HAL
archives-ouvertes.fr

Scalable sparse tensor decompositions in distributed memory systems

Oguz Kaya, Bora Uçar

▶ **To cite this version:**

Oguz Kaya, Bora Uçar. Scalable sparse tensor decompositions in distributed memory systems. International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), Nov 2015, Austin, TX, United States. 10.1145/2807591.2807624 . hal-01148202v2

HAL Id: hal-01148202

https://hal.inria.fr/hal-01148202v2

Submitted on 14 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scalable Sparse Tensor Decompositions in Distributed Memory Systems

[Technical Paper]

Oguz Kaya
INRIA and LIP (UMR 5668 CNRS, ENS Lyon, UCB Lyon 1, Inria) ENS Lyon, France
oguz.kaya@ens-lyon.fr

Bora Uçar
CNRS and LIP (UMR 5668 CNRS, ENS Lyon, UCB Lyon 1, Inria) ENS Lyon, France
bora.ucar@ens-lyon.fr

ABSTRACT

We investigate an efficient parallelization of the most common iterative sparse tensor decomposition algorithms on distributed memory systems. A key operation in each iteration of these algorithms is the matricized tensor times Khatri-Rao product (MTTKRP). This operation amounts to element-wise vector multiplication and reduction depending on the sparsity of the tensor. We investigate a fine and a coarse-grain task definition for this operation, and propose hypergraph partitioning-based methods for these task definitions to achieve the load balance as well as reduce the communication requirements. We also design a distributed memory sparse tensor library, HyperTensor, which implements a well-known algorithm for the CANDECOMP-/PARAFAC (CP) tensor decomposition using the task definitions and the associated partitioning methods. We use this library to test the proposed implementation of MTTKRP in CP decomposition context, and report scalability results up to 1024 MPI ranks. We observed up to 194 fold speedups using 512 MPI processes on a well-known real world data, and significantly better performance results with respect to a state of the art implementation.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms, Parallel algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs*; G.4 [**Mathematical Software**]: Algorithm design and analysis, Parallel and vector implementations

1. INTRODUCTION

Tensors or multi-dimensional arrays arise in many fields, including analysis of Web graphs [25], knowledge bases [9], product reviews at online stores [7], chemometrics [3], signal processing [16], computer vision [34], and more. Tensor decomposition algorithms are used as an important tool to understand the tensors and glean hidden or latent information from them. Considerable efforts are being put in designing numerical algorithms for different tensor decomposition problems (see a short [15] and a long survey [26]), and algorithmic and software contributions go hand in hand with these efforts [2, 6, 14, 23, 31].

One of the most common tensor decomposition approaches is the CANDECOMP/PARAFAC (CP) formulation, which approximates a given tensor as a sum of rank-one tensors. Two most common methods for computing a CP decomposition are (i) CP-ALS [10, 20], which is based on the alternating least squares method; and (ii) CP-OPT [1], which is based on the gradient descent method. Both of these methods are iterative, where the computational core of each iteration is a special operation called the matricized tensor times Khatri-Rao product (MTTKRP). When the input tensor is sparse and N dimensional, the MTTKRP operation amounts to element-wise multiplication of $N-1$ vectors and scaled reduction of those products according to the sparsity structure of the tensor. This computationally involved operation has received recent interest for efficient execution in different settings such as Matlab [2, 6], MapReduce [23], shared memory [31], and distributed memory [14].

We investigate an efficient parallelization of the MTTKRP operation in distributed memory environments for sparse tensors in the context of the CP decomposition methods CP-ALS and CP-OPT. For this purpose, we formulate two task definitions, a coarse-grain and a fine-grain one. These definitions are given by applying the owner-computes rule to a coarse and a fine-grain partition of the tensor nonzeros. We define the coarse-grain partition of a tensor as a partition of one of its dimensions. In matrix terms, a coarse-grain partition corresponds to a row-wise or a column-wise partition. Two very recent parallel algorithms DFacTo [14] and SPLATT [31] have coarse-grain tensor partitions and hence have coarse-grain tasks. We define the fine-grain partition of a tensor as a partition of its nonzeros. This has the same significance in matrix terms. Based on these two task granularities, we present two parallel algorithms for the MTTKRP operation. We address the computational load balance and communication cost reduction problems for the two algorithms, and present hypergraph partitioning-based models to tackle these problems with off-the-shelf partitioning tools.

The MTTKRP operation also arises in the close variants of CP-ALS and CP-OPT for some other tensor decomposition methods [28]; hence, it has been implemented as a standalone routine [5] to enable algorithm development. Once this operation is efficiently done, the other parts of the de-

SC '15, November 15-20, 2015, Austin, TX, USA
ACM ISBN 978-1-4503-2138-9.
DOI: http://dx.doi.org/10.1145/2807591.2807624

composition algorithms are usually straightforward. For this reason, most of the related work on high performance tensor decomposition algorithms focus on this particular operation. As hinted above, the majority of our contribution is also on the efficiency of the MTTKRP operation. Nonetheless, we design a library for the parallel CP-ALS algorithm to test the proposed MTTKRP algorithms in a suitable context and give experimental results using this library.

The organization of the rest of the paper is as follows. In the next section, we give the notation, describe the MTTKRP operation and CP-ALS method, and review the related work that influenced our efforts. Hypergraph theoretical definitions are also given in this section. In Section 3, we describe the coarse and fine-grain MTTKRP algorithms, analyze their efficient parallelization requirements, and present hypergraph models for reducing the parallelization overhead. Section 4 contains experimental results, where we report speedups and perform comparisons with a state of the art distributed memory CP-ALS implementation.

2. BACKGROUND AND NOTATION

Bold, upper case Roman letters are used for matrices as in $\mathbf{A}$. Matrix elements are shown with the corresponding lowercase letters as in $a_{i,j}$. Matrix sizes are sometimes shown in the lower right corner, e.g., $\mathbf{A}_{I\times J}$. Matlab notation is used to refer to the entire rows and columns of a matrix, e.g., $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ refer to the ith row and jth column of $\mathbf{A}$ respectively.

2.1 Tensors

We use calligraphic font to refer to tensors, e.g., $\mathcal{X}$. The *order* of a tensor is the number of its dimensions, which we denote with N. For the sake of simplicity of the notation and the discussion, we describe all the notation and the algorithms for $N = 3$, even though our algorithms and implementations have no such restriction. We explicitly generalize the discussion to general order-N tensors whenever we find necessary. As in matrices, an element of a tensor is denoted by a lowercase letter and subscripts corresponding to the indices of the element, e.g., the element (i, j, k) of a third-order tensor is $x_{i,j,k}$. A *fiber* in a tensor is defined by fixing every index but one, e.g., if $\mathcal{X}$ is a third-order tensor, $\mathcal{X}_{:,j,k}$ is a mode-1 fiber and $\mathcal{X}_{i,j,:}$ is a mode-3 fiber. A *slice* in a tensor is defined by fixing only one index, e.g., $\mathcal{X}_{i,:,:}$, refers to the ith slice of $\mathcal{X}$ in mode 1. We use $|\mathcal{X}_{i,:,:}|$ to denote the number of nonzeros in $\mathcal{X}_{i,:,:}$.

Tensors can be *matricized* in any mode. This is achieved by identifying a subset of the modes of a given tensor $\mathcal{X}$ as the rows and the other modes of $\mathcal{X}$ as the columns of a matrix and appropriately mapping the elements of $\mathcal{X}$ to those of the resulting matrix. We will be exclusively dealing with the matricizations of tensors along a single mode. For example, take $\mathcal{X} \in \mathbb{R}^{I_1\times\cdots\times I_N}$. Then $\mathbf{X}_{(1)}$ denotes the mode-1 matricization of $\mathcal{X}$ in such a way that the rows of $\mathbf{X}_{(1)}$ corresponds to the first mode of $\mathcal{X}$ and the columns corresponds to the remaining modes. The tensor element $x_{i_1,\ldots,i_N}$ corresponds to the element $\left(i_1, 1 + \sum_{j=2}^{N}\left[(i_j - 1)\prod_{k=1}^{j-1} I_k\right]\right)$ of $\mathbf{X}_{(1)}$. Specifically, each column of the matrix $\mathbf{X}_{(1)}$ becomes a mode-1 fiber of the tensor $\mathcal{X}$. Matricizations in the other modes are defined similarly.

