and p_3, chunks, respectively. Applying the *owner-computes rule* to the chunks of $\mathbf{A}$, each resulting task requires the corresponding mode-1 slices of $\boldsymbol{\mathcal{X}}$ and, consequently, the entirety of $\mathbf{B}$ and $\mathbf{C}$. The tasks for $\mathbf{B}$ and $\mathbf{C}$ similarly require the corresponding mode-2 and mode-3 slices and factors. We further decompose the tasks of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ using 2D decompositions on the tensor slices of size $p_2 \times p_3$, $p_1 \times p_3$ and $p_1 \times p_2$, respectively. We refer to each set of processors along which slices are distributed as a *layer*.

Layers are used during the MTTKRP stage of CPD-ALS. Consider the MTTKRP computations required to compute $\mathbf{A}_q$, the q-th chunk of $\mathbf{A}$. The processes in layer q further divide the rows of $\mathbf{A}_q$ into $l = p_2 \times p_3$ chunks, $\mathbf{A}_{q_1}, \mathbf{A}_{q_2}, \ldots, \mathbf{A}_{q_l}$, and each chunk is assigned to a process. Using a 2D decomposition on tensor layers allows us to limit the number of rows of $\mathbf{B}$ and $\mathbf{C}$ accessed by any single process to just the dimensions of the 2D chunk. Each set of l processes work collectively to perform the MTTKRP computations associated with $\mathbf{A}_q$. This is done in two steps. First, each process computes its own contribution to the $\mathbf{A}_q$ entries for which it has portions of the tensor. Second, each process aggregates the partial results computed by the other processors for the rows of $\mathbf{A}_q$ that it is storing.

DMS uses a *single* 3D decomposition of $\boldsymbol{\mathcal{X}}$ that is based on applying 1D decompositions to each factor. Processes are mapped to a 3D grid and given coordinates of the form

(q, r, s). The coordinate of a process identifies $\boldsymbol{\mathcal{X}}_{q,r,s}$, the 3D sub-tensor owned by process (q, r, s). Figure 2 illustrates the task decomposition over the factors and the resulting 3D decomposition over $\boldsymbol{\mathcal{X}}$. In subsequent discussions we identify processes by two identities: a linear numbering p_i and a mapping to the 3D grid that our decomposition operates on, $p_{q,r,s}$.

The discussion so far has provided a high-level overview of how the processes are organized and how the data is distributed among them. We now provide details on the specifics of the data distribution that DMS uses in order to balance the computations and minimize the communication overhead.

A. Tensor Partitioning

Our objective is to define a $p_1 \times p_2 \times p_3$ grid over $\boldsymbol{\mathcal{X}}$. The boundaries of each layer are chosen in order to minimize load imbalance by balancing the number of tensor nonzeros that each layer contains.

We begin with a random permutation of $\boldsymbol{\mathcal{X}}$. Uniformly distributing nonzeros removes any ordering from the data collection process that could result in load imbalance. Each mode is partitioned separately. Assume we are splitting a mode into p_1 parts. We greedily assign partition boundaries by adding consecutive indices until a partition has at least $nnz(\boldsymbol{\mathcal{X}})/p_1$ nonzeros. Slices can vary in density and adding a heavy slice 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 is closer to the target. After performing this process on all modes we have chosen the $p_1 \times p_2 \times p_3$ grid that defines the tensor decomposition and we can now distribute $\boldsymbol{\mathcal{X}}$ to p processes.

B. Factor Partitioning

We partition the factors after distributing the tensor. The matrix partitioning directly affects the number of factor rows which are exchanged during MTTKRP. Our objective is to minimize the total number of communicated factor rows, or the *communication volume*. We adapt a greedy method developed for two-dimensional sparse matrix-vector multiplication [15].

Chunks of $\mathbf{A}$ are partitioned independently. For each row r in $\mathbf{A}_q$, processes count the number of tensor partitions (and thus, processes) that contain a nonzero value in slice $\boldsymbol{\mathcal{X}}(r, :, :)$. Any row that is found in only a single partition is trivially claimed by the owner because it will not increase communication volume. Next, the master process in the layer 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. Process p_j is instructed 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 fraction of the remaining rows.

Following the partitioning of $\mathbf{A}_q$, we reorder the indices of $\boldsymbol{\mathcal{X}}$ in order to make the rows owned by each process contiguous. The partitioning step proceeds for the other modes similarly.

