HAL
open science

Parallel CP decomposition of sparse tensors using dimension trees

Oguz Kaya, Bora Uçar

To cite this version:

Oguz Kaya, Bora Uçar. Parallel CP decomposition of sparse tensors using dimension trees. [Research Report] RR-8976, Inria - Research Centre Grenoble – Rhône-Alpes. 2016. hal-01397464v1

HAL Id: hal-01397464

https://inria.hal.science/hal-01397464v1

Submitted on 15 Nov 2016 (v1), last revised 17 Oct 2018 (v2)

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.

Parallel CP decomposition of sparse tensors using dimension trees

Oguz Kaya, Bora Uçar

RESEARCH REPORT
N° 8976
November 2016
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8976--FR+ENG

Parallel CP decomposition of sparse tensors using dimension trees

Oguz Kaya*, Bora Uçar†

Project-Team ROMA

Research Report n° 8976 — November 2016 — 35 pages

Abstract: Tensor factorization has been increasingly used to address various problems in many fields such as signal processing, data compression, computer vision, and computational data analysis. CANDECOMP/PARAFAC (CP) decomposition of sparse tensors has successfully been applied to many well-known problems in web search, graph analytics, recommender systems, health care data analytics, and many other domains. In these applications, computing the CP decomposition of sparse tensors efficiently is essential in order to be able to process and analyze data of massive scale. For this purpose, we investigate an efficient computation and parallelization of the CP decomposition for sparse tensors. We provide a novel computational scheme for reducing the cost of a core operation in computing the CP decomposition with the traditional alternating least squares (CP-ALS) based algorithm. We then effectively parallelize this computational scheme in the context of CP-ALS in shared and distributed memory environments, and propose data and task distribution models for better scalability. We implement parallel CP-ALS algorithms and compare our implementations with an efficient tensor factorization library, using tensors formed from real-world and synthetic datasets. With our algorithmic contributions and implementations, we report up to 3.95x, 3.47x, and 3.9x speedups in sequential, shared memory parallel, and distributed memory parallel executions over the state of the art, and up to 1466x overall speedup over the sequential execution using 4096 cores on an IBM BlueGene/Q supercomputer.

Key-words: sparse tensors, CP decomposition, dimension tree, parallel algorithms

* Inria and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France
† CNRS and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes parallèles pour la décomposition CP des tenseurs creux avec des arbres des dimensions

Résumé : Nous nous intéressons à la décomposition CANDECOMP/PARAFAC (CP) de tenseurs creux. Nous proposons un nouveau schéma de calcul pour réduire le coût d'une opération dans le calcul de la décomposition CP avec l'algorithme d'alternance des moindres carrés (CP-ALS). Nous investiguons une parallélisation efficace de ce schéma de calcul dans le contexte de CP-ALS pour le parallélisme à mémoire partagée et à mémoire distribuée, et proposons des modèles de distribution des données et des tâches pour obtenir un meilleur passage à l'échelle. Nous développons une implémentation de CP-ALS et comparons ce dernier avec d'autres bibliothèques de factorisation de tenseurs, en utilisant des tenseurs venant des applications diverses. Nous obtenons des facteurs d'accélération 3.95x, 3.47x, and 3.9x en séquentiel, dans un système à mémoire partagée et dans un système à mémoire distribuée, contre d'autres implémentations. Aussi, nous rapportons des facteurs d'accélération jusqu'à 1466x sur 4096 noeuds d'IBM BlueGene/Q contre une execution séquentiel.

Mots-clés : tenseurs creux, decomposition CP, l'arbre de dimension, calcul paralléle

1 Introduction

With the higher dimensionality of data, tensors, or multi-dimensional arrays, have increasingly been used in many fields including the analysis of Web graphs [27], knowledge bases [10], recommender systems [33, 34, 40], signal processing [29], computer vision [42], health care [32], and many others [28]. Tensor decomposition algorithms are used as an effective tool for analyzing data in order to extract latent information within the data, or predict missing data elements. There have been considerable efforts in designing numerical algorithms for different tensor decomposition problems (see the survey [28]), and algorithmic and software contributions go hand in hand with these efforts [2, 6, 16, 21, 25, 26, 37, 38, 38].

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. The most popularly used algorithm for computing a CP decomposition is CP-ALS [11, 19], which is based on the alternating least squares method. Other variants also exist for computing a CP decomposition [1], yet CP-ALS is generally considered as the "workhorse" algorithm yielding the best trade-off in terms of accuracy and computational cost [28]. All these algorithms are iterative, in which the computational core of each iteration involves 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 $N-1$ matrix row vectors and their scaled sum reduction according to the nonzero structure of the tensor. When the dimensionality of the tensor increases, this operation gets computationally more expensive; hence, efficiently carrying out MTTKRP for higher dimensional tensors is of our particular interest due to the needs of emerging applications [32]. This operation has received recent interest for efficient execution in different settings such as MATLAB [2, 6], MapReduce [21], shared memory [38], and distributed memory [16, 25, 37]. We are interested in a fast computation of MTTKRP as well as CP-ALS for sparse tensors using efficient computational schemes and effective parallelization in shared and distributed memory environments.

Our contributions in this paper are as follows. We investigate the parallelization of CP-ALS algorithm in shared and distributed memory environments for sparse tensors. For the shared-memory computations, we propose a novel computational scheme that significantly reduces the computational cost while offering an effective parallelism. We then perform theoretical analyses corresponding to the computational gains and the memory utilization, which are also validated with experiments. We propose a fine-grain distributed memory parallel algorithm and compare it against a medium-grain variant [37]. Finally, we discuss effective partitioning routines for these algorithms.

The organization of the rest of the paper is as follows. In the next section, we introduce our notation, describe CP-ALS method, and present a data structure called dimension tree which enables efficient CP-ALS computations. Next, in Section 3 we explain how to use dimension trees to carry out MTTKRPs within CP-ALS. Afterwards, in Section 4 we discuss an effective shared and distributed memory parallelization of CP-ALS iterations. We introduce partitioning problems pertaining to distributed memory parallel performance, and use these partitioning methods in our experiments using real-world tensors. In Section 5 we give an overview of the existing literature. Finally, in Section 6 we present experimental results to demonstrate the performance gains using our algorithms for shared and distributed memory parallelism over an efficient state of the art implementation.

2 Background and notation

We denote the set $\{1, \ldots, N\}$ of integers as $\mathbb{N}_N$ for $N \in \mathbb{Z}^+$. For vectors, we use bold lowercase Roman letters, as in $\mathbf{x}$. For matrices, we use bold uppercase Roman letters, e.g., $\mathbf{X}$. For tensors, we generally follow the notation in Kolda and Bader's survey [28]. We represent tensors using bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{X}}$. The *order* of a tensor is defined as the number of its *dimensions*, or equivalently, *modes*, which we denote by N. We use italic lowercase letters with corresponding indices to represent vector, matrix, and tensor elements, e.g., x_i for a vector x, x_{ij} for a matrix $\mathbf{X}$, and x_{ijk} for a 3-dimensional tensor $\boldsymbol{\mathcal{X}}$. For the column vectors of a matrix, we use the same letter in lowercase and with a subscript corresponding to the column index, e.g., $\mathbf{x}_i$ to denote $\mathbf{X}(:, i)$. A *slice* of a tensor in the nth mode is a set of tensor elements obtained by fixing the index only along the nth mode. We use the MATLAB notation to refer to matrix rows and columns as well as tensors slices, e.g., $\mathbf{X}(i, :)$ and $\mathbf{X}(:, j)$ are the ith row and the jth column of $\mathbf{X}$, whereas $\boldsymbol{\mathcal{X}}(k, :, :)$ represents the kth slice of $\boldsymbol{\mathcal{X}}$ in the first dimension.

Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be an N-mode tensor whose size in mode n is I_n for $n \in \mathbb{N}_N$. The multiplication of $\boldsymbol{\mathcal{X}}$ along mode n with a vector $\mathbf{v} \in \mathbb{R}^{I_d}$ is a tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times 1 \times I_{n+1} \times \cdots \times I_N}$ with elements

$$y_{i_1,\ldots,i_{n-1},1,i_{n+1},\ldots,i_N} = \sum_{j=1}^{I_n} v_j x_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N}. \tag{1}$$

This operation is called tensor-times-vector multiply (TTV) and denoted by $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{v}$. This formulation of TTV is commutative [4].

A tensor $\boldsymbol{\mathcal{X}}$ can be *matricized*, meaning that a matrix $\mathbf{X}$ can be associated with $\boldsymbol{\mathcal{X}}$ by identifying a subset of its modes to correspond to the rows of $\mathbf{X}$, and the rest of the modes to correspond to the columns of $\mathbf{X}$. This involves a mapping of the elements of $\boldsymbol{\mathcal{X}}$ to those of $\mathbf{X}$. We will be exclusively dealing with the matricizations of tensors along a single mode, meaning that a single mode is mapped to the rows of the resulting matrix, and the rest of the modes correspond to its columns. We use $\mathbf{X}_{(d)}$ to denote matricization along the mode d, e.g., for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, the matrix $\mathbf{X}_{(1)}$ denotes the mode-1 matricization of $\boldsymbol{\mathcal{X}}$. Specifically, in this matricization the tensor element $x_{i_1,\ldots,i_N}$ corresponds to the element $\left(i_1, i_2 + \sum_{j=3}^{N} \left[(i_j - 1) \prod_{k=2}^{j-1} I_k\right]\right)$ of $\mathbf{X}_{(1)}$. Matricizations in other modes are defined similarly.

The *Hadamard product* of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^I$ is a vector $\mathbf{w} = \mathbf{u} * \mathbf{v}, \mathbf{w} \in \mathbb{R}^I$, where $w_i = u_i \cdot v_i$. The *outer product* of $K > 1$ vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(K)}$ of corresponding sizes $I_1, \ldots, I_K$ is denoted by $\boldsymbol{\mathcal{X}} = \mathbf{u}^{(1)} \circ \cdots \circ \mathbf{u}^{(K)}$ where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_K}$ is a K-dimensional tensor with elements $x_{i_1,\ldots,i_K} = \prod_{t \in \mathbb{N}_K} u^{(t)}_{i_t}$. The *Kronecker product* of vectors $\mathbf{u} \in \mathbb{R}^I$ and $\mathbf{v} \in \mathbb{R}^J$ results in a vector $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ where $\mathbf{w} \in \mathbb{R}^{IJ}$ is defined as

$$\mathbf{w} = \mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} u_1 \mathbf{v} \\ u_2 \mathbf{v} \\ \vdots \\ u_I \mathbf{v} \end{bmatrix}.$$

For matrices $\mathbf{U} \in \mathbb{R}^{I \times K}$ and $\mathbf{V} \in \mathbb{R}^{J \times K}$, their *Khatri-Rao product* corresponds to

$$\mathbf{W} = \mathbf{U} \odot \mathbf{V} = [\mathbf{u}_1 \otimes \mathbf{v}_1, \ldots, \mathbf{u}_K \otimes \mathbf{v}_K], \tag{2}$$

where $\mathbf{W} \in \mathbb{R}^{IJ \times K}$.