Given two matrices $\mathbf{A}_{I_1\times J_1}$ and $\mathbf{B}_{I_2\times J_2}$, the *Kronecker product* is defined as

$$\mathbf{A}\otimes\mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & \cdots & a_{1,J_1}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{I_1,1}\mathbf{B} & \cdots & a_{I_1,J_1}\mathbf{B} \end{bmatrix}.$$

For $\mathbf{A}_{I_1\times J}$ and $\mathbf{B}_{I_2\times J}$, the *Khatri-Rao product* is defined as

$$\mathbf{A}\odot\mathbf{B} = \begin{bmatrix}\mathbf{A}(:,1)\otimes\mathbf{B}(:,1) & \cdots & \mathbf{A}(:,J)\otimes\mathbf{B}(:,J)\end{bmatrix},$$

which is of size $I_1I_2 \times J$.

For $\mathbf{A}_{I\times J}$ and $\mathbf{B}_{I\times J}$, the *Hadamard product* is defined as

$$\mathbf{A}*\mathbf{B} = \begin{bmatrix} a_{1,1}b_{1,1} & \cdots & a_{1,J}b_{1,J} \\ \vdots & \ddots & \vdots \\ a_{I,1}b_{I,1} & \cdots & a_{I,J}b_{I,J} \end{bmatrix}.$$

The CP-decomposition of rank R (or with R components) of a given tensor $\mathcal{X}$ factorizes $\mathcal{X}$ into a sum of R rank-one tensors. For $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, it yields to $x_{i,j,k} \approx \sum_{r=1}^{R} a_{ir}b_{jr}c_{kr}$ and $\mathcal{X} \approx \sum_{r=1}^{R} a_r \circ b_r \circ c_r$, for $a_r \in \mathbb{R}^I$, $b_r \in \mathbb{R}^J$ and $c_r \in \mathbb{R}^K$, where $\circ$ is the outer product of the vectors. Here the matrices $\mathbf{A} = [a_1, \ldots, a_R]$, $\mathbf{B} = [b_1, \ldots, b_R]$, and $\mathbf{C} = [c_1, \ldots, c_R]$ are called the *factor matrices*, or *factors*. For N-mode tensors, we use $\mathbf{U}_1, \ldots, \mathbf{U}_N$ to refer to the factor matrices.

We are now equipped with the notation to present the Alternating Least Squares (ALS) method for obtaining a rank-R approximation of a tensor $\mathcal{X}$ with the CP-decomposition. A common formulation of CP-ALS is shown in Algorithm 1 for the 3rd order tensors. At each iteration, each factor matrix is recomputed while fixing the other two; e.g., $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$. This operation is performed in the following order: $\mathbf{M}_A = \mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})$, $\mathbf{V} = (\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$, and then $\mathbf{A} \leftarrow \mathbf{M}_A\mathbf{V}$. Here $\mathbf{V}$ is a dense matrix of size $R\times R$ and is easy to compute. The important issue is the efficient computation of the MTTKRP operations yielding $\mathbf{M}_A$, similarly $\mathbf{M}_B = \mathbf{X}_{(2)}(\mathbf{A}\odot\mathbf{C})$ and $\mathbf{M}_C = \mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A})$.

The sheer size of the Khatri-Rao products makes them impossible to compute explicitly; hence, efficient MTTKRP algorithms find other means to carry out the MTTKRP operation (see the next subsection).

Algorithm 1: CP-ALS for the 3rd order tensors

Input : $\mathcal{X}$: A 3rd order tensor
R: The rank of approximation
Output: CP decomposition $[\![\boldsymbol{\lambda}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$

repeat
  $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
  Normalize columns of $\mathbf{A}$
  $\mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C}\odot\mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C})^\dagger$
  Normalize columns of $\mathbf{B}$
  $\mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B}\odot\mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{B}^T\mathbf{B})^\dagger$
  Normalize columns of $\mathbf{C}$ and store the norms as $\boldsymbol{\lambda}$
until *no improvement or maximum iterations reached*

2.2 Related work

SPLATT [31] is an efficient implementation of the MTTKRP operation for sparse tensors on shared memory systems. It is our understanding that the codes are implemented for the 3-mode tensors—there is no experimental

results with the higher order tensors. Their discussion includes the generalization of the techniques to higher order tensors whenever relevant. SPLATT implements the MTTKRP operation based on the slices of the dimension in which the factor is updated, e.g., on the mode-1 slices when computing $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^{\dagger}$. Nonzeros of the fibers in a slice are multiplied with the corresponding rows of $\mathbf{B}$ and the results are accumulated to be later scaled with the corresponding row of $\mathbf{C}$ to compute the row of $\mathbf{A}$ corresponding to the slice. Parallelization is done using OpenMP directives, and the load balance (in terms of the number of nonzeros in the slices of the mode for which MTTKRP is computed) is achieved by using the dynamic scheduling policy. Hypergraph models are used to optimize cache performance by reducing the number of times a row of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are accessed. Smith et al. [31] also use N-partite graphs (where N is the order of the tensor) to reorder the tensors for all dimensions. Experiments are conducted on an HP ProLiant BL280c G6 server with dual 8-core E5-2670 Xeon processors running at 2.6GHz. Smith et al. implement a sparse-tensor vector product algorithm called TVec for carrying out the MTTKRP operation and report speedups with respect to this algorithm. They report 3.7x speedup on the serial execution and 29.8x speedup on 16-way parallel execution of the SPLATT with respect to the TVec.

GigaTensor [23] is an implementation of CP-ALS which follows the Map-Reduce paradigm. All important steps (the MTTKRPs and the computations of the $\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C}$) are performed using this paradigm. A distinct advantage of GigaTensor is that thanks to Map-Reduce, the issues of fault-tolerance, load balance, and out of core tensor data are automatically handled. On a real world data, speedup studies with upto 100 machines (each machine has 2 quad-core Intel 2.83 GHz CPUs) are presented, where the speedup with 100 machines is 1.4 times the speedup with 25 machines. One iteration of CP-ALS as implemented in GigaTensor takes more than 10^3 seconds for a random tensor of size $10^5 \times 10^5 \times 10^5$ with $10^5/50$ nonzeros on 35 machines, eventually reaching between 10^4 and 10^5 seconds for $10^9 \times 10^9 \times 10^9$ with $10^9/50$ nonzeros. The presentation [23] of GigaTensor focuses on three-mode tensors and expresses the map and the reduce functions for this case. To the best of our understanding, additional map and reduce functions are needed for higher order tensors, which would incur overheads.

DFacTo [14] is a distributed memory implementation of the MTTKRP operation. It performs two successive sparse matrix-vector multiplies (SpMV) to compute a column of the product $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. A crucial observation (made also elsewhere [26]) is that this operation can be implemented as $\mathbf{X}_{(2)}^T\mathbf{B}(:, r)$, which can be reshaped into a matrix to be multiplied with $\mathbf{C}(:, r)$ to form the rth column of the MTTKRP. Although SpMV is a well-investigated operation, there is a peculiarity here: the result of the first SpMV forms the values of the sparse matrix used in the second one. Therefore, there are sophisticated data dependencies between the two SpMVs. Notice that DFacTo is rich in SpMV operations: there are two SpMVs per factorization rank per dimension of the input tensor. DFacTo needs to store the tensor matricized in all dimensions, i.e., $\mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(N)}$. In low dimensions, this can be a slight memory overhead; yet in higher dimensions the overhead could be non-negligible. DFacTo uses MPI for parallelization yet fully store the factor matrices in all MPI ranks. The rows of $\mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(N)}$ are blockwise distributed (statically). With this partition, each process computes the corresponding rows of the factor matrices. Finally, DFacTo performs an MPI_Allgatherv operation to communicate the new results to all processes, which results in $(I_n/P)\log_2 P$ communication volume per process (assuming a hypercube algorithm) when computing the nth factor matrix having I_n rows using P processes. Experiments are presented on machines equipped with two 2.1 GHz 12-core AMD 6172 processors where up to 32 machines are used. In sequential runs, DFacTo is shown to be 5 times faster than GigaTensor and 10 times faster than a MATLAB implementation [5]. On a real world data set, DFacTo obtains about 3.5 speedup on 32 machines with respect to an execution on four machines.