V. DMS

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.

A. MTTKRP and Factor Inverse

After the tensor and matrix distribution, each process has the tensor nonzeros and the necessary non-local matrix rows residing in memory. Each process performs MTTKRP with $\boldsymbol{\mathcal{X}}_{q,r,s}$ to compute $\hat{\mathbf{A}}_{p_{q,r,s}}$. DMS uses the efficient fiber-based data structure of SPLATT to parallelize MTTKRP with OpenMP and to utilize the CPU cache hierarchy of modern multi-core architectures [8]. The result of the multiplication may have a combination of local and non-local rows. All processes of layer q exchange non-local rows and add the received partial products to their local $\hat{\mathbf{A}}_{p_{q,r,s}}$. After this step we have $\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ distributed row-wise among all processes.

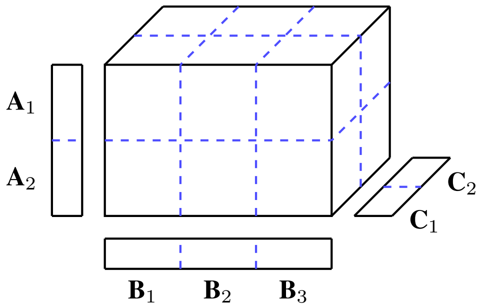
$\mathbf{B}^\intercal\mathbf{B}$ and $\mathbf{C}^\intercal\mathbf{C}$ are $F \times F$ matrices that comfortably fit in the memory of each process. Assume $\mathbf{B}^\intercal\mathbf{B}$ and $\mathbf{C}^\intercal\mathbf{C}$ are already resident in each process' memory. Processes redundantly compute $\mathbf{M} = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1}$ in $O(F^3)$ time, which is negligible for the very low-rank problems that we are interested in. We compute the final matrix multiplication in block form to exploit our distribution scheme

$$\begin{bmatrix} \mathbf{A}_{p_1} \\ \mathbf{A}_{p_2} \\ \dots \\ \mathbf{A}_{p_p} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{A}}_{p_1} \\ \hat{\mathbf{A}}_{p_2} \\ \dots \\ \hat{\mathbf{A}}_{p_p} \end{bmatrix} \mathbf{M} = \begin{bmatrix} \hat{\mathbf{A}}_{p_1}\mathbf{M} \\ \hat{\mathbf{A}}_{p_2}\mathbf{M} \\ \dots \\ \hat{\mathbf{A}}_{p_p}\mathbf{M} \end{bmatrix}.$$

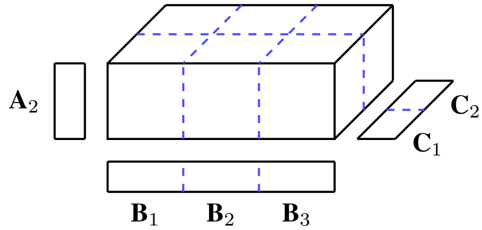
Node-level matrix multiplication is further parallelized with OpenMP. We do a 1D decomposition on the rows of $\mathbf{A}_{p_i}$ to extract parallelism.

B. Column Normalization

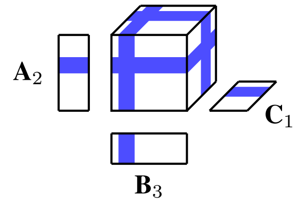
After computing the new $\mathbf{A}$, we normalize its columns and store the norms in λ. Processes first compute the column norms of $\mathbf{A}_{p_i}$ and collectively find the global λ with a parallel reduction. Finally, each process normalizes the columns of $\mathbf{A}_{p_i}$ with λ. Nodes parallelize the normalization process by finding thread-local norms which are reduced in parallel before the global λ is found.



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



(b) Processes $p_{2,:,:}$ collectively own $\mathbf{A}_2$.



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

Figure 2: The 3D decomposition scheme used by DMS.

C. Exchanging the Updated Rows of A

All process layers must exchange the updated rows of $\mathbf{A}$. This communication is a dual of the MTTKRP exchange. Any processes that sent partial MTTKRP products to process i must now receive the updated rows of $\mathbf{A}_{p_i}$.