The rank-R CP-decomposition of a tensor $\mathcal{X}$ expresses or approximates $\mathcal{X}$ as a sum of R rank-1 tensors. For instance, for $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, one writes $\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ where $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$, and $\mathbf{c}_r \in \mathbb{R}^K$. This decomposition results in the element-wise approximation (or equality) $x_{i,j,k} \approx \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$. The minimum R value rendering this approximation an equality is called as the *rank* (or CP-rank) of the tensor $\mathcal{X}$, which is NP-hard [20]. Here, the matrices $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_R]$, $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_R]$, and $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_R]$ 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 having $I_1, \ldots, I_N$ rows and R columns, and $\mathbf{u}_j^{(i)}$ to refer to the jth column of $\mathbf{U}^{(i)}$. The standard algorithm for computing CP decomposition is the Alternating Least Squares (CP-ALS) method, which establishes a good trade-off between the convergence rate (number of iterations) and the iteration cost [28]. It is an iterative algorithm, shown in Algorithm 1, that progressively updates the factors $\mathbf{U}^{(n)}$ in an alternating fashion staring from an initial guess. CP-ALS stops until it can no longer improve the solution, or it reaches the allowed maximum number of iterations. The initial factor matrices can be random or cane be computed using the truncated SVD of the matricizations of $\mathcal{X}$ [28]. Each iteration of CP-ALS consists of N *subiterations*, where in the nth subiteration $\mathbf{U}^{(n)}$ is updated using $\mathcal{X}$ as well as the current values of all other factor matrices.

Algorithm 1 CP-ALS: ALS algorithm for computing CP decomposition

Input: $\mathcal{X}$: An N-mode tensor, $\mathcal{X} \in \mathbb{R}^{I_1, \ldots, I_N}$
R: The rank of CP decomposition
$\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$: The initial factor matrices
Output: $[\![\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$: The Rank-R CP decomposition of $\mathcal{X}$

1: **repeat**
2: **for** $n = 1, \ldots, N$ **do**
3: $\mathbf{M}^{(n)} \leftarrow \mathcal{X}_{(n)}(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)})$
4: $\mathbf{H}^{(n)} \leftarrow \mathbf{U}^{(N)T}\mathbf{U}^{(N)} * \cdots * \mathbf{U}^{(n+1)T}\mathbf{U}^{(n+1)} * \mathbf{U}^{(n-1)T}\mathbf{U}^{(n-1)} * \cdots * \mathbf{U}^{(1)T}\mathbf{U}^{(1)}$
5: $\mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)}\mathbf{H}^{(n)\dagger}$ ▶ $\mathbf{H}^{(n)\dagger}$ is the pseudoinverse of $\mathbf{H}^{(n)}$.
6: $\lambda \leftarrow$ COLUMN-NORMALIZE$(\mathbf{U}^{(n)})$
7: **until** convergence is achieved or the maximum number of iterations is reached.
8: **return** $[\![\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

Computing the matrix $\mathbf{M}^{(n)} \in \mathbb{R}^{I_n \times R}$ at Line 3 of Algorithm 1 is the sole part involving the tensor $\mathcal{X}$, and it is the most expensive computational step of the CP-ALS algorithm, both for sparse and dense tensors. The operation $\mathcal{X}_{(n)}(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)})$ is called the *matricized tensor-times Khatri-Rao product* (MTTKRP). The Khatri-Rao product of the involved $\mathbf{U}^{(n)}$s defines a matrix of size $(\prod_{i \neq n} I_i) \times R$ according to (2), and can get very costly in terms of computational and memory requirements when I_i or N is large—which is the case for many real-world sparse tensors. To alleviate this, various methods are proposed in the literature that enable performing MTTKRP without forming the Khatri-Rao product. One such formulation [5], also used in Tensor Toolbox [6], expresses MTTKRP in terms of a series of TTVs and computes the resulting matrix $\mathbf{M}^{(n)}$ column by column. With this formulation, the rth column of $\mathbf{M}^{(n)}$ can be computed using $N-1$ TTVs as in

$$\mathbf{M}^{(n)}(:, r) \leftarrow \mathcal{X} \times_1 \mathbf{u}_r^{(1)} \times_2 \cdots \times_{n-1} \mathbf{u}_r^{(n-1)} \times_{n+1} \mathbf{u}_r^{(n+1)} \times_{n+2} \cdots \times_N \mathbf{u}_r^{(N)} . \tag{3}$$

Once $\mathbf{M}^{(n)}$ is obtained, the Hadamard product of the matrices $\mathbf{U}^{(i)T}\mathbf{U}^{(i)}$ of size $R \times R$ is com-

puted for $1 \leq i \leq N, i \neq n$ to form the matrix $\mathbf{H}^{(n)} \in \mathbb{R}^{R \times R}$. Note that within the subiteration n, only $\mathbf{U}^{(n)}$ is updated among all factor matrices. Therefore, for efficiency, one can precompute all matrices ${\mathbf{U}^{(i)}}^T\mathbf{U}^{(i)}$ of size $R \times R$ for $1 \leq i \leq N$, then update $\mathbf{U}^{(n)}\mathbf{U}^{(n)}$ once $\mathbf{U}^{(n)}$ changes. As the rank R of approximation is chosen as a small constant in practice for sparse tensors (less than 50) [43], performing these Hadamard products to compute $\mathbf{H}^{(n)}$ and the matrix-matrix multiplication to compute ${\mathbf{U}^{(n)}}^T\mathbf{U}^{(n)}$ become relatively cheap compared with the TTV step. Once both $\mathbf{M}^{(n)}$ and $\mathbf{H}^{(n)}$ are computed, another matrix-matrix multiplication is performed using $\mathbf{M}^{(n)}$ and the pseudoinverse of $\mathbf{H}^{(n)}$ in order to update the matrix $\mathbf{U}^{(n)}$ (which is not expensive as R is small). Finally, $\mathbf{U}^{(n)}$ is normalized column-wise, and the column vector norms are stored in the vector $\lambda \in \mathbb{R}^R$.

The converge is achieved when the relative improvement in the error norm, i.e., $\mathcal{X} - \sum_{r=1}^{R} \lambda_r (\mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)})$, is small. The cost of this computation is insignificant.

3 Computing CP decomposition using dimension trees

A *dimension tree* partitions the mode indices of an N-dimensional tensor in a hierarchical manner for computing tensor decompositions efficiently. It was first used in the hierarchical Tucker format representing the hierarchical Tucker decomposition and SVD of a tensor [18], which was introduced as a computationally feasible alternative to the original Tucker decomposition for higher order tensors.

We propose a novel way of using dimension trees for computing the *standard* CP decomposition of tensors with a formulation that asymptotically reduces the computational cost. In doing so, we do not alter the original CP decomposition in any way. The reduction in the computational cost is made possible by storing partial TTV results, and hence by trading off more memory. A similar idea of reusing partial results was moderately explored by Baskaran et al. [7] for computing the Tucker decomposition of sparse tensors, which we generalized using dimension trees for better computational gains [24] in the standard algorithm for Tucker decomposition. Here, we adopt the same data structure for reducing the cost of MTTKRP operations in computing the CP decomposition. We first provide a modified formal definition of the dimension tree, and then illustrate how we use it in computing the CP decomposition.

Definition 1. *A dimension tree $\mathcal{T}$ for N dimensions is a rooted tree with N leaf nodes. In a dimension tree, each non-leaf node has at least two children. The root of the tree is denoted by* $\textsc{Root}(\mathcal{T})$*, and the leaf nodes are denoted by* $\textsc{Leaves}(\mathcal{T})$*. Each $t \in \mathcal{T}$ is associated with a* ***mode set*** *$\mu(t) \subseteq \mathbb{N}_N$ satisfying the following properties:*

1. $\mu(\textsc{Root}(\mathcal{T})) = \mathbb{N}_N$.
2. *For each non-leaf node $t \in \mathcal{T}$, the mode sets of its children partition $\mu(t)$.*
3. *The ith leaf node, denoted by $\mathcal{L}_i \in \textsc{Leaves}(\mathcal{T})$, has $\mu(\mathcal{L}_i) = \{i\}$.*

For the simplicity of the presentation, we assume without loss of generality that the sequence $\mathcal{L}_1, \ldots, \mathcal{L}_N$ corresponds to a left-to-right traversal of the leaves of the dimension tree, assuming a left-to-right ordering for the children of each node. If this is not the case, we can relabel the tensor modes accordingly. We define the *inverse mode set* of a node t as $\gamma(t) = \mathbb{N}_N \setminus \mu(t)$. For each node t

with a parent $P(t)$, $\mu(t) \subset \mu(P(t))$ holds due to the second property, which yields $\gamma(t) \supset \gamma(P(t))$. If a dimension tree has the height $\lceil \log(N) \rceil$ with its first $\lfloor \log(N) \rfloor$ levels forming a complete binary tree, we call it a (balanced) binary dimension tree (BDT). In Fig. 1, we demonstrate a BDT for 4 dimensions (associated with a sparse tensor described later).

3.1 Using dimension trees to perform successive tensor-times-vector multiplies

At each subiteration of the CP-ALS algorithm, using (3), $\mathcal{X}$ is multiplied with the column vectors of matrices in $N-1$ modes in performing the MTTKRP. Some of these TTVs involve the same matrices as the preceding subiterations. As the TTV operation is commutative, this opens up the possibility to factor out and reuse TTV results that are common in consecutive subiterations for reducing the computational cost. For instance, in the first subiteration of CP-ALS using a 4-mode tensor $\mathcal{X}$, for each $r \in \mathbb{N}_R$ we compute $\mathcal{X} \times_2 \mathbf{u}_r^{(2)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and eventually update $\mathbf{u}_r^{(1)}$, whereas in the second subiteration we compute $\mathcal{X} \times_1 \mathbf{u}_r^{(1)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and update $\mathbf{u}_r^{(2)}$. In these two subiterations, the matrices $\mathbf{U}^{(3)}$ and $\mathbf{U}^{(4)}$ remain unchanged, and both TTV steps involve the TTV of $\mathcal{X}$ with $\mathbf{u}_r^{(3)}$ and $\mathbf{u}_r^{(4)}$. Hence, we can compute $\mathcal{Y}_r = \mathcal{X} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$, then reuse it in the first and the second subiterations as $\mathcal{Y}_r \times_2 \mathbf{u}_r^{(2)}$ and $\mathcal{Y}_r \times_1 \mathbf{u}_r^{(1)}$ to obtain the final TTV results for these modes.