Tensor Toolbox [6] is a MATLAB toolbox for handling tensors. It provides many essential operations and enables fast and efficient realizations of complex algorithms in MATLAB for sparse tensors [5]. Among those operations, MTTKRP implementations are provided and used in CP-ALS method. Here, each column of the output is computed by performing $N-1$ sparse tensor vector multiplications. Another well-known MATLAB toolbox is the N-way toolbox [2] which is essentially for dense tensors and incorporates now support for sparse tensors [1] through Tensor Toolbox. Tensor Toolbox and the related software provide excellent means for rapid prototyping of algorithms and also efficient programs for tensor operations that can be handled within MATLAB.

2.3 Hypergraphs and hypergraph partitioning

A hypergraph $H = (V, E)$ is defined as a set of vertices V and a set of hyperedges E. Each hyperedge is a set of vertices. The vertices of a hypergraph can be associated with weights denoted by $\mathsf{w}[\cdot]$, and the hyperedges can be associated with costs denoted by $\mathsf{c}[\cdot]$. For a given integer $K \geq 2$, a K-way vertex partition of a hypergraph $H = (V, E)$ is denoted as $\Pi = \{V_1, \ldots, V_K\}$, where the parts are non-empty, mutually exclusive, $V_k \cap V_\ell = \emptyset$ for $k \neq \ell$; and collectively exhaustive, $V = \bigcup V_k$.

Let $W_k = \sum_{v \in V_k}$ be the total weight in V_k and $W_{avg} = \sum_{v \in V} \mathsf{w}[v]/K$ be the average part weight. If each part $V_k \in \Pi$ satisfies the *balance criterion*

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{1}$$

we say that Π is *balanced* where ε represents the maximum allowed imbalance ratio.

In a partition Π, a hyperedge that has at least one vertex in a part is said to *connect* that part. The number of parts connected by a hyperedge h, i.e., *connectivity*, is denoted as λ_h. Given a vertex partition Π of a hypergraph $H = (V, E)$, one can measure the size of the cut induced by Π as

$$\chi(\Pi) = \sum_{h \in E} \mathsf{c}[h](\lambda_h - 1) \ . \tag{2}$$

This cut measure is called the *connectivity-1* cutsize metric.

Given $\varepsilon > 0$ and an integer $K > 1$, the standard hypergraph partitioning problem is defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [27].

A recent variant of the above problem is the *multi-constraint hypergraph partitioning* [13, 24]. In this variant, each vertex

has an associated vector of weights. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint for each weight. Let $\mathsf{w}[v,i]$ denote the C weights of a vertex v for $i = 1,\ldots,C$. In this variant, the balance criterion (1) is rewritten as

$$W_{k,i} \leq W_{avg,i}\,(1+\varepsilon) \text{ for } k = 1,\ldots,K \text{ and } i = 1,\ldots,C \quad (3)$$

where the ith weight $W_{k,i}$ of a part V_k is defined as the sum of the ith weights of the vertices in that part (i.e., $W_{k,i} = \sum_{v\in V_k} \mathsf{w}[v,i]$), $W_{avg,i}$ is the average part weight for the ith weight of all vertices (i.e., $W_{avg,i} = \sum_{v\in V} \mathsf{w}[v,i]/K$), and ε again represents the allowed imbalance ratio.

3. PARALLELIZATION

A common approach in implementing the MTTKRP is to explicitly matricize a tensor across all modes, and then perform the Khatri-Rao product using the matricized tensors [14, 31]. Matricizing a tensor in a mode i requires column index values up to $\prod_{k\neq i}^{N} I_k$, which can exceed the integer value limits supported by modern architectures when using tensors of higher order and very large dimensions. Also, matricizing across all modes results in N replications of a tensor; which can exceed the memory limitations. Hence, in order to be able to handle large tensors we store them in coordinate format for the MTTKRP operation, which is also the method of choice in Tensor Toolbox [6].

With a tensor stored in the coordinate format, MTTKRP operation can be performed as shown in Algorithm 2. As seen on Line 1 of this algorithm, a row of $\mathbf{B}$ and a row of $\mathbf{C}$ are retrieved, and their Hadamard product is computed and scaled with a tensor entry to update a row of $\mathbf{M}_A$. In general, for an N-mode tensor

$$\mathbf{M}_{U_1}(i_1,:) \leftarrow \mathbf{M}_{U_1}(i_1,:) + x_{i_1,i_2\ldots,i_N}[\mathbf{U}_2(i_2,:)*\cdots*\mathbf{U}_N(i_N,:)]$$

is computed. Here, indices of the corresponding rows of the factor matrices and $\mathbf{M}_{U_1}$ coincide with indices of the unique tensor entry of the operation.

Algorithm 2: MTTKRP for the 3rd order tensors

Input : $\mathcal{X}$: tensor
$\mathbf{B},\mathbf{C}$: Factor matrices in all modes except the first
I_A: Number of rows of the factor $\mathbf{A}$
R: Rank of the factors
Output: $\mathbf{M}_A = \mathbf{X}_{(1)}(\mathbf{B}\odot\mathbf{C})$

Initialize $\mathbf{M}_A$ to zeros of size $I_A \times R$
foreach $x_{i,j,k} \in \mathcal{X}$ **do**
1 $\quad\mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$

As factor matrices are accessed row-wise, we define computational units in terms of the rows of factor matrices. It follows naturally to partition all factor matrices row-wise and use the same partition for the MTTKRP operation for each mode of an input tensor across all CP-ALS iterations to prevent extra communication. A crucial issue is the task definitions, as this pertains to the issues of load balancing and communication. We identify a *coarse-grain* and a *fine-grain* task definition for this computational kernel.

In the *coarse-grain* task definition, ith atomic task consists of computing the row $\mathbf{M}_A(i,:)$ using the nonzeros in the tensor slice $\mathcal{X}_{i,:,:}$ and the rows of $\mathbf{B}$ and $\mathbf{C}$ corresponding to the nonzeros in that slice. The input tensor $\mathcal{X}$ does not change throughout the iterations of tensor decomposition algorithms; hence it is viable to make the whole slice $\mathcal{X}_{i,:,:}$ available to the process holding $\mathbf{M}_A(i,:)$ so that the MTTKRP operation can be performed by only communicating the rows of $\mathbf{B}$ and $\mathbf{C}$. Yet, as CP-ALS requires the MTTKRP in all modes, and each nonzero $x_{i,j,k}$ belongs to slices $\mathcal{X}(i,:,:)$, $\mathcal{X}(:,j,:)$, and $\mathcal{X}(:,:,k)$, we need to replicate tensor entries in the owner processes of these slices. This may require up to N times replication of the tensor, depending on its partitioning. Note that an explicit matricization always requires exactly N replications of tensor entries.

In the *fine-grain* task definition, an atomic task corresponds to the multiplication of a tensor entry with the Hadamard product of the corresponding rows of $\mathbf{B}$ and $\mathbf{C}$. Here, tensor nonzeros are *partitioned* among processes with no replication to induce a task partition by following the owner-computes rule. This necessitates communicating the rows of $\mathbf{B}$ and $\mathbf{C}$ that are needed by these atomic tasks. Furthermore, partial results on the rows of $\mathbf{M}_A$ need to be communicated, as without duplicating tensor entries, we cannot in general compute all contributions to a row of $\mathbf{M}_A$. Here, the partition of $\mathcal{X}$ should be useful in all modes, as the CP-ALS method requires the MTTKRP in all modes.

The coarse-grain task definition resembles to the one-dimensional (1D) row-wise (or column-wise) partitioning of sparse matrices, whereas the fine-grain one resembles to the two-dimensional (nonzero-based) partitioning of sparse matrices for parallel sparse matrix-vector multiply (SpMV) operations. As is confirmed for SpMV in modern applications, 1D partitioning usually leads to harder problems of load balancing and communication cost reduction. The same phenomenon is likely to be observed in tensors as well. Nonetheless, we cover the coarse-grain task definition, as it is used in the state of the art parallel MTTKRP [14, 31] methods, which partition the matricized tensor row-wise (or equivalently, partition the input tensor by slices).

3.1 Coarse-grain task model

In the coarse-grain task model, computing the rows of $\mathbf{M}_A$, $\mathbf{M}_B$, and $\mathbf{M}_C$ are defined as the *atomic tasks* which are partitioned across all processes. Let μ_A denote the partition of the first mode's indices among the processes, i.e., $\mu_A(i) = p$, if the process p is responsible for computing $\mathbf{M}_A(i,:)$. Similarly, let μ_B and μ_C define the partition of the second and the third mode indices. The process owning $\mathbf{M}_A(i,:)$ needs the entire tensor slice $\mathcal{X}_{i,:,:}$; similarly, the process owning $\mathbf{M}_B(j)$ needs $\mathcal{X}_{:,j,:}$ and the owner of $\mathbf{M}_C(k,:)$ needs $\mathcal{X}_{:,:,k}$. This necessitates duplication of some tensor nonzeros to prevent unnecessary communication.