D. Forming the New $\mathbf{A}^\intercal\mathbf{A}$

Each process needs the updated $\mathbf{A}^\intercal\mathbf{A}$ 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

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

Each process first forms its local $\mathbf{A}_{p_i}^\intercal\mathbf{A}_{p_i}$. A 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.

E. Residual Computation

Convergence is tested after every iteration. Residual computation cost is bounded by $\langle\boldsymbol{\mathcal{X}},\boldsymbol{\mathcal{Z}}\rangle$, which uses $4F{\cdot}nnz(\boldsymbol{\mathcal{X}})$ FLOPs. We can instead cache $\hat{\mathbf{A}}$ and rewrite Equation (2) as $\langle\boldsymbol{\mathcal{X}},\boldsymbol{\mathcal{Z}}\rangle =$

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

where $\mathbf{1}$ is the vector of all ones. This reduces the computation to $2IF$ FLOPs. Each process computes its own local $\mathbf{1}^\intercal\left(\hat{\mathbf{A}}_{p_i} * \mathbf{A}_{p_i}\right)\lambda$. 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 $\langle\boldsymbol{\mathcal{X}},\boldsymbol{\mathcal{Z}}\rangle$. DMS then iterates until the residual is below some threshold or we have reached the maximum number of iterations.

F. Complexity Analysis

The cost of CPD-ALS is bounded by MTTKRP and its associated communication. Both 1D and 3D distributed algorithms distribute work such that each process does $O(nnz(\boldsymbol{\mathcal{X}})/p)$ work. They differ, however, in the overheads associated with communication. In our discussion we will use two collective communication operations: All-Reduce and All-to-All. Derivation of their complexities can be found in [16].

Assume that $\boldsymbol{\mathcal{X}}$ is of dimension $I{\times}I{\times}I$ and processes are arranged in a $\sqrt[3]{p}{\times}\sqrt[3]{p}{\times}\sqrt[3]{p}$ grid. A 3D decomposition has two communication overheads to consider: reducing non-local rows during MTTKRP and sending updated rows of $\mathbf{A}_{p_i}$ after an iteration. In the worst case, every process has nonzeros in all $(I/\sqrt[3]{p})$ slices of the layer. Processes must send all but their owned rows, totaling

$$\frac{I}{\sqrt[3]{p}} - \frac{I}{p} = \frac{I(p^{2/3}-1)}{p}.$$

The communication will involve all $p^{2/3}$ processes in the layer. Sending all $(I/\sqrt[3]{p})$ rows and adding to $\hat{\mathbf{A}}_{p_i}$ is most efficiently implemented as an All-Reduce communication whose total complexity is

$$\frac{I(p^{2/3}-1)}{p}\log p^{2/3} = O\left(\frac{I}{\sqrt[3]{p}}\log p\right). \qquad (4)$$

The worst case of the update stage is sending (I/p) rows to all $p^{2/3}$ neighbors. The cost of this operation as an All-to-All communication is

$$\frac{I}{p}\left(p^{2/3}-1\right) = O\left(\frac{I}{\sqrt[3]{p}}\right). \qquad (5)$$

Ultimately, the total overhead associated with our 3D decomposition is the sum of Equations (4) and (5),

$$T_o^{3D} = O\left(\frac{I}{\sqrt[3]{p}}\log p\right) + O\left(\frac{I}{\sqrt[3]{p}}\right) = O\left(\frac{I}{\sqrt[3]{p}}\log p\right). \qquad (6)$$

In comparison, a 1D decomposition will send up to (I/p) rows to all p processes. The communication overhead due to the 1D decomposition using an All-to-All communication is

$$T_o^{1D} = \frac{I}{p}(p-1) = O\left(I\right). \tag{7}$$

No partial MTTKRP results need to be communicated, however, so Equation (7) is the only communication associated with a 1D decomposition. Comparing Equations (6) and (7) shows us that only the 3D decomposition has scalable communication costs. We experimentally evaluate this observation in Section VII-A.

G. Extensions to Higher Modes

Extending our distributed CPD-ALS algorithm to tensors with an arbitrary number of modes is straightforward. Suppose $\boldsymbol{\mathcal{X}}$ is a tensor with n modes and we wish to compute factors $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(n)}$.

Our tensor distribution does an independent partitioning of each mode to define process layers, resulting in an n-dimensional tensor decomposition. During the partitioning of a mode we do not use information from other modes. Thus, there are no complications to consider when generalizing to higher modes. Our factor partitioning is also not dependent on the number of modes and is extended similarly.