Algorithm 2 DTREE-TTV: Dimension tree-based TTV with R vectors in each mode

Input: t: A node of the dimension tree
Output: Tensors $T_:(t)$ of t are computed.

```
if EXISTS(T_:(t)) then
    return                                          ▶ Tensors T_:(t) are already computed.
DTREE-TTV(P(t))                                     ▶ Compute the parent's tensors T_:(P(t)) first.
for r = 1, ..., R do                                ▶ Now update all tensors of t using parent's.
    T_r(t) ← T_r(P(t)) ×_{d_1} u_r^{(d_1)} ×_{d_2} ··· ×_{d_|δ(t)|} u_r^{(d_|δ(t)|)}
```

We associate a dimension tree $\mathcal{T}$ for N dimensions with an N-dimensional tensor $\mathcal{X}$ as follows. With each node $t \in \mathcal{T}$, we associate R tensors $T_1(t), \ldots, T_R(t)$. $T_r(t)$ corresponds to the TTV result $T_r(t) = \mathcal{X} \times_{y_1} \mathbf{u}_r^{(y_1)} \times_{y_2} \cdots \times_{y_{|\gamma(t)|}} \mathbf{u}_r^{(y_{|\gamma(t)|})}$, where $y_i \in \gamma(t)$ are distinct elements of the inverse mode set $\gamma(t)$. Therefore, $\gamma(t)$ corresponds to the set of modes for which TTV is performed on $\mathcal{X}$ to form $T_r(t)$. For the root of the tree, $\gamma(\text{ROOT}(\mathcal{T})) = \emptyset$; thus, all tensors of the root node correspond to the original tensor $\mathcal{X}$. Since $\gamma(t) \supset \gamma(P(t))$ for a node t and its parent $P(t)$, tensors of $P(t)$ can be used as partial results to update the tensors of t. Let $\delta(t) = \gamma(t) \setminus \gamma(P(t))$ denote the *multiplication modes* of the node t. We can then compute each tensor of t from its parent's as $T_r(t) = T_r(P(t)) \times_{d_1} \mathbf{u}_r^{(d_1)} \times_{d_2} \cdots \times_{d_{|\delta(t)|}} \mathbf{u}_r^{(d_{|\delta(t)|})}$ for $\delta(t) = \{d_1, \ldots, d_{|\delta(t)|}\}$. This procedure is called DTREE-TTV and is shown in Algorithm 2. DTREE-TTV first checks if the tensors of t are already computed, and immediately returns if so. This happens, for example, when two children of t call DTREE-TTV on t consecutively, in which case for the second DTREE-TTV call, the tensors of t would be already computed. If the tensors of t are not already computed, DTREE-TTV on $P(t)$ is called first to make sure that $P(t)$'s tensors are up-to-date. Then, each $T_r(t)$ is computed by performing a TTV on the corresponding tensor $T_r(P(t))$ of the parent. We use the notation $T_:(t)$ to denote all R tensors of a node t.

Algorithm 3 DTREE-CP-ALS: Dimension tree-based CP-ALS algorithm

Input: $\mathcal{X}$: An N-mode tensor
R: The rank of CP decomposition
Output: $[\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\mathcal{X}$

1: $\mathcal{T} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\mathcal{X}$) ▶ The tree has leaves $\{\mathcal{L}_1, \ldots, \mathcal{L}_N\}$.
2: **for** $n = 2 \ldots N$ **do**
3: $\quad \mathbf{W}^{(n)} \leftarrow \mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$
4: **repeat**
5: $\quad$ **for** $n = 1, \ldots, N$ **do**
6: $\quad\quad$ **for all** $t \in \mathcal{T}$ **do**
7: $\quad\quad\quad$ **if** $n \in \gamma(t)$ **then**
8: $\quad\quad\quad\quad$ DESTROY($T_{:}(t)$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
9: $\quad\quad$ DTREE-TTV($\mathcal{L}_n$) ▶ Perform the TTVs for the leaf node tensors.
10: $\quad\quad$ **for** $r = 1, \ldots, R$ **do** ▶ Form $\mathbf{M}^{(n)}$ column-by-column (done implicitly).
11: $\quad\quad\quad \mathbf{M}^{(n)}(:, r) \leftarrow T_r(\mathcal{L}_n)$ ▶ $T_r(\mathcal{L}_n)$ is a vector of size I_n for the leaf node $\mathcal{L}_n$.
12: $\quad\quad \mathbf{H}^{(n)} \leftarrow \mathbf{W}^{(N)} * \cdots * \mathbf{W}^{(n+1)} * \mathbf{W}^{(n-1)} * \cdots * \mathbf{W}^{(1)}$
13: $\quad\quad \mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)}\mathbf{H}^{(n)^\dagger}$
14: $\quad\quad \lambda \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
15: $\quad\quad \mathbf{W}^{(n)} \leftarrow \mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$
16: **until** converge is achieved or the maximum number of iterations is reached.
17: **return** $[\![\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

3.2 Dimension tree-based CP-ALS algorithm

At the nth subiteration of Algorithm 1, we need to compute $\mathcal{X} \times_1 \mathbf{u}_r^{(1)} \times_2 \cdots \times_{n-1} \mathbf{u}_r^{(n-1)} \times_{n+1} \mathbf{u}_r^{(n+1)} \times_{n+2} \cdots \times_N \mathbf{u}_r^{(N)}$ for all $r \in \mathbb{N}_R$ in order to form $\mathbf{M}^{(n)}$. Using a dimension tree $\mathcal{T}$ with leaves $\mathcal{L}_1, \ldots, \mathcal{L}_N$, we can perform this simply by executing DTREE-TTV($\mathcal{L}_n$), after which the rth tensor of $\mathcal{L}_n$ provides the rth column of $\mathbf{M}^{(n)}$. Once $\mathbf{M}^{(n)}$ is formed, the remaining steps follow as before. We show the whole CP-ALS using a dimension tree in Algorithm 3. At Line 1, we construct a dimension tree $\mathcal{T}$ with leaf nodes $\mathcal{L}_1, \ldots, \mathcal{L}_N$. This tree can be constructed in any way that respects the properties of a dimension tree described in Section 3; but for our purposes we assume that it is formed as a BDT. At Line 8 within the nth subiteration, we destroy all tensors of a node t if its set of multiplied modes $\gamma(t)$ involve n, as in this case its tensors involve multiplication using the old value of $\mathbf{U}^{(n)}$ which is about to change. Afterwards, DTREE-TTV is called at Line 9 for the leaf node $\mathcal{L}_n$ to compute its tensors. Next, the rth column of $\mathbf{M}^{(n)}$ is formed using $T_r(\mathcal{L}_n)$ for $r \in \mathbb{N}_R$. Once $\mathbf{M}^{(n)}$ is ready, $\mathbf{H}^{(n)}$ and $\mathbf{U}^{(n)}$ are computed as before, after which $\mathbf{U}^{(n)}$ is normalized.

Performing TTVs in CP-ALS using a BDT in this manner provides significant computational gains with a moderate increase in the memory cost. We now state two theorems pertaining to the computational and memory efficiency of DTREE-CP-ALS.

Theorem 1. *Let $\mathcal{X}$ be an N-mode tensor. The total number of TTVs at each iteration of Algorithm 3 using a BDT is at most $RN\lceil \log N \rceil$.*

Proof. As we assume that the sequence $\mathcal{L}_1, \ldots, \mathcal{L}_N$ corresponds to the left-to-right ordering of the leaves of the dimension tree, for each internal node t, the subtree rooted at t has the leaves $\mathcal{L}_i, \mathcal{L}_{i+1}, \ldots, \mathcal{L}_{i+k-1}$ corresponding to k consecutive mode indices for some positive integers i and k. As we have $\mu(\mathcal{L}_i) = i$ for the ith leaf node, we obtain $\mu(t) = \{i, i+1, \ldots, i+k-1\}$ due to the second property of dimension trees. As a result, for each leaf node $\mathcal{L}_{i+k'}$ for $0 \leq k' < k$, we

have $i + k' \in \mu(t)$; hence $i + k' \notin \gamma(t)$ as $\gamma(t) = \mathbb{N}_N \setminus \mu(t)$. Therefore, within an iteration of Algorithm 3 the tensors of t get computed at the ith subiteration, stay valid (not destroyed) and get reused until the $i+k-1$th subiteration, and finally get destroyed in the following subiteration. Once destroyed, the tensors of t are never recomputed in the same iteration, as all the modes associated with the leaf tensors in its subtree are processed (which are the only nodes that can reuse the tensors of t). As a result, in one CP-ALS iteration, tensors of every tree node (except the root) gets to be computed and destroyed exactly once. Therefore, the total number of TTVs in one iteration becomes the sum of the number of TTVs performed to compute the tensors of each node in the tree once. In computing its tensors, every node t has R tensors, and for each tensor it performs TTVs for each dimension in the set $\delta(t)$ in Algorithm 2, except the root node whose tensors all equal to $\mathcal{X}$ and never change. Therefore, we can express the total number of TTVs performed within a CP-ALS iteration due to one of these R tensors as

$$\sum_{t \in \mathfrak{T}, t \neq \textsc{Root}(\mathfrak{T})} |\delta(t)| = \sum_{t \in \mathfrak{T}, t \neq \textsc{Root}(\mathfrak{T})} |\mu(P(t)) \setminus \mu(t)| \tag{4}$$

Since in a BDT, every non-leaf node t has exactly two children, say t_1 and t_2, we obtain $|\mu(t) \setminus \mu(t_1)| + |\mu(t) \setminus \mu(t_2)| = |\mu(t)|$, as $\mu(t)$ is partitioned into two sets $\mu(t_1)$ and $\mu(t_2)$. With this observation, we can reformulate (4) as

$$\sum_{t \in \mathfrak{T}, t \neq \textsc{Root}(\mathfrak{T})} |\mu(P(t)) \setminus \mu(t)| = \sum_{t \in \mathfrak{T} \setminus \textsc{Leaves}(\mathfrak{T})} |\mu(t)| \ . \tag{5}$$

Note that in constructing a BDT, at the root node we start with the mode set $\mu(\textsc{Root}(\mathfrak{T})) = \mathbb{N}_N$. Then, at each level $l > 0$, we form the mode sets of the nodes at level l by partitioning the mode sets of their parents at level $l-1$. As a result, at each level l, each dimension $n \in \mathbb{N}_N$ can appear in only one set $\mu(t)$ for a node t belonging to the level l of the BDT. With this observation in mind, rewriting (5) by iterating over nodes by levels of the BDT yields

$$\begin{aligned} \sum_{t \in \mathfrak{T} \setminus \textsc{Leaves}(\mathfrak{T})} |\mu(t)| &= \sum_{l=1}^{\lceil \log N \rceil} \sum_{t \in \mathfrak{T} \setminus \textsc{Leaves}(\mathfrak{T}), \textsc{Level}(t) = l} |\mu(t)| \\ &\leq \sum_{l=1}^{\lceil \log N \rceil} N = N \lceil \log N \rceil \ . \end{aligned}$$

As there are R tensors in each BDT tree node, the overall cost becomes $RN\lceil \log N \rceil$ TTVs for a CP-ALS iteration. Note that the traditional scheme [38] incurs $R(N-1)$ TTVs for each mode, and $RN(N-1)$ TTVs in total for a CP-ALS iteration. □

Theorem 2. *For an N-mode tensor $\mathcal{X}$, the total number of tree tensors that are not destroyed is at most $R\lceil \log N \rceil$ at any instant of Algorithm 3 using a BDT.*

Proof. Note that at the beginning of the nth subiteration of Algorithm 3, the tensors of each node $t \in \mathfrak{T}$ involving n in $\gamma(t)$ are destroyed at Line 8. These are exactly the tensors that do not lie in the path from the leaf $\mathcal{L}_n$ to the root, as they do not involve n in their mode set μ. The TTV result for $\mathcal{L}_n$ depends only on the nodes on this path from $\mathcal{L}_n$ to the root; therefore, at the end of the nth subiteration, only the tensors of the nodes on this path will be computed using Algorithm 2. As this path length cannot exceed $\lceil \log N \rceil$ in a BDT, and each node has R tensors, the number of non-destroyed tensors cannot exceed $R\lceil \log N \rceil$ at any instant of Algorithm 3. □

Theorem 2 puts a nice upper bound of $R\lceil \log N \rceil$ on the maximum number of allocated tree tensors, thus on the maximum memory utilization, in DTREE-CP-ALS.