One needs to take the context of CP-ALS into account when parallelizing the MTTKRP method. First, the output $\mathbf{M}_A$ of MTTKRP is transformed into $\mathbf{A}$. Since $\mathbf{A}(i,:)$ is computed simply by multiplying $\mathbf{M}_A(i,:)$ with the matrix $(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$, we make the process which owns $\mathbf{M}_A(i,:)$ responsible for computing $\mathbf{A}(i,:)$. Second, N MTTKRP operations follow one another in an iteration. Assuming that every process has the required rows of the factor matrices while executing MTTKRP for the first mode, it is advisable to implement the MTTKRP in such a way that its output $\mathbf{M}_A$, after transformed into $\mathbf{A}$, is communicated. This way, all processes would have the necessary data for executing the MTTKRP for the next mode. With these in mind, the coarse-grain parallel MTTKRP method executes Algorithm 3 at each process p.

Algorithm 3: Coarse-grain MTTKRP for the first mode of 3rd order tensors at process p within CP-ALS

Input : I_p, indices where $\mu_A(i) = p$
$\mathcal{X}_{I_p,:,:}$, tensor slices
$\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$, $R \times R$ matrices
$\mathbf{B}(J_p,:), \mathbf{C}(K_p,:)$, Rows of the factor matrices, where
J_p and K_p correspond to the unique second and third mode indices in $\mathcal{X}_{I_p,:,:}$

On exit : $\mathbf{A}(I_p,:)$ is computed, its rows are sent to processes needing them; $\mathbf{A}^T\mathbf{A}$ is available

Initialize $\mathbf{M}_A(I_p,:)$ to all zeros of size $|I_p| \times R$.
foreach $i \in I_p$ **do**
  foreach $x_{i,j,k} \in \mathcal{X}_{i,:,:}$ **do**
1     $\mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$

2 $\mathbf{A}(I_p,:) \leftarrow \mathbf{M}_A(I_p,:)(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
foreach $i \in I_p$ **do**
3   Send $\mathbf{A}(i,:)$ to all processes having nonzeros in $\mathcal{X}_{i,:,:}$.
Receive $\mathbf{A}(i,:)$ from $\mu_A(i)$ for each owned nonzero $x_{i,j,k}$.
4 Locally compute $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reduce the results to form $\mathbf{A}^T\mathbf{A}$.

As seen in Algorithm 3, the process p computes $\mathbf{M}_A(i,:)$ for all i with $\mu_A(i) = p$ on Line 1. This is possible without any communication, if we assume the preconditions that $\mathbf{B}^T\mathbf{B}$, $\mathbf{C}^T\mathbf{C}$, and the required rows of $\mathbf{B}$ and $\mathbf{C}$ are available at each process. We need to satisfy this precondition for the MTTKRP operations in the remaining modes. Once $\mathbf{M}_A(I_p,:)$ is computed, it is multiplied on Line 2 with $(\mathbf{B}^T\mathbf{B}*\mathbf{C}^T\mathbf{C})^\dagger$ to obtain $\mathbf{A}(I_p,:)$. Then, the process p sends the rows of $\mathbf{A}(I_p,:)$ to other processes who will need them (messages destined to the same process are combined into a single message), then receive the ones that it will need. More precisely, the process p sends $\mathbf{A}(i,:)$ to the process r, where $\mu_B(j) = r$ or $\mu_C(k) = r$ for a nonzero $x_{i,j,k} \in \mathcal{X}$, thereby satisfying the stated precondition for the remaining MTTKRP operations. Since computing $\mathbf{A}(i,:)$ necessitates that $\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$ are available at each process, we also need to compute $\mathbf{A}^T\mathbf{A}$ and make the result available to all processes. We perform this by computing local multiplications $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reducing the partial results on Line 4.

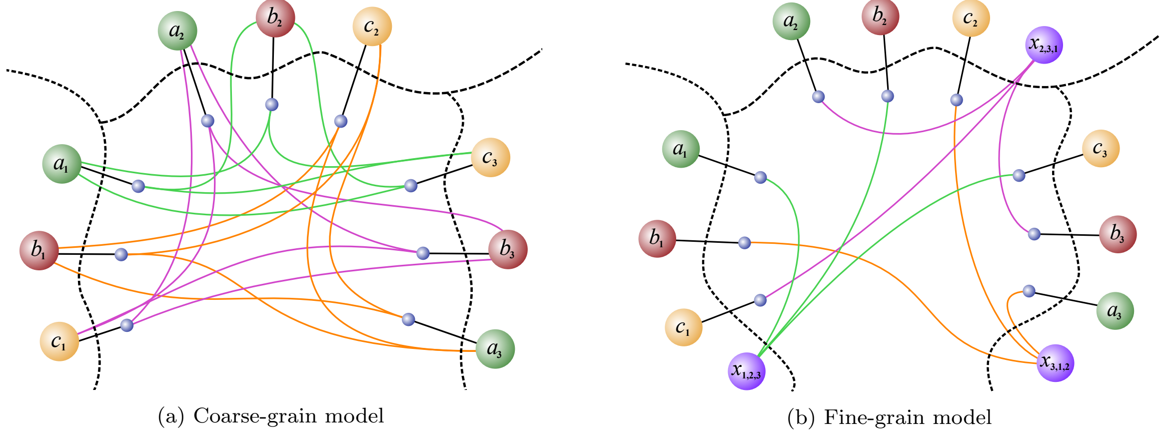
We now investigate the problems of obtaining load balance and reducing the communication cost in an iteration of CP-ALS given in Algorithm 3; that is, when Algorithm 3 is applied N times, once for each mode. Line 1 is the computational core; the process p performs $\sum_{i\in I_p} |\mathcal{X}_{i,:,:}|$ many Hadamard products (of vectors of size R) without any communication. In order to achieve load balance, one should partition the slices of the first mode equitably by taking the number of nonzeros into account. Line 2 does not involve communication, where load balance can be achieved by assigning almost equal number of slices to the processes. Line 4 requires almost equal number of slices to the processes for load balance. The operations at Lines 2 and 4 can be performed very efficiently using BLAS3 routines; therefore, their costs are generally negligible in compare to the cost of the sparse irregular computations at Line 1. There are two lines, 3 and 4, requiring communication. Line 4 requires the collective all-reduce communication, and one can rely on efficient MPI implementations. The communication at Line 3 is irregular and would be costly. Therefore, the problems of computational load balance and communication cost need to be addressed with regards to Lines 1 and 3, respectively.

Let a_i be the task of computing $\mathbf{M}_A(i,:)$ at Line 1. Let also T_A be the set of tasks a_i for $i = 1, \ldots, I_A$, and b_j, c_k, T_B, and T_C be defined similarly. In the CP-ALS algorithm, there are dependencies among the triplets a_i, b_j, and c_k of the tasks for each nonzero $x_{i,j,k}$. Specifically, the task a_i depends on the tasks b_j and c_k. Similarly, b_j depends on the tasks a_i and c_k, and c_k depends on the tasks a_i and b_j. These dependencies are the source of the communication at Line 3; e.g., $\mathbf{A}(i,:)$ should be sent to the processes that own $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ for the each nonzero $x_{i,j,k}$ in the slice $\mathcal{X}_{i,:,:}$. These dependencies can be modeled using graphs or hypergraphs by identifying the tasks with vertices, and their dependencies with edges and hyperedges, respectively. As is well-known in similar parallelization contexts [11, 21, 22], the standard graph partition related metric of "edge cut" would loosely relate to the communication cost. We therefore propose a hypergraph model for modeling the communication and computational requirements of the MTTKRP operation in the context of CP-ALS.