An efficient MTTKRP algorithm for a general number of modes is found in [8]. 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 $n-1$.

Residual computation again is easily extended. Generalized MTTKRP computes

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

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.

VI. Experimental Methodology

A. Experimental Setup

We implemented two versions of DMS. The first uses the 3D decomposition described in Section IV and is denoted DMS-3D. Our second method, DMS-1D, uses a separate 1D decomposition for each mode. DMS-1D uses the same computational kernels as DMS-3D but skips aggregation of non-local MTTKRP products. Both DMS implementations avoid storing unnecessary non-local factor rows. Only the rows corresponding to non-empty tensor slices are stored and communicated.

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 two.

Table I: Summary of datasets.

Dataset	I	J	K	nnz	density
Netflix	480K	18K	2K	100M	5.4e-06
Delicious	532K	17M	3M	140M	6.1e-12
NELL	3M	2M	25M	143M	9.0e-13
Amazon	5M	18M	2M	1.7B	1.1e-10
Random1	20M	20M	20M	1.0B	1.3e-13
Random2	50M	5M	5M	1.0B	8.0e-13

nnz is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively. **density** is defined by $nnz/(IJK)$.

We compare against DFACTO, which to our knowledge is the fastest tensor factorization software available today. DFACTO is implemented in C++ and uses MPI for distributed memory parallelism. DFACTO uses the same 1D decomposition as DMS-1D, but each process explicitly stores entire factors. All processes perform a local MTTKRP and results are gathered so that all processes then have the complete MTTKRP output. Factors are updated redundantly on all processes and the iteration proceeds.

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 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 [17] 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 [18] 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. [19] 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.

VII. Results

A. Effects of Distribution on Communication Volume

Table II presents results for communication volume. We define the communication volume as the total number of factor rows sent and received per iteration, per MPI rank. We only count communication that is a consequence of the tensor decomposition, i.e., MTTKRP aggregation and exchanging updated rows. The worst case communication

[1] *http://cs.umn.edu/~shaden/software/*

Table II: Communication volume with eight MPI ranks.

Dataset	Naive	1D	8x1x1	4x2x1	2x2x2
Netflix	937K	0.92	**0.08**	0.29	0.73
Delicious	38.5M	0.41	**0.08**	0.18	0.21
NELL	56.3M	0.41	**0.13**	0.17	0.23
Amazon	46.9M	no-mem	0.19	**0.16**	0.19
Random1	112.5M	no-mem	1.24	0.98	**0.80**
Random2	112.5M	no-mem	**0.31**	0.44	0.79

Table values are the ratio of communication volume to Naive. The volume is averaged over all MPI ranks. **no-mem** indicates the configuration required more memory than available on eight nodes. **Naive** is the volume sent using a pessimistic approach, V_{max}. **1D** is a separate one-dimensional decomposition for each mode. **8x1x1**, **4x2x1**, and **2x2x2** are various 3D configurations. Longer modes received more ranks in the non-symmetric configurations.

volume for a mode 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

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

Note that for 3D decompositions it is possible to communicate V_{max} during both MTTKRP aggregation and also row updates. In practice, we found that V_{max} was never reached.

DFACTO always has a communication volume of V_{max}. DMS-1D 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-1D has a smaller communication volume than V_{max} on all datasets that we were able to collect results for.

Despite the added communication step due to aggregation of MTTKRP partial results, 3D configurations exhibited lower communication volumes than 1D on all datasets. 3D decompositions averaged 74% lower communication volume than V_{max}. Among the 3D configurations there is no clear winner; the best configuration is closely tied to the lengths of each tensor mode. Tensors with one mode significantly longer than the rest (e.g., Netflix and Delicious) achieved the best results when all ranks were used to partition the long mode. In contrast, Random1, with its equal mode lengths, performed best with a symmetric configuration.

B. Scaling

Table III compares the runtime and scalability of our methods and DFACTO. We scale from two to sixty-four nodes and measure the time to perform one iteration of CPD-ALS. Each node has eight processors available which we utilize. DMS is a hybrid MPI+OpenMP code and so we use one MPI rank and eight OpenMP threads per node. We use 3D configurations that assign ranks proportional to mode length. DFACTO is a pure MPI code and so we use eight MPI ranks per node.