4 Parallel CP-ALS for sparse tensors using dimension trees

We first present shared memory parallel algorithms involving efficient parallelization of the dimension tree-based TTVs in Section 4.1. Later, in Section 4.2, we present distributed memory parallel algorithms that use this shared memory parallelization.

4.1 Shared memory parallelism

Using a sparse tensor, the most time consuming part within a CP-ALS iteration is the TTV step involving costly sparse irregular tensor operations. The rest of the iteration involves dense matrix algebra of small matrices. For this reason, the rapid execution of the sparse TTV step plays a crucial role in obtaining high performance.

One key idea we use for obtaining high performance is performing TTVs for all R tensors $T_:(t)$ of a node $t \in \mathcal{T}$ in a *vectorized* manner. We illustrate this on a 4-dimensional tensor $\mathcal{X}$ and a BDT, and for clarity, we put the mode set $\mu(t)$ of each tree node t in the subscript, as in t_{1234}. Let t_{1234} represent the root of the BDT with $T_r(t_{1234}) = \mathcal{X}$ for all $r \in \mathbb{N}_R$. The two children of t_{1234} are t_{12} and t_{34} with the corresponding tensors $T_r(t_{12}) = T_r(t_{1234}) \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and $T_r(t_{34}) = T_r(t_{1234}) \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$, respectively. Since $T_r(t_{1234})$s are identical for all $r \in \mathbb{N}_R$, the nonzero pattern of all tensors $T_r(t_{12})$ are identical for all $r \in \mathbb{N}_R$. The same is true for $T_r(t_{34})$s. The same argument applies to the children t_1 and t_2 of t_{12}, as well as t_3 and t_4 of t_{34}. As a result, each node in the tree involves R tensors with identical nonzero patterns. This opens up two possibilities in terms of efficiency. First, it is sufficient to compute only one set of nonzero indices for each node $t \in \mathcal{T}$ to represent the nonzero structure of all of its tensors, which reduces the computational and memory cost by a factor of R. Second, we can perform the TTVs for all tensors at once in a "vectorized" manner by modifying (1) to perform R TTVs of the form $\mathcal{Y}^{(r)} \leftarrow \mathcal{X}^{(r)} \times_d \mathbf{v}_r$ for $\mathbf{V} = [\mathbf{v}_1 | \dots | \mathbf{v}_R] \in \mathbb{R}^{I_d \times R}$ as follows:

$$\mathbf{y}^{(:)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N} = \sum_{j=1}^{I_d} \mathbf{V}(j,:) * \mathbf{x}^{(:)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N} \tag{6}$$

where $\mathbf{y}^{(:)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N}$ and $\mathbf{x}^{(:)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N}$ are vectors of size R with elements $y^{(r)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N}$ and $x^{(r)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N}$ in $\mathcal{Y}^{(r)}$ and $\mathcal{X}^{(r)}$, for all $r \in \mathbb{N}_R$. We call this operation as tensor-times-multiple-vector multiplication (TTMV) as R column vectors of $\mathbf{V}$ are multiplied simultaneously with R tensors of identical nonzero patterns. We can similarly extend this formula to the multiplication $\mathcal{Z}^{(r)} \leftarrow \mathcal{Y}^{(r)} \times_{d_2} \mathbf{W} = (\mathcal{X}^{(r)} \times_{d_1} \mathbf{v}_r) \times_{d_2} \mathbf{w}_r$ in two modes d_1 and d_2, $d_1 < d_2$, with matrices $\mathbf{V} \in \mathbb{R}^{I_{d_1} \times R}$ and $\mathbf{W} \in \mathbb{R}^{I_{d_2} \times R}$ as

$$\begin{aligned}\mathbf{z}^{(:)}_{i_1,\dots,i_{d_1-1},1,i_{d_1+1},\dots,i_{d_2-1},1,i_{d_2+1},\dots,i_N} &= \sum_{j_2=1}^{I_{d_2}} \mathbf{W}(j,:) * \mathbf{y}^{(:)}_{i_1,\dots,i_{d_1-1},1,i_{d_1+1},\dots,i_{d_2-1},j_2,i_{d_2+1},\dots,i_N} \\ &= \sum_{(j_1,j_2)=(1,1)}^{(I_{d_1},I_{d_2})} \mathbf{V}(j_1,:) * \mathbf{W}(j_2,:) * \mathbf{x}^{(:)}_{i_1,\dots,i_{d_1-1},j_1,i_{d_1+1},\dots,i_{d_2-1},j_2,i_{d_2+1},\dots,i_N}.\end{aligned} \tag{7}$$

The formula similarly generalizes to d dimensions for $d > 2$ where each dimension adds another Hadamard product with a corresponding matrix row. This "thick" mode of operation provides a significant performance gain thanks to the increase in locality. Also, performing TTVs in this manner in CP-ALS effectively reduces the $RN\lceil \log N \rceil$ TTVs required in Theorem 1 to $N\lceil \log N \rceil$ TTMV calls within an iteration. In our approach, for each node $t \in \mathfrak{T}$, we store a single set $\mathcal{I}_t$ containing index tuples of the form $(i_1, \ldots, i_N)$ representing the nonzeros $x^{(r)}_{i_1,\ldots,i_N} \in T_r(t)$ for all $r \in \mathbb{N}_R$. Also, for each such index tuple we hold a vector of size R corresponding to the values of this nonzero in each one of R tensors, and we denote this value vector as $\mathcal{V}_t(i_1, \ldots, i_N)$.

The sparsity of input tensor $\mathcal{X}$ determines the sparsity of the tensors of at the nodes of the dimension tree. Computing this sparsity structure at each TTV can get very expensive, and is redundant. As $\mathcal{X}$ stays fixed, we can compute the sparsity of each tensor once, and reuse it in all CP-ALS iterations. To this end, we need a data structure that can express this sparsity, while exposing parallelism to update the tensor elements in numerical TTV computations. We now describe computing this data structure in what we call the *symbolic TTV* step.

4.1.1 Symbolic TTV

For simplicity, we proceed with describing how to perform the symbolic TTV using the same 4-dimensional tensor $\mathcal{X}$ and BDT. However, the approach naturally generalizes to any N-dimensional tensor and dimension tree.

The first information we need is the set of nonzero indices $\mathcal{I}_t$ for each node t in a dimension tree, which we determine as follows. As mentioned, the two children t_{12} and t_{34} of t_{1234} have the corresponding tensors $T_r(t_{12}) = T_r(t_{1234}) \times_3 \mathbf{u}^{(3)}_r \times_4 \mathbf{u}^{(4)}_r$ and $T_r(t_{34}) = T_r(t_{1234}) \times_1 \mathbf{u}^{(1)}_r \times_2 \mathbf{u}^{(2)}_r$, respectively. Using (7), the nonzero indices of $T_r(t_{12})$ and $T_r(t_{34})$ take the form $(i, j, 1, 1)$ and $(1, 1, k, l)$, respectively (in practice, tensor indices in the multiplied dimensions are always 1; hence we omit storing them), and such a nonzero index exist in these tensors only if there exists a nonzero $x_{i,j,k,l} \in T_r(t_{1234})$. Determining the set $\mathcal{I}_{t_{12}}$ (or $\mathcal{I}_{t_{34}}$) can simply be done by starting with a list $\mathcal{I}_{t_{1234}}$ of tuples, then replacing each tuple (i, j, k, l) in the list with the tuple (i, j) (or with (k, l) for $\mathcal{I}_{t_{34}}$). This list may contain duplicates, which can efficiently be eliminated by sorting. Once the set of nonzeros for t_{12} (or t_{34}) is determined, we proceed to detect the nonzero patterns of its children t_1 and t_2 (or, t_3 and t_4).

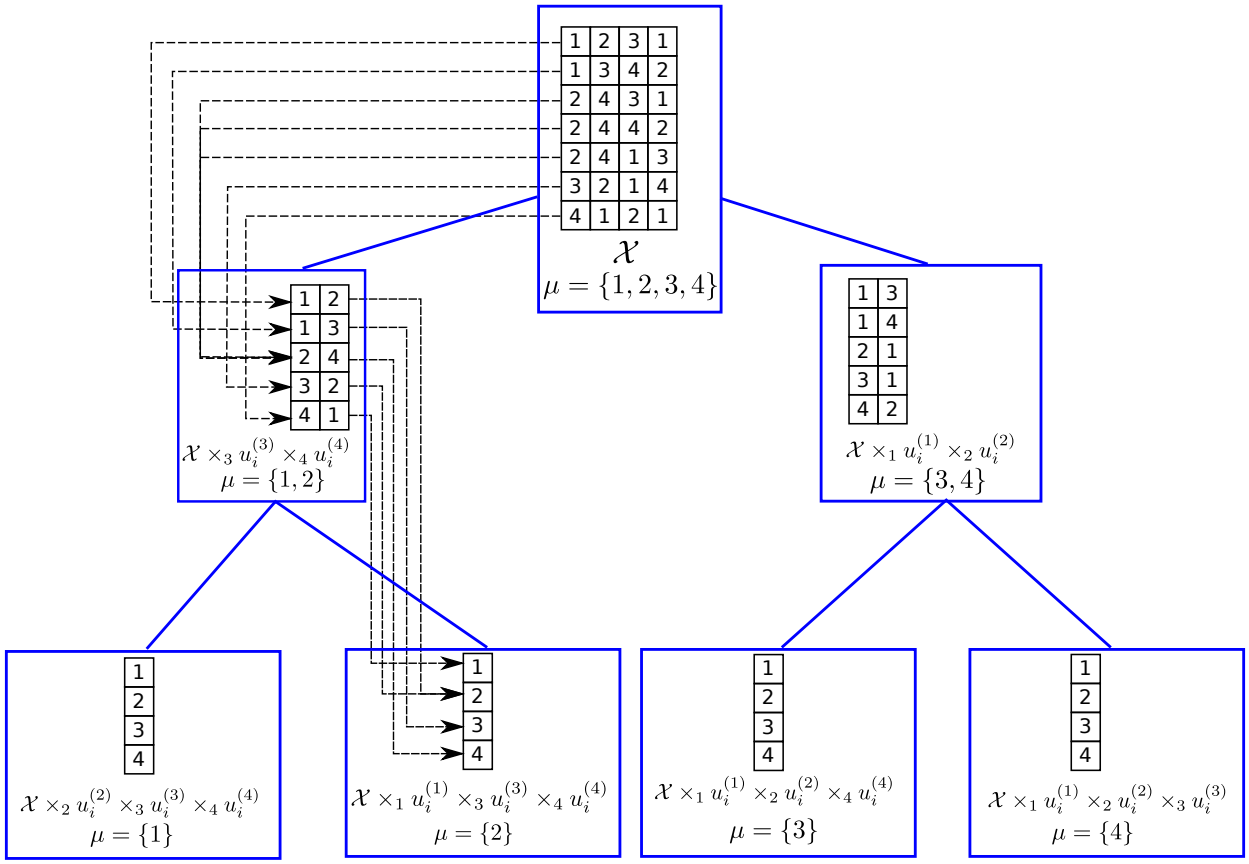
To be able to carry out the numerical calculations (6) and (7) for each nonzero at a node t, we need to identify the set of nonzeros of the parent node $P(t)$'s tensors that contribute to this nonzero. Specifically, at t_{12}, for each nonzero index $(i, j) \in \mathcal{I}_{t_{12}}$ we need to bookmark all nonzero index tuples of t_{1234} of the form $(i, j, k, l) \in \mathcal{I}_{t_{1234}}$, as it is this set of nonzeros of $T_r(t_{1234})$ that contribute to the nonzero of $T_r(t_{12})$ with index $(i, j, 1, 1)$ in (7). Therefore, for each such index tuple of t_{12} we need a *reduction set* $\mathcal{R}_{t_{12}}(i, j)$ which keeps track of all such index tuples of the parent. We determine these sets simultaneously with $\mathcal{I}_t$. In Fig. 1, we illustrate a sample a BDT for a 4-dimensional sparse tensor with $\mathcal{I}_t$, shown with arrays, and $\mathcal{R}_t$, shown using arrows.