For a given tensor $\mathcal{X}$, we build a hypergraph $H = (V, E)$ with the vertex set V and hyperedge set E as follows. The vertex set V corresponds to the set of tasks. We abuse the notation and use a_i to refer to a task and its corresponding vertex in V; the meaning should be clear from the context. We can then set $V = T_A \cup T_B \cup T_C$. Since one needs load balance on Line 1 for each mode, we use vectors of size N for vertex weights. We set $\mathsf{w}[a_i, 1] = |\mathcal{X}_{i,:,:}|$, $\mathsf{w}[b_j, 2] = |\mathcal{X}_{:,j,:}|$, and $\mathsf{w}[c_k, 3] = |\mathcal{X}_{:,:,k}|$ as the vertex weights in the indicated modes, and assign weight 0 in all other modes. The dependencies causing communication at Line 3 are one-to-many: for a nonzero $x_{i,j,k}$, any one of a_i, b_j, and c_k is computed by using the data associated with the other two. Following the guidelines in building the hypergraph models [33], we add a hyperedge corresponding to each task (or each row of the factor matrices) in $T_A \cup T_B \cup T_C$, in order to model the data dependencies to that specific task. We use n_i^a to denote the hyperedge corresponding to the task a_i, and define n_i^b and n_k^c similarly. For each nonzero $x_{i,j,k}$, we add the vertices a_i, b_j, and c_k to the hyperedges n_i^a, n_j^b, and n_k^c. Let J_i and K_i be all unique 2nd and 3rd mode indices, respectively, of the nonzeros in $\mathcal{X}_{i,:,:}$. Then, the hyperedge n_i^a contains the vertex a_i, all vertices b_j for $j \in J_i$, and all vertices c_k for $k \in K_i$.

Figure 1a demonstrates the hypergraph model for the coarse-grain task decomposition of a sample $3 \times 3 \times 3$ tensor with nonzeros $(1, 2, 3)$, $(2, 3, 1)$, and $(3, 1, 2)$. Labels for the hyperedges, shown as small blue circles, are not provided; yet it should be clear from the context, e.g., $a_i \in n_i^a$ is shown with a non-curved black line between the vertex and the hyperedge. For the first, second, and third nonzeros, we add the corresponding vertices to related hyperedges, which are shown with green, magenta, and orange curves, respectively. It is interesting to note that the N-partite graph of Smith et al. [31] and this hypergraph model are related; their adjacency structure when laid as a matrix give the same matrix.

Consider now a P-way partition of the vertices of $H = (V, E)$ and the association of each part with a unique process for obtaining a P-way parallel CP-ALS. The task a_i incurs $|\mathcal{X}_{i,:,:}|$ Hadamard products in the first mode in the



(a) Coarse-grain model

(b) Fine-grain model

Figure 1: Coarse and fine-grain hypergraph models for the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3), (2,3,1), (3,1,2)\}$.

process $\mu_A(i)$, and has no other work when computing the MTTKRPs $\mathbf{M}_B = \mathbf{X}_{(2)}(\mathbf{A} \odot \mathbf{C})$ and $\mathbf{M}_C = \mathbf{X}_{(3)}(\mathbf{A} \odot \mathbf{B})$ in the other modes. This observation (combined with the corresponding one for b_js and c_ks) shows that any partition satisfying the balance condition for each weight component (3) corresponds to a balanced partition of Hadamard products (Line 1) in each mode. Now consider the messages, which are of size R, sent by the process p regarding $\mathbf{A}(i,:)$. These messages are needed by the processes that have slices in modes 2 and 3 that have nonzeros whose first mode index is i. These processes correspond to the parts connected by the hyperedge n_i^a. For example, in Figure 1*a* due to the nonzero $(3,1,2)$, a_3 has a dependency to b_1 and c_2, which are modeled by the curves from a_3 to n_1^b and n_2^c. Since a_3, b_1, and c_2 reside in different parts, a_3 contributes one to the connectivity of n_1^b and n_2^c, which accurately captures the total communication volume by increasing the total connectivity-1 metric by two.

Notice that the part $\mu_A(i)$ is always connected by n_i^a—assuming that the slice $\mathcal{X}_{i,:,:}$ is not empty. Therefore, minimizing the connectivity-1 metric (2) *exactly* amounts to minimizing the total communication volume required for MTTKRP in all computational phases of a CP-ALS iteration, if we set $\mathsf{c}[\cdot] = R$ for all hyperedges.

In the gradient-descent based algorithm CP-OPT [1], all factors are updated simultaneously based on their values in the previous iteration; hence, one can use the same hypergraph with a slight modification in vertex weights. In this case, we need to use a single-constraint hypergraph partitioning by setting $\mathsf{w}[a_i] = |\mathcal{X}_{i,:,:}|$, $\mathsf{w}[b_j] = |\mathcal{X}_{:,j,:}|$, and $\mathsf{w}[c_k] = |\mathcal{X}_{:,:,k}|$ to induce load balance by partitioning the hypergraph.

3.2 Fine-grain task model

In the fine-grain task model we assume a *partition* of the nonzeros of the tensor; hence there is no duplication. Let $\mu_{\mathcal{X}}$ denote the partition of the tensor nonzeros, e.g., $\mu_{\mathcal{X}}(i,j,k) = p$ if the process p holds $x_{i,j,k}$. Whenever $\mu_{\mathcal{X}}(i,j,k) = p$, the process p performs the Hadamard product of $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ in the first mode, $\mathbf{A}(i,:)$ and $\mathbf{C}(k,:)$ in the second mode, and $\mathbf{A}(i,:)$ and $\mathbf{B}(j,:)$ in the third mode; then scales the result with $x_{i,j,k}$. Let again μ_A, μ_B, and μ_C denote the partition on the indices of the corresponding modes; that is, $\mu_A(i) = p$ denotes that the process p is responsible for computing $\mathbf{M}_A(i,:)$. As in the coarse-grain model, we take the CP-ALS context into account while designing the MTTKRP algorithm: (i) the process which is responsible for $\mathbf{M}_A(i,:)$ is also held responsible for $\mathbf{A}(i,:)$; (ii) at the beginning, all rows of the factor matrices $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ where $\mu_{\mathcal{X}}(i,j,k) = p$ are available to the process p; (iii) at the end, all processes get the $R \times R$ matrix $\mathbf{A}^T\mathbf{A}$; and (iv) each process has all the rows of $\mathbf{A}$ that are required for the upcoming MTTKRP operations in the second or the third modes. With these in mind, the fine-grain parallel MTTKRP method executes Algorithm 4 at each process p.

Algorithm 4: Fine-grain MTTKRP for the first mode of 3rd order tensors at process p within CP-ALS

Input : $\mathcal{X}_p$, tensor nonzeros where $\mu_{\mathcal{X}}(i,j,k) = p$
I_p, the first mode indices where $\mu_A(I_p) = p$
F_p, the set of unique first mode indices in $\mathcal{X}_p$
$\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$: $R \times R$ matrices
$\mathbf{B}(J_p,:), \mathbf{C}(K_p,:)$: Rows of the factor matrices,
where
J_p and K_p correspond to the unique second and third mode indices in $\mathcal{X}_p$

On exit : $\mathbf{A}(I_p,:)$ is computed, its rows are sent to processes needing them; $\mathbf{A}(F_p,:)$ is updated; and $\mathbf{A}^T\mathbf{A}$ is available

Initialize $\mathbf{M}_A(F_p,:)$ to all zeros of size $|F_p| \times R$
foreach $x_{i,j,k} \in \mathcal{X}_p$ **do**
1 $\quad \mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$
foreach $i \in F_p \setminus I_p$ **do**
2 $\quad$ Send $\mathbf{M}_A(i,:)$ to $\mu_A(i)$
3 Receive contributions to $\mathbf{M}_A(i,:)$ for $i \in I_p$ and add them up
4 $\mathbf{A}(I_p,:) \leftarrow \mathbf{M}_A(I_p,:)(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
foreach $i \in I_p$ **do**
5 $\quad$ Send $\mathbf{A}(i,:)$ to all processes having nonzeros in $\mathcal{X}_{i,:,:}$.
foreach $i \in F_p \setminus I_p$ **do**
6 $\quad$ Receive $\mathbf{A}(i,:)$ from $\mu_A(i)$
7 Locally compute $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reduce the results to form $\mathbf{A}^T\mathbf{A}$