DMS is faster than DFACTO on all datasets. DMS-3D is $37\times$ faster on Amazon and $67\times$ faster on Delicious when both methods use 64 nodes. Our success is due to several key optimizations. DMS begins faster on small node counts due to an MTTKRP algorithm which on average is over $5\times$ faster [8]. As we add nodes, DMS out-scales DFACTO due to its ability to exploit parallelism in the dense matrix operations that take place after MTTKRP. DMS also uses significantly less memory than DFACTO, which is unable to factor some of our large datasets even with 64 nodes. 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, DMS-1D only needs one copy of each (and in practice, almost always less than one copy).

We see that a 3D decomposition is faster than 1D in all cases and also has a smaller memory footprint, resulting in the ability to compute on a smaller number of nodes than 1D. The improvement in runtime can be attributed to two details. First, DMS-3D consistently has a smaller communication volume than DMS-1D and thus spends less time communicating. Second, limiting the number of factor rows which are accessed during MTTKRP results in better utilization of the CPU cache hierarchy. Processes do the same amount of work in both decompositions, but DMS-3D accesses a smaller amount of memory during computation.

VIII. CONCLUSIONS AND FUTURE WORK

We introduced DMS, a CPD-ALS algorithm for distributed memory systems that uses a novel 3D tensor decomposition. The decomposition reduced memory consumption, communication volume, and resultingly, runtime. DMS was implemented as a lightweight MPI+OpenMP hybrid that further reduced memory footprint. We compared against a state of the art distributed CPD-ALS tool and found DMS to be over $35\times$ faster on a tensor with 1.7 billion nonzeros and over $65\times$ faster on a tensor with 140 million nonzeros.

As tensor factorization continues to grow in popularity, there exist several items of future work. Adding constraints such as non-negativity is of serious interest to the community. Distributed optimization algorithms such as ADMM have been applied to tensor factorization with promising results [12]. Likewise, methods which combine independent factorizations show strong potential for large-scale parallel architectures. We need more research on these algorithms for the parallel architectures of today and tomorrow.

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.

	Netflix			Delicious			NELL		
Nodes	**DFacTo**	**DMS-1D**	**DMS-3D**	**DFacTo**	**DMS-1D**	**DMS-3D**	**DFacTo**	**DMS-1D**	**DMS-3D**
2	6.07	0.99	**0.88**	no-mem	4.78	**4.08**	no-mem	6.71	**6.57**
4	3.24	0.56	**0.39**	no-mem	2.87	**2.18**	no-mem	3.98	**3.51**
8	1.90	0.46	**0.19**	28.01	1.93	**1.30**	no-mem	2.46	**1.99**
16	1.34	0.32	**0.12**	25.54	1.27	**0.72**	no-mem	1.57	**1.37**
32	0.95	0.17	**0.07**	24.93	0.77	**0.68**	no-mem	1.13	**0.84**
64	0.82	0.15	**0.06**	25.15	0.59	**0.37**	no-mem	0.65	**0.53**

(a)

	Amazon			Random1			Random2		
Nodes	**DFacTo**	**DMS-1D**	**DMS-3D**	**DFacTo**	**DMS-1D**	**DMS-3D**	**DFacTo**	**DMS-1D**	**DMS-3D**
4	no-mem	no-mem	no-mem	no-mem	no-mem	no-mem	no-mem	no-mem	no-mem
8	no-mem	no-mem	**6.06**	no-mem	no-mem	**20.17**	no-mem	no-mem	**15.91**
16	64.14	10.60	**3.25**	no-mem	12.41	**12.05**	no-mem	no-mem	**7.43**
32	50.91	7.22	**1.88**	no-mem	10.07	**8.06**	no-mem	8.96	**5.84**
64	45.29	6.83	**1.21**	no-mem	7.71	**5.78**	no-mem	5.16	**3.64**

(b)

Table III: **Scaling Results.** Table values are seconds per iteration of CPD-ALS. **no-mem** indicates the configuration required more memory than available. Each node has eight cores.

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] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] N. D. Sidiropoulos, E. E. Papalexakis, and C. Faloutsos, "A parallel algorithm for big tensor decomposition using randomly compressed cubes (PARACOMP)," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1–5.

[14] A. L. de Almeida and A. Y. Kibangou, "Distributed large-scale tensor decomposition," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 26–30.

[15] 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.

[16] 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.

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

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

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