Next, we provide the following theorem to help us analyze the computational and the memory cost of symbolic TTV using a BDT for sparse tensors.

Lemma 1. *Let $\mathcal{X}$ be an N-mode sparse tensor. The total number of index arrays in a BDT of $\mathcal{X}$ is at most $N(\lceil \log N \rceil + 1)$.*

Proof. Each node t in the dimension tree holds an index array for each mode in its mode set $\mu(t)$.

Figure 1 – BDT of a 4-dimensional sparse tensor $\mathcal{X} \in \mathbb{R}^{4\times4\times4\times4}$ having 7 nonzeros. Each closed box refers to a tree node. Within each node, the index array and the mode set corresponding to that node are given. The reduction sets of two nodes in the tree are indicated with the dashed lines.



As stated in the proof of Theorem 1, the total size of mode sets at each tree level is at most N. Therefore, the total number of index arrays cannot exceed $(\lceil \log N \rceil + 1)N$ in a BDT. □

Lemma 1 shows that the storage requirement for the tensor indices of a BDT cannot exceed $(\lceil \log N \rceil + 1)$-times the size of the original tensor in the coordinate format (which has N index arrays of size $nnz(\mathcal{X})$). This renders the approach very suitable for higher dimensional tensors. In computing the symbolic TTV, we sort $\mu(t)$ index arrays for each node $t \in \mathcal{T}$. In addition, for each node t of a BDT, we sort an extra array to determine the reduction set $\mathcal{R}_t$. In the worst case, each array can have up to $nnz(\mathcal{X})$ elements. Therefore, combining with the number of index arrays as given in Lemma 1, the overall worst case cost of sorting becomes $O(N(\lceil \log N \rceil + 1) + 2N - 1)nnz(\mathcal{X}) \log(nnz(\mathcal{X})) = O(N \log N \; nnz(\mathcal{X}) \log(nnz(\mathcal{X}))$. We note, however, that both the total index array size and sorting cost are pessimistic overestimates, since the nonzero structure of real-world tensors exhibits significant locality in indices. For example, on two tensors from our experiments (Delicious and Flickr), we observed a reduction factor of 2.57x and 5.5x in the number of nonzeros of the children of the root of the BDT. Consequently, the number of nonzeros in a node's tensors reduces dramatically as we approach to the leaves. In comparison, existing approaches [36] sort the original tensor once with a cost of $O(Nnnz(\mathcal{X}) \log(nnz(\mathcal{X})))$ at the expense of computing TTVs from scratch in each CP-ALS iteration.

Symbolic TTV is a one-time computation whose cost is amortized. Normally, choosing an appropriate rank R for a sparse tensor $\mathcal{X}$ requires several executions of CP-ALS. Also, CP-ALS is known to be sensitive to the initialization of the factor matrices; therefore, it is often executed with multiple initializations [28]. In all of these use cases, the tensor $\mathcal{X}$ is fixed; therefore, the symbolic TTV is required only once. Moreover, CP-ALS usually have a number of iterations which involve many costly numeric TTV calls. As a result, the cost of the subsequent numeric TTV calls over many iterations and many CP-ALS executions easily amortizes that of this symbolic preprocessing. Nevertheless, in case of need, this step can efficiently be parallelized in multiple ways.First, symbolic TTV is essentially a sorting of multiple index arrays; hence, one can use parallel sorting methods. Second, the BDT structure naturally exposes a coarser level of parallelism; once a node's symbolic TTV is computed, one can proceed with those of its children in parallel, and process the whole tree in this way. Finally, in a distributed memory setting where we partition the tensor to multiple processes, each process can perform the symbolic TTV on its local tensor in parallel. We benefit only from the this parallelism in our implementation.

4.1.2 Shared memory parallelism

After forming the dimension tree $\mathcal{T}$ for a tensor $\mathcal{X}$ with symbolic structures $\mathcal{I}_t$ and $\mathcal{R}_t$ for all tree nodes, we can perform numeric TTV computations in parallel. In Algorithm 4, we provide the shared memory parallel TTV algorithm, called Smp-Dtree-Ttv, for a node t of a dimension tree. The goal of Smp-Dtree-Ttv is to compute the tensor values $\mathcal{V}_t$ for a given node t. Similar to Algorithm 2, it starts by checking if $\mathcal{V}_t$ is already computed, and returns immediately in that case. Otherwise, it calls Smp-Dtree-Ttv on the parent node $P(t)$ to make sure that parent's tensor values $\mathcal{V}_{P(t)}$ are available. Once $\mathcal{V}_{P(t)}$ is ready, the algorithm proceeds with computing $\mathcal{V}_t$ for each nonzero index $(i_1, \ldots, i_N) \in \mathcal{I}_t$. As for each index $(i_1, \ldots, i_N) \in \mathcal{I}_t$, the reduction set is defined during the symbolic TTV, the $\mathcal{V}_t$ can independently be updated in parallel. In performing this update, for each element $\mathcal{V}_{P(t)}(j_1, \ldots, j_N)$ of the parent, the algorithm multiplies this vector

Algorithm 4 SMP-DTREE-TTV

Input: t: A dimension tree node/tensor
Output: Numerical values $\mathcal{V}_t$ are computed.

```
if EXISTS(V_t) then
    return                                          ▶ Numerical values V_t are already computed.
SMP-DTREE-TTV(P(t))                                 ▶ Compute the parent's values V_P(t) first.
parallel for all (i_1, ..., i_N) ∈ I_t do           ▶ Compute each V_t(i_1, ..., i_N) in parallel.
    V_t(i_1, ..., i_N) ← zeros(1, R)                ▶ Initialize with a zero vector of size 1 × R.
    for all (j_1, ..., j_N) ∈ R(i_1, ..., i_N) do   ▶ Perform updates using elements in R.
        r ← V_P(t)(j_1, ..., j_N)
        for all d ∈ δ(t) do                         ▶ Multiply the element with corresponding matrix rows.
            r ← r * U^(d)(j_d, :)
        V_t(i_1, ..., i_N) ← V_t(i_1, ..., i_N) + r ▶ Do the update due to element (j_1, ..., j_N).
```

with the rows of the corresponding matrices of the TTV in the multiplication modes $\delta(t)$ of t, then adds it to $\mathcal{V}_t(i_1, \ldots, i_N)$.

For shared-memory CP-ALS, we replace Line 9 of Algorithm 3 with SMP-DTREE-TTV($\mathcal{L}_n$). The parallelization of the rest of the computations is trivial. In computing the matrices $\mathbf{W}^{(n)}$ and $\mathbf{U}^{(n)}$ at Lines 3,13, and 15, we use parallel dense BLAS kernels. Computing the matrix $\mathbf{H}^{(n)}$ at Line 12 and normalizing the columns of $\mathbf{U}^{(n)}$ are embarrassingly parallel element-wise matrix operations. We skip the details of the parallel convergence check whose cost is negligible.

4.2 Distributed memory parallelism

Parallelizing CP-ALS in a distributed memory setting involves defining unit parallel tasks, data elements, and their interdependencies. Following to this definition, we partition and distribute tensor elements as well as factor matrices to all available processes. We provide a *fine-grain* and a *medium-grain* parallel task model together with the associated distributed memory parallel algorithms. We start the discussion with the following theorem that enable us to distribute tensor nonzeros for parallelization.

Lemma 2 (Distributive property of tensor-times-vector multiplies)**.** *Let $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ be tensors in $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with $\mathcal{X} = \mathcal{Y} + \mathcal{Z}$. Then, for any $n \in \mathbb{N}_N$ and $\boldsymbol{u} \in \mathbb{R}^{I_n}$ $\mathcal{X} \times_n \boldsymbol{u} = \mathcal{Y} \times_n \boldsymbol{u} + \mathcal{Z} \times_n \mathbf{u}$ holds.*

Proof. Using (1) we express the element-wise result of $\mathcal{X} \times_n \mathbf{u}$ as

$$
\begin{aligned}
(\mathcal{X} \times_n \mathbf{u})_{i_1,\ldots,1,\ldots,i_N} &= \sum_{j=1}^{I_n} u_j (x_{i_1,\ldots,j,\ldots,i_N} - z_{i_1,\ldots,j,\ldots,i_N} + z_{i_1,\ldots,j,\ldots,i_N}) \\
&= \sum_{j=1}^{I_n} u_j (x_{i_1,\ldots,j,\ldots,i_N} - z_{i_1,\ldots,j,\ldots,i_N}) + \sum_{j=1}^{I_n} u_j z_{i_1,\ldots,j,\ldots,i_N} \\
&= (\mathcal{Y} \times_n \mathbf{u})_{i_1,\ldots,1,\ldots,i_N} + (\mathcal{Z} \times_n \mathbf{u})_{i_1,\ldots,1,\ldots,i_N}
\end{aligned}
$$

□

By extending the previous lemma to P summands and all but one mode TTV, we obtain the next corollary.

Corollary 1. *Let $\mathcal{X}$ and $\mathcal{X}_1, \ldots, \mathcal{X}_P$ be tensors in $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with $\sum_{i=1}^{P} \mathcal{X}_i = \mathcal{X}$. Then, for any $n \in \mathbb{N}_N$ and $\boldsymbol{u}_i \in \mathbb{R}^{I_i}$ for $i \in \mathbb{N}_N \setminus \{n\}$ we obtain $\textsc{Ttv}(\mathcal{X}, \{\boldsymbol{u}_i\}_{i \neq n}) = \textsc{Ttv}(\mathcal{X}_1, \{\boldsymbol{u}_i\}_{i \neq n}) + \cdots + \textsc{Ttv}(\mathcal{X}_P, \{\boldsymbol{u}_i\}_{i \neq n})$, where $\textsc{Ttv}(\mathcal{X}, \{\boldsymbol{u}_i\}_{i \neq n})$ is "all-but-one-mode" TTV of $\mathcal{X}$ defined as $\mathcal{X} \times_1 \boldsymbol{u}_1 \times_2 \cdots \times_{n-1} \boldsymbol{u}_{n-1} \times_{n+1} \boldsymbol{u}_{n+1} \times_{n+2} \cdots \times_N \boldsymbol{u}_N$.*

Proof. Multiplying the tensors $\mathcal{X}$ and $\mathcal{X}_1, \ldots, \mathcal{X}_P$ in any mode $n' \neq n$ in the equation gives tensors $\mathcal{X}' = \mathcal{X} \times_{n'} \mathbf{u}$ and $\mathcal{X}'_i = \mathcal{X}_i \times_{n'} \mathbf{u}$, and $\mathcal{X}' = \sum_{i=1}^{P} \mathcal{X}'_i$ holds by the distributive property. The same process is repeated in the remaining modes to obtain the desired result. □

4.2.1 Fine-grain parallelism

Corollary 1 allows us to partition the tensor $\mathcal{X}$ in the sum form $\mathcal{X}_1 + \cdots + \mathcal{X}_P$ for any $P > 1$, then perform TTVs on each tensor partition $\mathcal{X}_p$ independently, finally sum up these results to obtain the TTV result for $\mathcal{X}$ multiplied in $N-1$ modes. As $\mathcal{X}$ is sparse, an intuitive way to achieve this decomposition is by partitioning its nonzeros to P tensors where P is the number of available distributed processes. This way, for any dimension n, we can perform the TTV of $\mathcal{X}_p$ with the columns of the set of factor matrices $\{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\}$ in all modes except n. This yields a "local" matrix $\mathbf{M}_p^{(n)}$ at each process p, and all these local matrices must subsequently be "assembled" by summing up their rows corresponding to the same row indices. In order to perform this assembling of rows, we also partition the rows of matrices $\mathbf{M}^{(n)}$ so that each row is "owned" by a process that is responsible for holding the final row sum. We represent this partition with a vector $\boldsymbol{\sigma}^{(n)} \in \mathbb{R}^{I_n}$ where $\sigma^{(n)}(i) = p$ implies that the final value of $\mathbf{M}^{(n)}(i,:)$ resides at the process p. We assume the same partition $\boldsymbol{\sigma}^{(n)}$ on the corresponding factor matrices $\mathbf{U}^{(n)}$, as this enables each process to compute the rows of $\mathbf{U}^{(n)}$ that it owns using the rows of $\mathbf{M}^{(n)}$ belonging to that process without incurring any communication.