In Algorithm 4, $\mathcal{X}_p$ and I_p denote the set of nonzeros and the set of first mode indices assigned to the process p. As seen in the algorithm, the process p has the required data

to carry out all Hadamard products corresponding to all $x_{i,j,k} \in \mathcal{X}_p$ on Line 1. After scaling by $x_{i,j,k}$, the Hadamard products are locally accumulated in $\mathbf{M}_A$, which contains a row for all i where $\mathcal{X}_p$ has a nonzero on the slice $\mathcal{X}_{i,:,:}$. Notice that a process generates a partial result for each slice of the first mode in which it has a nonzero. The relevant row indices (the set of unique first mode indices in $\mathcal{X}_p$) are denoted by F_p. Some indices in F_p are not owned by the process p. For such an $i \in F_p \setminus I_p$, the process p sends its contribution to $\mathbf{M}_A(i,;)$ to the owner process $\mu_A(i)$ at Line 2. Again, messages of the process p destined to the same process are combined. Then, the process p receives contributions to $\mathbf{M}_A(i,:)$ where $\mu_A(i) = p$ at Line 3. Each such contribution is accumulated, and the new $\mathbf{A}(i,:)$ is computed by a multiplication with $(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$ at Line 4. Finally, the updated $\mathbf{A}$ is communicated to satisfy the precondition of the upcoming MTTKRP operations. Note that the communications at Lines 5 and 6 are the duals of those at Lines 3 and 2, respectively, in the sense that the directions of the messages are reversed and the messages always pertain to the same row indices. For example, $\mathbf{M}_A(i,:)$ is sent to the process $\mu_A(i)$ at Line 2 and $\mathbf{A}(i,:)$ is received from $\mu_A(i)$ at Line 6. Hence, the process p generates partial results for each slice $\mathcal{X}_{i,:,:}$ on which it has a nonzero; either keeps it or sends it to the owner of $\mu_A(i)$ (at Line 2), and if so, receives the updated row $\mathbf{A}(i,:)$ later (at Line 6). Finally, the algorithm finishes by an all-reduce operation to make $\mathbf{A}^T\mathbf{A}$ available to all processes for the upcoming MTTKRP operations.

We now investigate the computational and communication requirements of Algorithm 4. As before, the computational core is at Line 1, where the Hadamard products of the row vectors of $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ are computed for each nonzero $x_{i,j,k}$ that a process owns, and the result is added to $\mathbf{M}_A(i,:)$. Therefore, each nonzero $x_{i,j,k}$ incurs three Hadamard products (one per mode) in its owner process $\mu_{\mathcal{X}}(i,j,k)$ in an iteration of the CP-ALS algorithm. Other computations are either efficiently handled with BLAS3 routines (Lines 4 and 7), or they (Line 3) are not as many as the number of Hadamard products. The significant communication operations are the sends at Lines 2 and 5, and the corresponding receives at Lines 3 and 6. Again, the all-reduce operation at Line 7 would be handled efficiently by the MPI implementation. Therefore, the problems of computational load balance and reduced communication cost need to be addressed by partitioning the tensor nonzeros equitably among processes (for load balance at Line 1) while trying to confine all slices of all modes to a small set of processes to reduce the communication (to reduce communication at Lines 2 and 6).

We propose a hypergraph model for the computational load and communication volume of the fine-grain parallel MTTKRP operation in the context of CP-ALS. For a given tensor, the hypergraph $H = (V, E)$ with the vertex set V and hyperedge set E is defined as follows. Since we need to partition the nonzeros and the indices along each mode, V has four types of vertices: a_i, for $i = 1, \ldots, I_A$; b_j, for $j = 1, \ldots, I_B$; c_k for $k = 1, \ldots, I_C$; and the vertex $x_{i,j,k}$ we define for each nonzero in $\mathcal{X}$ by abusing the notation. The vertex $x_{i,j,k}$ represents the Hadamard products $\mathbf{B}(j,:)*\mathbf{C}(k,:)$, $\mathbf{A}(i,:)*\mathbf{C}(k,:)$, and $\mathbf{A}(i,:)*\mathbf{B}(j,:)$ to be performed during the MTTKRP operations at different modes. There are three types of hyperedges in E, and they represent the dependencies of the Hadamard products to the rows of the factor matrices. The hyperedges are defined as follows: $E = E_A \cup E_B \cup E_C$ where E_A contains a hyperedge n_i^a for each $\mathbf{A}(i,:)$; E_B contains a hyperedge n_j^b for each $\mathbf{B}(j,:)$; and E_C contains a hyperedge n_k^c for each $\mathbf{C}(k,:)$. Initially, n_i^a, n_j^b and n_k^c contain only the corresponding vertices a_i, b_j, and c_k. Then, for each nonzero $x_{i,j,k} \in \mathcal{X}$, we add the vertex $x_{i,j,k}$ to the hyperedges n_i^a, n_j^b and n_k^c to model the data dependency to the corresponding rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. As in the previous subsection, one needs a multi-constraint formulation for three reasons. First, since $x_{i,j,k}$ vertices represent the Hadamard products, they should be evenly distributed among the processes. Second, as the owners of the rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are possible destinations of partial results (at Line 2), it is advisable to spread them evenly. Third, almost equal number of rows per process implies balanced memory usage and BLAS3 operations at Lines 4 and 7; even though this operation is not our main concern for the computational load. With these constraints, we associate a weight vector of size $N+1$ with each vertex. For a vertex $x_{i,j,k}$ we set $\mathsf{w}[x_{i,j,k}] = [0,0,0,3]$, for vertices a_i, b_j and c_k, we set $\mathsf{w}[a_i] = [1,0,0,0]$, $\mathsf{w}[b_j] = [0,1,0,0]$, $\mathsf{w}[c_k] = [0,0,1,0]$.

In Figure 1b, we demonstrate the fine-grain hypergraph model for the sample tensor of the previous section. We exclude the vertex weights from the figure. Similar to the coarse-grain model, dependency to the rows of $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are modeled by connecting each vertex $x_{i,j,k}$ to the hyperedges n_i^a, n_j^b, and n_k^c to capture the communication cost of the parallel fine-grain MTTKRP.

Consider now a P-way partition of the vertices of $H = (V, E)$ and associate each part with a unique process to obtain P-way parallel CP-ALS with the fine-grain MTTKRP operation. The partition of the $x_{i,j,k}$ vertices uniquely partition the Hadamard products. Satisfying the fourth balance constraint, as in (3), balances the computational loads of the processes. Similarly, satisfying the first three constraints leads to an equitable partition of the rows of factor matrices among processes. Let $\mu_{\mathcal{X}}(i,j,k) = p$ and $\mu_A(i) = \ell$; that is, the vertex $x_{i,j,k}$ and a_i are in different parts. Then, the process p sends a partial result on $\mathbf{M}_A(i,:)$ to the process ℓ at Line 2 of Algorithm 4. By looking at these messages carefully, we see that all processes whose corresponding parts are in the connectivity set of n_A^i will send a message to $\mu_A(i)$. Since $\mu_A(i)$ is also in this set, we see that the total volume of send operations concerning $\mathbf{M}_A(i,:)$ at Line 2 is equal to $\lambda_{n_i^a} - 1$. Therefore, the connectivity-1 cut size metric (2) over the hyperedges in E_A encodes the total volume of messages sent at Line 2, if we set $\mathsf{c}[\cdot] = R$. Since the send operations at Line 5 are duals of the send operations at Line 2, total volume of messages sent at Line 5 for the first mode is also equal to this number. By extending this reasoning to all hyperedges, we see that the cumulative (over all modes) volume of communication is equal to the connectivity-1 cut-size metric (2).

The presence of multiple constraints usually causes difficulties for the current state of the art partitioning tools (Aykanat et al.[4] discuss the issue for PaToH [12]). In preliminary experiments, we observed that this was the case for us as well. In an attempt to circumvent this issue, we designed the following alternative. We start with the same hypergraph, and remove the vertices corresponding to the rows of factor matrices. Then use a single constraint partitioning on the resulting hypergraph to partition the vertices

of nonzeros. We are then faced with the problem of assigning the rows of factor matrices, given the partition on the tensor nonzeros. Any objective function (relevant to our parallelization context) in trying to solve this problem would likely be a variant of the NP-complete Bin-Packing Problem [17, p.124], as in the SpMV case [8, 32]. Therefore, we heuristically handle this problem, by making the following observations: (i) the modes can be partitioned one at a time; (ii) each row corresponds to a removed vertex for which there is a corresponding hyperedge; (iii) the partition of $x_{i,j,k}$ vertices define a connectivity set for each hyperedge. This is essentially multi-constraint partitioning which takes advantage of the listed observations. We use connectivity of corresponding hyperedges as *key value* of rows to impose an order. Then, rows are visited in decreasing order of key values. Each visited row is assigned to the least loaded part in the connectivity set of its corresponding hyperedge, if that part is not overloaded. Otherwise, the visited vertex is assigned to the least loaded part at that moment. Note that in the second case we add one to the total communication volume; otherwise the total communication volume obtained by partitioning $x_{i,j,k}$ vertices remains the same. Note also that key values correspond to the volume of receives at Line 3 of Algorithm 4; hence, this method also aims to balance the volume of receives at Line 3, and the volume of dual send operations at Line 5.