This approach amounts to a fine-grain parallelism where each fine-grain computational task corresponds to performing TTV operations due to a nonzero element $x_{i_1,\ldots,i_N} \in \mathcal{X}$. Specifically, according to (7), the process p needs the matrix rows $\mathbf{U}^{(1)}(i_1,:), \ldots, \mathbf{U}^{(N)}(i_N,:)$ for each nonzero $x_{i_1,\ldots,i_N}$ in its local tensor $\mathcal{X}_p$ in order to perform its local TTVs. For each dimension n, we represent the union of all these "required" row indices for the process p by $F_p^{(n)}$. Similarly, we represent the set of "owned" rows by the process p by $I_p^{(n)}$. In this situation, the set $F_p^{(n)} \setminus I_p^{(n)}$ correspond to the rows of $\mathbf{M}^{(n)}$ for which the process p generates a partial TTV result, which need to be sent to their owner processes. Equally, it represents the set of rows of $\mathbf{U}^{(n)}$ that are not owned by the process p and are needed in its local TTVs according to (7). These rows of $\mathbf{U}^{(n)}$ are similarly to be received from their owners in order to carry out the TTVs at process p. Hence, a "good" partition in general involves a significant overlap of $F_p^{(n)}$ and $I_p^{(n)}$ to minimize the cost of communication.

In Algorithm 5, we describe the fine-grain parallel algorithm that operates in this manner at process p. The elements $\mathcal{X}_p$, $I_p^{(n)}$, and $F_p^{(n)}$ are determined in the partitioning phase, and are provided as input to the algorithm. Each process starts with the subset $F_p^{(n)}$ of rows of each factor matrix $\mathbf{U}^{(n)}$ that it needs for its local computations. Similar to Algorithm 3, at Line 1 we start by forming the dimension tree for the local tensor $\mathcal{X}_p$. We then compute the matrices $\mathbf{W}^{(n)}$ corresponding to $\mathbf{U}^{(n)T}\mathbf{U}^{(n)}$ using the initial factor matrices. We do this step in parallel in which each process computes the local contribution $[\mathbf{U}^{(n)}(I_p^{(n)},:)]^T\ \mathbf{U}^{(n)}(I_p^{(n)},:)$ due to its owned rows. Afterwards, we perform an All-Reduce communication to sum up these local results to

Algorithm 5 DMP-DTREE-CP-ALS: Dimension tree-based CP-ALS algorithm

Input: $\mathcal{X}_p$: An N-mode tensor
$I_p^{(n)}$: The index set with elements $i \in I_p^{(n)}$ where $\sigma^{(n)}(i) = p$
$F_p^{(n)}$: The set of all unique indices of $\mathcal{X}_p$ in mode n
$\mathbf{U}^{(1)}(F_p^{(1)}, :), \ldots, \mathbf{U}^{(N)}(F_p^{(N)}, :)$: Distributed initial matrices (rows needed by process p)
R: The rank of CP decomposition
Output: $[\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\mathcal{X}_p$ with distributed $\mathbf{U}^{(n)}$

$\mathcal{T} = \{\mathcal{L}_1, \ldots, \mathcal{L}_N, \mathcal{I}_1, \ldots\} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\mathcal{X}_p$)
for $n = 2 \ldots N$ **do**
  $\mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($[\mathbf{U}^{(n)}(I_p^{(n)}, :)]^T\ \mathbf{U}^{(n)}(I_p^{(n)}, :)$)
repeat
  for $n = 1 \ldots N$ **do**
    for all $t \in \mathcal{T}$ **do** ▶ Invalidate tree tensors that are multiplied in mode n.
      if $n \in \gamma(t)$ **then**
        DESTROY($\mathcal{V}_t$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
    SMP-DTREE-TTV($\mathcal{L}_n$) ▶ Perform the TTVs for the leaf node tensors.
    $\mathbf{M}^{(n)}(F_p^{(n)}, :) \leftarrow \mathcal{V}_{\mathcal{L}_n}(:)$ ▶ Form $\mathbf{M}^{(n)}$ using leaf tensors (done implicitly).
    COMM-FACTOR-MATRIX($\mathbf{M}^{(n)}$, "*fold*", $F_p^{(d)}, I_p^{(d)}, \sigma^{(d)}$) ▶ Assemble $\mathbf{M}^{(n)}(I_p^{(n)}, :)$.
    $\mathbf{H}^{(n)} \leftarrow \mathbf{W}^{(N)} * \cdots * \mathbf{W}^{(n+1)} * \mathbf{W}^{(n-1)} * \cdots * \mathbf{W}^{(1)}$
    $\mathbf{U}^{(n)}(I_p^{(n)}, :) \leftarrow \mathbf{M}^{(n)}(I_p^{(n)}, :)\mathbf{H}^{(n)\dagger}$
    $\lambda \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
    COMM-FACTOR-MATRIX($\mathbf{U}^{(n)}$, "*expand*", $F_p^{(d)}, I_p^{(d)}, \sigma^{(d)}$) ▶ Send/Receive $\mathbf{U}^{(n)}$.
    $\mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($[\mathbf{U}^{(n)}(I_p^{(n)}, :)]^T\ \mathbf{U}^{(n)}(I_p^{(n)}, :)$)
until converge is achieved or the maximum number of iterations is reached.

obtain a copy of $\mathbf{W}^{(n)}$ at each process. The cost of this communication is typically negligible as $\mathbf{W}^{(n)}$ is a small matrix of size $R \times R$. The main CP-ALS subiteration for mode n begins with destroying tensors in the tree that will become invalid after updating $\mathbf{U}^{(n)}$. Next, we perform

Algorithm 6 COMM-FACTOR-MATRIX: Communication routine for factor matrices

Input: $\mathbf{M}$: Distributed factor matrix to be communicated
comm : The type of communication. "*fold*" sums up the partial results in owner processes, whereas "*expand*" communicates the final results from owners to all others.
F_p: The rows used by process p
I_p: The rows owned by process p
σ: The ownership of each row of $\mathbf{M}$
Output: Rows of $\mathbf{M}$ are properly communicated.

```
if comm = "fold" then
    for all i ∈ F_p \ I_p do                    ▶ Send non-owned rows to their owners.
        Send M(i,:) to the process σ(i).
    for all i ∈ I_p do                          ▶ Gather all partial results of owned rows together.
        Receive and sum up all partial results for M(i,:).
else if comm = "expand" then
    for all i ∈ I_p do                          ▶ Send owned rows to all processes in need.
        Send M(i,:) to the all processes p' with i ∈ F_p'.
    for all i ∈ F_p \ I_p do                    ▶ Receive rows that are needed locally.
        Receive M(i,:) from the process σ(i).
```

SMP-DTREE-TTV on the leaf node $\mathcal{L}_n$, and obtain the "local" matrix $\mathbf{M}^{(n)}$. Then, the partial results for the rows of $\mathbf{M}^{(n)}$ are communicated to be assembled at their owner processes. We name this as the *fold* communication step following the convention from the fine-grain parallel sparse matrix computations. Afterwards, we form the matrix $\mathbf{H}^{(n)}$ locally at each process p in order to compute the owned part $\mathbf{U}^{(n)}(I_p^{(n)},:)$ using the recently assembled $\mathbf{M}^{(n)}(I_p^{(n)},:)$. Once the new distributed $\mathbf{U}^{(n)}$ is computed, we normalize it column-wise and obtain the vector λ of norms. The computational and the communication costs of this step are negligible. The new $\mathbf{U}^{(n)}$ is finalized after the normalization, and we then perform an *expand* communication step in which we send the rows of $\mathbf{U}^{(n)}$ from the owner processes to all others in need. This is essentially the inverse of the *fold* communication step in the sense that each process p that sends a partial row result of $\mathbf{M}^{(n)}(i,:)$ to another process q in the *fold* step receives the final result for the corresponding row $\mathbf{U}^{(n)}(i\ :)$ from the process q in the *expand* communication. Finally, we update the matrix $\mathbf{W}^{(n)}$ using the new $\mathbf{U}^{(n)}$ in parallel.

The expand and the fold communications at Lines 11 and 15 constitute the most expensive communication steps. We outline these communications in Algorithm 6. In the expand communication, the process p sends the partial results for the set $F_p \setminus I_p$ of rows to their owner processes, while similarly receiving all partial results for its set I_p of owned rows and summing them up. Symmetrically, in the fold communication, the process p sends the rows with indices I_p, and receives the rows with indices $F_p \setminus I_p$. The exact set of row indices that needs to be communicated in fold and expand steps depends on the partitioning of $\mathcal{X}$ and the factor matrices. As this partition does not change once determined, the communicated rows between p and q stays the same in CP-ALS iterations. Therefore, in our implementation we determine this row set once outside the main CP-ALS iteration, and reuse it at each iteration.

4.2.2 Medium-grain parallelism

For an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and using $P = \prod_{i=1}^{N} P_i$ processes, the medium-grain decomposition imposes a partition with $P_1 \times \cdots \times P_N$ Cartesian topology on the dimensions of $\mathcal{X}$. Specifically, for each dimension n, the index set $\mathbb{N}_{I_n}$ is partitioned into P_n sets $\mathcal{I}_1^{(n)}, \ldots, \mathcal{I}_{P_n}^{(n)}$. With this partition, the process with the index $(p_1, \ldots, p_N) \in P_1 \times \cdots \times P_N$ gets $\mathcal{X}(\mathcal{I}_{p_1}^{(1)}, \ldots, \mathcal{I}_{p_N}^{(n)})$ as its local tensor. Each factor matrix $\mathbf{U}^{(n)}$ is also partitioned following this topology where the set of rows $\mathbf{U}^{(n)}(\mathcal{I}_j^{(n)}, :)$ is owned by the processes with index $(p_1, \ldots, p_N)$ where $j \in \mathbb{S}_{P_N}$ and $p_n = j$, even though these rows are to be further partitioned among the processes having $p_n = j$. As a result, one advantage of the medium-grain partition is that only the processes with $p_n = j$ need to communicate with each other in mode n. This does not necessarily reduce the volume of communication, but it can reduce the number of communicated messages by a factor of P_n in the nth dimension.