3.3 Discussion

The pros and cons of the coarse and fine-grain models resemble to those of the 1D and 2D partitionings in the SpMV context. There are two advantages of the coarse-grain model over the fine-grain one. First, there is only one communication/computation step in the coarse-grain model; whereas fine-grain model requires two computation and communication phases. Second, the coarse-grain hypergraph model is smaller than the fine-grain hypergraph model (one vertex per index of each dimension versus additionally having one vertex per nonzero). However, one major limitation with the coarse-grain parallel decomposition is that restricting all nonzeros in an $(N-1)$ dimensional slice of an N dimensional tensor to reside in the same process poses a significant constraint towards communication minimization; an $(N-1)$-dimensional data typically has very large "surface" which necessitates gathering numerous rows of the factor matrices. As N increases, one might expect this phenomenon to increase the communication volume even further. Also, if the tensor $\mathfrak{X}$ is not large enough in one of its modes, this poses a granularity problem which potentially causes load imbalance and limits the parallelism. None of these issues exists in the fine-grain task decomposition.

4. EXPERIMENTS

We conducted our experiments on the IDRIS Ada cluster, which consists of 304 nodes each having 128 GBs of memory and four 8-core Intel Sandy Bridge E5-4650 processors running at 2.7 GHz. We ran our experiments up to 1024 cores, and assigned one MPI process per core. The cluster allowed runs up to 2048 cores; but none of the methods we experimented with scaled to that point. All codes we used in our benchmarks were compiled using the Intel C++ compiler (version 14.0.1.106) with -O3 option for compiler optimizations, -axAVX,SSE4.2 to enable SSE optimizations whenever possible, and -mkl option to use the Intel MKL library (version 11.1.1) for LAPACK, BLAS and VML (Vector Math Library) routines. We also compiled DFacTo code using the Eigen (version 3.2.4) library [19] with EIGEN_USE_MKL_ALL flag set to ensure that it utilizes the routines provided in the Intel MKL library for the best performance. We used PaToH [12] (version 3.2) with default options to partition the hypergraphs that we formed. We created all partitions offline, and ran our experiments on these partitioned tensors on the cluster. We do not report timings for partitioning hypergraphs, which is quite costly, for two reasons. First, in most applications (see the provenance of our data set below), the tensors from the real-world data are built incrementally; hence, a partition for the updated tensor can be formed by refining the partition of the previous tensor. Also, one can decompose a tensor multiple times with different ranks of approximation. Thereby, the cost of an initial partitioning of a tensor can be amortized across multiple runs. Second, we had to partition the data on a system different from the one used for CP-ALS runs. It would not be very useful to compare the runtimes from different systems and repeating the same runs on different system would not add much to the presented results. We also note that the proposed coarse and fine-grain MTTKRP formulations are independent from the partitioning method.

Table 1: Size of tensors used in the experiments

Tensor	I_1	I_2	I_3	I_4	#nonzeros
Netflix	480K	17K	2K	-	100M
NELL-B	3.2M	301	638K	-	78M
Delicious	530K	17M	2.4M	1K	140M
Flickr	319K	28M	1.6M	713	112M

Our dataset for experiments consists of four sparse tensors, whose sizes are given in Table 1. The first tensor has user × movie × time dimensions, and is formed from the Netflix Prize competition [7]. In this tensor, the nonzeros correspond to the user reviews of movies, and review date extends the data to the third dimension. The values of the nonzeros are determined by the corresponding review scores given by the users. The tensor NELL-B comes from the Never Ending Language Learning (NELL) knowledge database of the "Read the Web" project [9], which consists of tuples of the form $(entity, relation, entity)$ such as ('Chopin','plays musical instrument','piano'). The nonzeros of this tensor correspond to these entries discovered by NELL from the web, and the values are set to be the "belief" scores given by the learning algorithms used in NELL. Delicious and Flickr are the datasets for the web-crawl of Delicious.com and Flickr.com during 2006 and 2007, which is formed by Görlitz et al. [18]. These datasets consist of tuples of the form (users × resources × tags × time); hence we naturally form 4-mode tensors out of these tuples.

We now briefly describe the methods compared in the experiments. **DFacTo** is the CP-ALS routine we compare our methods against, which uses block partitioning of the rows of the matricized tensors to distribute matrix nonzeros equally among processes. In the method **ht-coarsegrain-block**, we operate the coarse-grain CP-ALS implementation in HyperTensor on a tensor whose slices in each mode are partitioned consecutively to ensure equal number of nonzeros among processes, similarly to the DFacTo's approach. The method **ht-coarsegrain-hp** runs the same computational kernel, but it uses the hypergraph partitioning of the tensor

for the coarse-grain task definition given in Section 3.1. The method **ht-finegrain-random** partitions the tensor nonzeros as well as the rows of the factor matrices randomly to establish load balance. It uses the fine-grain CP-ALS implementation in HyperTensor. Finally, the method **ht-finegrain-hp** corresponds to the fine-grain CP-ALS implementation in HyperTensor operating on the tensor partitioned according to the hypergraph model proposed for the fine-grain task decomposition in Section 3.2. We benchmark our methods against DFacTo on Netflix and NELL-B datasets only; as DFacTo can only process 3-mode tensors. Hence, for Flickr and Delicious tensors we only compare our methods using different partitioning schemes. We let the CP-ALS implementations run for 20 iterations on each data with $R = 10$, and record the average time spent per iteration.

In Figures 2a and 2b, we show the time spent per CP-ALS iteration on Netflix and NELL-B tensors for all algorithms. In Figures 2c and 2d, we show the speedups per CP-ALS iteration on Flickr and Delicious tensors for methods excluding DFacTo. While performing the comparison with DFacTo, we preferred to show the time spent per iteration instead of the speedup, as the sequential running time of our methods differs from that of DFacTo. Also in Table 2, we show the statistics regarding the communication and computation requirements of the 512-way partitions of the Netflix tensor for all proposed methods.

We first observe in Figure 2a that on the Netflix tensor ht-finegrain-hp clearly outperforms all other methods by achieving a speedup of 194x with 512 cores over a sequential execution, whereas ht-coarsegrain-hp, ht-coarsegrain-block, DFacTo, and ht-finegrain-random could only yield to 69x, 63x, 49x, and 40x speedups, respectively. Table 2 shows that on 512-way partitioning of the same tensor, ht-finegrain-random, ht-coarsegrain-block, and ht-coarsegrain-hp result in total send communication volumes of 142M, 80M, and 77M units respectively; whereas ht-finegrain-hp partitioning requires only 7.6M units; which explains the superior parallel performance of ht-finegrain-hp. Similarly in Figures 2b, 2c, and 2d ht-finegrain-hp achieves 81x, 129x, and 123x speedups on NELL-B, Flickr, and Delicious tensors, while the best of all other methods could only obtain 39x, 55x, and 65x, respectively, upto 1024 cores. As seen from these figures, ht-finegrain-hp is the fastest of all proposed methods. It experiences slow-downs at 1024 cores for all instances. Even though other methods could scale to 1024 cores in some instances, they still remained to be significantly slower than ht-finegrain-hp.