One can design an algorithm specifically for the medium-grain decomposition [37]. However, using the fine-grain algorithm on a medium-grain partition effectively provides a medium-grain algorithm. For this reason, we do not need nor provide a separate algorithm for the medium-grain task model, and use the fine-grain algorithm with a proper medium-grain partition instead, which equally benefits from the topology.

4.2.3 Partitioning

The distributed memory algorithms that we described require partitioning the data and the computations, as in any distributed memory algorithm. In order to reason about their computational load balance and communication cost, we use hypergraph models. Once the models are built, different hypergraph partitioning methods can be used to partition the data and the computations. We discuss a few partitioning alternatives.

4.2.4 Partitioning for the fine-grain parallelism

We propose a hypergraph model (see Appendix A for the standard definitions) for the computational load and the communication volume of the fine-grain parallelization given in Algorithm 5. For the simplicity of the discussion, we present the model for a 3rd order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and factor matrices $\mathbf{U}^{(1)} \in \mathbb{R}^{I_1 \times R}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{I_2 \times R}$, and $\mathbf{U}^{(3)} \in \mathbb{R}^{I_3 \times R}$. For these inputs, we construct a hypergraph $H = (V, E)$ with the vertex set V and the hyperedge set E. The generalization of the model to higher order tensors should be clear from this construction.

The vertex set $V = V^{(1)} \cup V^{(2)} \cup V^{(3)} \cup V^{(\mathcal{X})}$ of the hypergraph involves four types of vertices. The first three types correspond to the rows of the matrices $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$, and $\mathbf{U}^{(3)}$. In particular, we have vertices $v_i^{(1)} \in V^{(1)}$ for $i \in \mathbb{N}_{I_1}$, $v_j^{(2)} \in V^{(2)}$ for $j \in \mathbb{N}_{I_2}$, and $v_k^{(3)} \in V^{(3)}$ for $k \in \mathbb{N}_{I_3}$. These vertices represent the "ownership" of the corresponding matrix rows, and we assign unit weight to each such vertex. The fourth type of vertices are denoted by $v_{i,j,k}^{(\mathcal{X})}$, which we define for each nonzero $x_{i,j,k} \in \mathcal{X}$. This vertex type relates to the TTV operations performed due to the nonzero element $x_{i,j,k} \in \mathcal{X}$ in performing MTTKRP using (7) in all modes. In the N-dimensional case, this includes up to N Hadamard products involving the value of the nonzero $x_{i,j,k}$, and the corresponding matrix rows. The exact number of performed Hadamard products depends on how nonzero indices coincide as TTVs and cannot be determined before a partitioning takes place. In our earlier work [25], this cost was exactly N Hadamard products per nonzero, as the MTTKRPs were computed without

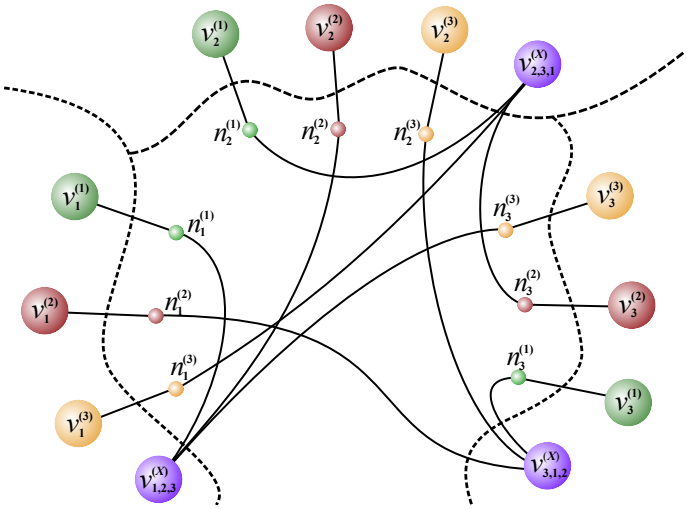


Figure 2 – Fine-grain hypergraph model for the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$ and a 3-way partition of the hypergraph. The objective is to minimize the cutsize of the partition while maintaining a balance on the total part weights corresponding to each vertex type (shown with different colors).

reusing partial results. In the current case, we assign a cost of N to each vertex $v_{i,j,k}^{(\mathcal{X})}$ to represent an upper bound on the computational cost, and expect this to lead load balance in practice. With these vertex definitions, one can use multi-constraint partitioning (10). Balancing the first three types corresponds to balancing the number of matrix rows at each process (which provides the memory balance as well as the computational balance in dense matrix operations), whereas balancing the fourth type corresponds to balancing the computational load due to TTVs.

As TTVs are carried out using (7), data dependencies to the rows of $\mathbf{U}^{(1)}(i,:)$, $\mathbf{U}^{(2)}(j,:)$, and $\mathbf{U}^{(3)}(k,:)$ take place when performing Hadamard products due to each nonzero $x_{i,j,k}$. We introduce three types of hyperedges in $E = E^{(1)} \cup E^{(2)} \cup E^{(3)}$ to represent these dependencies as follows: $E^{(1)}$ contains a hyperedge $n_i^{(1)}$ for each matrix row $\mathbf{U}^{(1)}(i,:)$, $E^{(2)}$ contains a hyperedge $n_j^{(2)}$ for each row $\mathbf{U}^{(2)}(j,:)$, and $E^{(3)}$ contains a hyperedge $n_k^{(3)}$ for each row $\mathbf{U}^{(3)}(k,:)$. Initially, $n_i^{(1)}$, $n_j^{(2)}$ and $n_k^{(3)}$ contain the corresponding vertices $v_i^{(1)}$, $v_j^{(2)}$, and $v_k^{(3)}$, as the owner of a matrix row has a dependency to it by default. In computing the MTTKRP using (7), each nonzero $x_{i,j,k}$ requires access to $\mathbf{U}^{(1)}(i,:)$, $\mathbf{U}^{(2)}(j,:)$, and $\mathbf{U}^{(3)}(k,:)$. Therefore, we add the vertex $v_{i,j,k}^{(\mathcal{X})}$ to the hyperedges $n_i^{(1)}$, $n_j^{(2)}$ and $n_k^{(3)}$ to model this dependency. In Fig. 2, we demonstrate this fine-grain hypergraph model on a sample tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$, yet we exclude the vertex weights for clarity. Each vertex type and hyperedge type is shown using a different color in the figure.

Consider now a P-way partition of the vertices of $H = (V,E)$ where each part is associated with a unique process to obtain a P-way parallel Algorithm 5. We consider the first subiteration of Algorihm 5 that updates $\mathbf{U}^{(1)}$, and assume that each process already has all data to carry out the local TTVs at Line 9. Now suppose that the nonzero $x_{i,j,k}$ is owned by the process p and the matrix row $\mathbf{U}^{(1)}(i,:)$ is owned by the process q. Then, the process p computes a partial result for $\mathbf{M}^{(1)}(i,:)$ which needs to be sent to the process q at Algorithm 3 of Algorithm 6. By construction

of the hypergraph, we have $v_i^{(1)} \in n_i^{(1)}$ which resides at the process q, and due to the nonzero $x_{i,j,k}$ we have $v_{i,j,k}^{(\mathfrak{X})} \in n_i^{(1)}$ which resides at the process p; therefore, this communication is accurately represented in the *connectivity* $\lambda_{n_i^{(1)}}$ of the hyperedge $n_i^{(1)}$. In general, the hyperedge $n_i^{(1)}$ incurs $\lambda_{n_i^{(1)}} - 1$ messages to transfer the partial results for the matrix row $\mathbf{M}^{(1)}(i,:)$ to the process q at Algorithm 3. Therefore, the *connectivity-1 cutsize* metric (9) over the hyperedges exactly encodes the total volume of messages sent at Algorithm 3, if we set $\mathsf{c}[\cdot] = R$. Since the send operations at Algorithm 8 are duals of the send operations at Algorithm 3, the total volume of messages sent at Algorithm 8 for the first mode is also equal to this number. By extending this reasoning to all other modes, we obtain that the cumulative (over all modes) volume of communication in one iteration of Algorithm 5 equals to the connectivity-1 cut-size metric. As the communication due to each mode take place in different stages, one might alternatively use a multi-objective hypergraph model to minimize the communication volume due to each mode (or equivalently, hyperedge type) independently.

As discussed above, the proper model for partitioning the data and the computations for the fine-grain parallelism calls for a multi-constraint and a multi-objective partitioning formulation to achieve the load balance and minimize the communication cost with a single call to a hypergraph partitioning routine. Since these formulations are expensive, we follow a two-step approach. In the first step, we partition only the nonzeros of the tensor on the hypergraph $H = (V^{(\mathfrak{X})}, E)$ using just one load constraint due to the vertices in $V^{(\mathfrak{X})}$, and we thereby avoid multi-constraint partitioning. We also avoid multi-objective partitioning by treating all hyperedge types as the same, and thereby aim to minimize the total communication volume across all dimensions, which works good in practice. Once the nonzero partitioning is settled we partition the rows of the factor matrices in a way to *balance* the communication—which is not achievable using standard partitioning tools—as discussed in Section 4.2.6.

We now discuss three methods for partitioning the described hypergraph.

Random: This approach visits the vertices of the hypergraph and assigns each visited vertex to a part chosen uniformly at random. It is expected to balance the TTV work assigned to each process while ignoring the cost of communication. We use random partitioning only as a "worst case" point of reference for other methods.

Standard: In this standard approach, we feed the hypergraph to a standard hypergraph partitioning tool to obtain balance on the number of tensor nonzeros and the amount of TTV work assigned to a process while minimizing the communication volume. This approach promises significant reductions in communication cost with respect to the others, yet imposes significant computational and memory requirements.

Label propagation-like: Given that the standard partitioning approach is too costly, we developed a fast hypergraph partitioning heuristic which has reasonable memory and computational costs. The method is based on the balanced label-propagation algorithm [35, 41], and includes some additional adaptations to handle hypergraphs. The heuristic starts with an initial assignment of vertices to parts, and then proceeds with multiple passes over the hypergraph. At each pass, the vertices are visited in an order, and are possibly moved to other parts in order to reduce the cutsize while respecting the balance constraints.

For the heuristic to be efficient on hypergraphs, some adaptations are needed. Each pass involves two types of updates. In the first step, each hyperedge chooses a "preferred part" by considering the current part of its vertices. Next, each vertex updates its part according to the

preferred parts of the hyperedges that include the vertex. In both steps, the most dominant part index is chosen for the update. The heuristic runs in linear time on the size of the hypergraph per iteration, and requires a memory of $2|V| + |E| + 4P$. Running the algorithm for a few iterations provides reasonably good partitions. This basic algorithm can have may variants. In one variant, we visit the vertices in an order imposed by an increasing ordering by size of the hyperedges. This variant has an overhead of sorting the hyperedges. In another variant, we reweigh the preference of a hyperedge of size s by the multiplier $(1 - \frac{1}{P})^{s-1}$. This last variant has a memory overhead for storing the weights for efficiency purposes; for each size s, the value $(1 - \frac{1}{P})^{s-1}$ is needed.