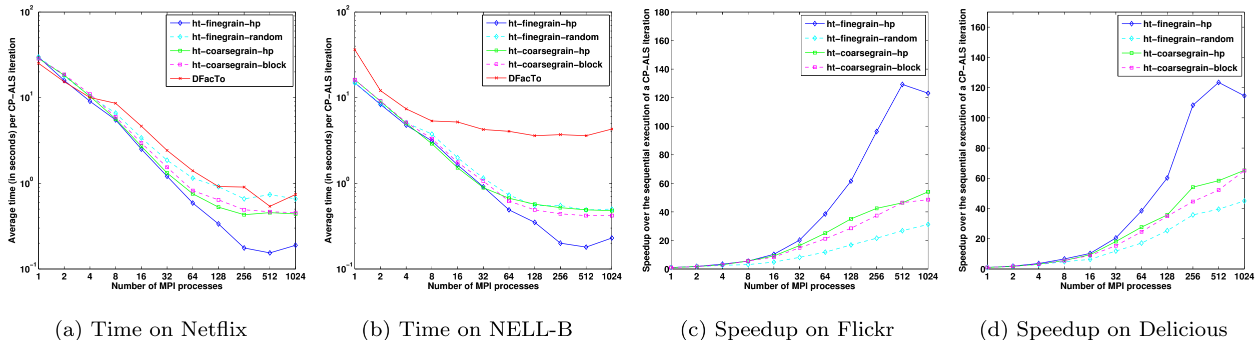
One point to note in Figure 2b is that our MTTKRP kernels get notably more efficient than DFacTo on NELL-B, despite that DFacTo uses optimized kernels from the Eigen and MKL libraries. We believe that the main reason for this difference is due to DFacTo's column-by-column way of computing the factor matrices in order to be able to use these column vectors in the SpMV kernels. We instead operate our sparse computations on the row vectors of the factor matrices, and compute all columns of the factor matrices simultaneously. This vector-wise mode of operation on the rows of the factors can significantly improve the data locality and cache efficiency which leads to this performance difference. Another point in the same figure is that HyperTensor running with ht-coarsegrain-block partitioning scales better than DFacTo, even though they use similar consecutive row or slice partitioning of the matricized tensor or the tensor. This is mainly due to the fact that DFacTo uses MPI_Allgatherv to communicate local rows of factor matrices to all processes, which incurs significant increase in the communication volume, whereas our coarse-grain computation kernel performs point-to-point communications only with processes that need these rows. Finally, in Figure 2a, DFacTo has a slight edge over our kernels in sequential execution. This could be due to the fact that it requires $3(|\mathcal{X}| + F)$ multiply/add operations across all modes, where F is the average number of non-empty fibers and is upper-bounded by $|\mathcal{X}|$, whereas our implementation always taks $6|\mathcal{X}|$ operations.

Partitioning metrics in the Table 2 show that ht-finegrain-hp is able to successfully reduce the total communication volume, which brings about its improved scalability. However, we do not see the same effect when using hypergraph partitioning for the coarse-grain task model, due to inherent limitations of 1D partitionings. Another point to note is that despite reducing the total communication volume explicitly, imbalance in the communication volume still exists among the processes, as this objective is not directly captured by the hypergraph models. However, the most important limitation on further scalability is the communication latency. The results in Table 2 show that each process communicates with almost all others especially on small dimensions of the tensors, and the number of messages is doubled for the fine-grain computation kernel—recall that it has two communication phases. To alleviate this issue, one can try to limit the latency following previous work [29, 32], and we plan to exploit this possibility in the future work.

Since DFacTo implements the MTTKRP operation in a series of parallel SpMV operations, one may consider using graph/hypergraph partitioning techniques to speed up the parallel execution. We did not investigate these alternatives for two reasons. First, our results on the parallel performance along with the statistical description of the ht-coarsegrain-block and ht-coarsegrain-hp partitions in Table 2 show that there is not much hope to obtain decent parallel performance from 1D partitioning of tensors (or row-wise partitioning of matricized tensors). Second, as stated in Section 2.2, there are sophisticated data dependencies between the successive SpMV operations in DFacTo, which requires further effort to apply the graph/hypergraph models.

5. CONCLUSION

We have investigated the efficient parallelization of the matricized tensor times Khatri-Rao product (MTTKRP) operation in the context of the alternating least squares (ALS) method for the CANDECOMP/PARAFAC (CP) decomposition. This operation is the computational core of the CP-ALS algorithm as well as many of its variants. We have taken a close look at the computational tasks and formulated a coarse and a fine-grain task definition. The coarse-grain task definition is based on a one-dimensional partitioning of the tensor. Such partitioning are inherently limited in scalability—the same limits as in the one-dimensional partitioning based SpMV operations. The fine-grain task definition is based on a nonzero partitioning of the tensor, and overcomes these limitations. We have identified the parallelization requirements of the coarse-grain and fine-grain MTTKRP methods, and presented hypergraph models to meet those needs. We have designed a library for implementing the CP-ALS algorithm and presented scalability results



(a) Time on Netflix (b) Time on NELL-B (c) Speedup on Flickr (d) Speedup on Delicious

Figure 2: Time for parallel CP-ALS iteration on Netflix and NELL-B, and speedups on Flickr and Delicious.

Table 2: Statistics for the computation and communication requirements in one CP-ALS iteration for 512-way partitionings of the Netflix tensor.

Mode	Comp. load		Comm. volume		Num. msg.	
	Max	Avg	Max	Avg	Max	Avg
ht-finegrain-hp						
1	196672	196251	21079	6367	734	316
2	196672	196251	18028	5899	1022	1016
3	196672	196251	3545	2492	1022	1018
ht-finegrain-random						
1	197507	196251	272326	252118	1022	1022
2	197507	196251	29282	22715	1022	1022
3	197507	196251	7766	4300	1013	1003
ht-coarsegrain-hp						
1	364181	196251	302001	136741	511	511
2	349123	196251	59523	12228	511	511
3	737570	196251	23524	2000	511	507
ht-coarsegrain-block						
1	198602	196251	239337	142006	448	447
2	367966	196251	33889	12458	511	445
3	737570	196251	24659	2049	511	394

on up to 1024 cores. The experiments showed that the proposed fine-grain MTTKRP can achieve the best performance with respect to other alternatives with a good partitioning, reaching up to 194x speedups on 512 cores.

In our analysis and experiments, we identified the communication latency as the dominant hindrance for further scalability of the fastest proposed method. We will investigate this in the future. We also note that the size of the hypergraphs that we build can cause discomfort to all existing partitioning tools. Methods that partition huge hypergraphs efficiently and effectively are needed for handling larger tensors than those treated in this work.

During the revision process, we have come across to a new study called DMS on distributed memory tensor factorization [30] which uses SPLATT formulation. The communication is based on all-to-all primitives. The current versions of the two codes (ours using MPI, DMS using OpenMP+MPI) do not allow a useful comparison. We plan to update our codes and do a comparison in the near future.

6. ACKNOWLEDGMENTS

This work was performed using HPC resources from GENCI-[TGCC/CINES/IDRIS] (Grant 2015-100570). Additional computational resources were used from the PSMN computing center at ENS Lyon.

7. REFERENCES

[1] E. Acar, D. M. Dunlavy, and T. G. Kolda. A scalable optimization approach for fitting canonical tensor decompositions. *Journal of Chemometrics*, 25(2):67–86, February 2011.

[2] C. A. Andersson and R. Bro. The N-way toolbox for MATLAB. *Chemometrics and Intelligent Laboratory Systems*, 52(1):1–4, 2000.

[3] C. J. Appellof and E. Davidson. Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents. *Analytical Chemistry*, 53(13):2053–2056, 1981.

[4] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 2008.

[5] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[6] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6. Available online, February 2015.

[7] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[8] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[9] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[10] J. D. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[11] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[12] Ü . V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.*

Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm, 1999.

[13] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering*. PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.

[14] J. H. Choi and S. V. N. Vishwanathan. DFacTo: Distributed factorization of tensors. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *27th Advances in Neural Information Processing Systems*, pages 1296–1304, Montreal, Quebec, Canada, 2014.

[15] P. Comon. Tensors: A brief introduction. *IEEE Signal Processing Magazine*, 31(3):44–53, May 2014.

[16] L. De Lathauwer and B. De Moor. From matrix to tensor: Multilinear algebra and signal processing. In *Institute of Mathematics and Its Applications Conference Series*, volume 67, pages 1–16, 1998.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[18] O. Görlitz, S. Sizov, and S. Staab. Pints: Peer-to-peer infrastructure for tagging systems. In *Proceedings of the 7th International Conference on Peer-to-peer Systems*, IPTPS'08, page 19, Berkeley, CA, USA, 2008. USENIX Association.

[19] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[20] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.

[21] B. Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25:99–108, 2000.

[22] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[23] U. Kang, 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*, KDD '12, pages 316–324, New York, NY, USA, 2012. ACM.

[24] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998.

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

[26] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[27] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley–Teubner, Chichester, U.K., 1990.

[28] 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.

[29] R. O. Selvitopi, M. M. Ozdal, and C. Aykanat. A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):632–645, March 2015.

[30] S. Smith and G. Karypis. DMS: Distributed sparse tensor factorization with alternating least squares. Technical Report 15-007, Department of Computer Science and Engineering, University of Minnesota, May 2015.

[31] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *29th IEEE International Parallel & Distributed Processing Symposium*, 2015. To appear.

[32] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[33] B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review*, 49(4):595–603, 2007.

[34] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Computer Vision—ECCV 2002*, pages 447–460. Springer, 2002.