4.2.5 Partitioning for the medium-grain parallelism

Similar to the fine-grain model, one can use a hypergraph model for the medium-grain parallel computations to reduce the communication volume using hypergraph partitioners. However, medium-grain variant is analogous to checkerboard partitioning designed for matrices [13, 14], and calls for a multi-constrained partitioning. Specifically, for a 3-dimensional tensor with a process topology $P_1 \times P_2 \times P_3$ where $P = P_1 P_2 P_3$, the hypergraph is to be partitioned in three phases; using one load constraint in the first phase, P_1 constraints (where each constraint is obtained from the first phase) in the second phase, and $P_1 P_2$ constraints (obtained from the second phase partitioning) in the third phase. As P can be large, the number of constraints P_1 and $P_1 P_2$ can similarly get large, and in this case the state of the art partitioners do not perform well both in terms of partition quality and speed. For higher dimensional tensors, this situation only gets worse. That is why explicit communication reduction using hypergraph partitioning for the medium-grain algorithm is not feasible in practice. Hence, we use the partitioning heuristic by Smith and Karypis [37] to partition medium-grain hypergraphs for load balance, and to expect a communication reduction due to partition topology indirectly. We also determine the partition topology by choosing P_1, P_2, and P_3 proportional to the tensor dimensions I_1, I_2, and I_3.

4.2.6 Mode partitioning

Once the nonzero partitioning is obtained for the given fine- or medium-grain parallelism, we proceed with partitioning the mode indices (or, equivalently, the rows of the factor matrices) using a similar heuristic common in similar work [25, 37]. For each matrix row i in dimension n, we identify the processes that have a data dependency to that row. These are exactly the processes which have at least one nonzero with index i in the nth dimension. Next, all row indices are sorted in increasing order of the number of dependent processes. Finally, each row is greedily assigned to the process having the minimum total communication volume among all processes dependent to that row.

5 Related work

There has been many recent advances in the efficient computations of tensor factorizations in general, and CP decomposition in particular. In [5], Bader and Kolda show how to efficiently carry out MTTKRP as well as other fundamental tensor operations on sparse tensors in MATLAB. GigaTensor [21] is a parallel implementation of CP-ALS using the Map-Reduce framework. DFacTo [16] is a C++ implementation with distributed memory parallelism using MPI, and it uses a particular formulation of MTTKRP using sparse matrix-vector multiplication. Splatt [37]

Table 1 – Real-world tensors used in the experiments.

Tensor	I_1	I_2	I_3	I_4	#nonzeros
Delicious	1.4K	532K	17M	2.4M	140M
Flickr	731	319K	28M	1.6M	112M
Netflix	480K	17K	2K	-	100M
NELL	3.2M	301	638K	-	78M
Amazon	6.6M	2.4M	23K	-	1.3B

is an efficient parallelization of MTTKRP and CP-ALS both in shared [38] and distributed memory [37] environments using OpenMP and MPI, and is implemented in C. It uses a medium-grain distributed parallelism with a Cartesian partitioning of the tensor, and generalizes this technique to the tensor completion problem [39]. It is the fastest publicly available CP-ALS implementation in the existing literature, and their approach translates to performing $N(N-1)$ TTMVs in performing the MTTKRP in the main CP-ALS iteration. Karlsson et al. similarly discuss a parallel computation of the tensor completion problem [22]. Baskaran et al. [7] provide a shared-memory parallel implementation for the Tucker decomposition of sparse tensors. Kaya et al. [26] provide efficient shared and distributed memory parallelization of the Tucker decomposition for sparse tensors using OpenMP and MPI. Austin et al. [3] discuss a high performance distributed memory parallelization of dense Tucker factorization in the context of data compression. Finally, Perros et al. [32] perform efficient computation of hierarchical Tucker decomposition for sparse tensors.

6 Experiments

We first investigate how CP-ALS implementations compare using a single thread to assess the algorithmic impact of using a BDT in the same implementation. Then, we compare these implementations using multiple threads to evaluate their shared memory parallel performance. Finally, we investigate the medium- and the fine-grain distributed memory parallel algorithms.

We experimented with five real-world tensors, sizes are shown in Table 1, which are explained in Appendix B. We also used three random tensors with 4, 8, and 16 dimensions. In these random tensors, each dimension is of size 10M and there are 100M nonzeros with a uniform random distribution of indices. Using these tensors, we measure the effect of tensor dimensionality on the performance. The same appendix includes the specifications of the run time environment.

We provide the dimension-tree based CP-ALS implementation in our tensor factorization library called HyperTensor. It is a C++11 implementation providing shared and distributed memory parallelism through OpenMP and MPI libraries. We compared our code against Splatt v1.1.1 [37].

6.1 Shared memory performance

We compare the shared memory performance of the dimension tree-based CP-ALS algorithm with the state of the art. We experiment with three methods called **ht-tree**, **ht-flat**, and **splatt**. The **ht-tree** method, implemented in HyperTensor, uses a BDT to carry out TTVs. The **ht-flat** method is the same implementation as **ht-tree** except that it uses a 2-level flat dimension tree. In this tree, N leaf nodes are directly connected to the root, hence no intermediate results

Table 2 – Sequential CP-ALS run time per iteration (in seconds).

	splatt	ht-flat	ht-tree
Delicious	66.571	59.791	33.560
Flickr	43.556	35.505	22.008
Netflix	8.171	13.551	8.380
NELL	8.271	9.479	6.548
Amazon	214.610	217.428	210.665
Random4D	224.729	157.494	109.326
Random8D	1527.111	734.935	384.386
Random16D	4401.567	3369.697	1115.925

are generated. However, TTVs are performed one mode at a time to benefit from the index compression to reduce the operation count. As a result, this method performs $N-1$ TTVs for each mode in an iteration just as Splatt; we thus expect comparable performance. The method **splatt** corresponds to the parallel CP-ALS implementation in Splatt.

We ran all algorithms for 20 iterations with the rank of approximation $R = 20$, and recorded the average time spent per CP-ALS iteration. In Table 2, we give the sequential per-iteration run time of all three methods. We first note that **splatt** runs slightly faster than **ht-flat** on three dimensional Amazon and NELL tensors (1% and 15%), and notably faster on Netflix tensor (1.65x). This is so because Splatt has a special and tuned implementation for 3-dimensional tensor; whereas we have a single code for all dimensions. On all higher dimensional tensors, **ht-flat** performs significantly better than **splatt**, up to 2.08x on Random8D, which shows the efficiency of our implementation for N-dimensional tensors even before using a BDT.

We now measure the effect of dimension trees by comparing **ht-tree** and **ht-flat** in Table 2. These two methods use the same TTV implementation, where **ht-tree** uses a BDT. On Delicious, Flickr, Netflix, and NELL, **ht-tree** obtains 1.78x, 1.61x, 1.61x, and 1.45x speedups over **ht-flat** thanks to the BDT. Likewise, on random tensors, we observed speedups of 1.44x, 1.91x, and 3.01x on tensors Random4D, Random8D, and Random16D, respectively, using **ht-tree**. This validates our performance expectation (Theorem 1) that as the dimensionality of the tensor increases, a BDT results in significantly less TTVs and better performance. Comparing **ht-tree** with **splatt** similarly yielded speedups of 1.99x, 1.97x, 1.26x, 2.05x, and 3.95x on tensors Delicious, Flickr, NELL, Random4D, Random8D, and Random16D, respectively, which meets our expectation of performance gain from Theorem 1. On Amazon, **ht-tree** was only 1% faster whereas on Netflix **splatt** was only 3% faster, which was the only instance in which **splatt** had a slight edge over **ht-tree**.

In **ht-tree**, the symbolic TTV step took 18%, 14.6%, 31%, 34%, and 21% of the total execution time of CP-ALS on Delicious, Flickr, Netflix, NELL, and Amazon tensors, respectively. On Random4D, Random8D, and Random16D tensors, it took 6.6%, 3.7%, and 4.3% of the total run time. In practice, CP-ALS is executed multiple times with different initial matrices and ranks of approximation using the same symbolic dimension tree construct, which renders this cost even less important.

In Table 3, we give run time results of all methods with shared memory parallelism using 14 threads. We first note that in HyperTensor, using dimension trees consistently yields better

Table 3 – Shared memory parallel CP-ALS run time per iteration (in seconds).

	splatt	ht-flat	ht-tree
Delicious	8.290	8.892	4.141
Flickr	5.812	5.332	3.208
Netflix	0.681	1.243	0.782
NELL	1.329	1.200	0.909
Amazon	24.380	28.453	25.742
Random4D	20.063	21.935	13.647
Random8D	86.903	106.833	39.827
Random16D	349.150	425.796	100.624

execution times. Using **ht-tree**, we obtain 2.14x, 1.66x, 1.59x, 1.32x, and 1.11x speedups over **ht-flat** on Delicious, Flickr, Netflix, NELL, and Amazon tensors, respectively. For random tensors, we get 1.61x, 2.68x, and 4.23x speedups on Random4D, Random8D, and Random16D. Comparing **ht-tree** with **splatt**, we observe speedups of 2x, 1.8x, 1.46x, 1.47x, 2.18x, and 3.47x on tensors Delicious, Flickr, NELL, Random4D, Random8D, and Random16D, respectively. This demonstrates that the use of a BDT in CP-ALS computations can be efficiently parallelized in a shared memory setting (on top of significantly reducing the amount of TTV work in the sequential case). On three dimensional tensors Amazon and Netflix, **splatt** has a slight edge over **ht-tree** by 5% and 14%. Another point to note is that in general **splatt** has somewhat better parallel speedups (over its own sequential run time) than **ht-flat** and **ht-tree**. This is mostly due to the fact that TTV is a memory-bound computation; hence, once the memory bandwidth is fully utilized, one cannot expect further speedups through multi-threading. When performing TTVs, our implementation makes slightly more memory accesses due to extra pointer arrays involved in the dimension tree nodes, which saturates the bandwidth earlier, and in turn somewhat affects the parallel speedups. Nevertheless, using **ht-tree** we achieve up to 3.47x faster runs over **splatt** in a shared memory parallel execution.

6.2 Distributed memory performance

We compare the performance and the scalability of the fine- and the medium-grain parallel CP-ALS algorithms. In these experiments, we do not use Splatt software to benchmark medium-grain parallelization for two reasons. First, we would like to compare the effect of load balance and communication cost in different algorithms using different partitionings, while isolating the effects of the efficiency of local CP-ALS computations. Since Splatt's medium-grain implementation does not use BDT for local TTVs and is slower, comparing it against HyperTensor's fine-grain implementation which has faster local TTVs would not be fair. Second, we were not able to get Splatt work on our distributed system despite our full efforts. Therefore, we instead performed medium-grain partitioning of tensors following the description of Splatt's heuristic [37], and ran HyperTensor on these partitions which incurs the same cost in terms of the communication volume and the number of messages as Splatt, while using more efficient TTV kernels. For local CP-ALS computations, we use the BDT-based method **ht-tree** for shared memory parallelism, as it gives the best performance. This way, the experiments become more precise in terms of

Table 4 – Time spent per iteration (in seconds) for our distributed memory parallel CP-ALS using two threads per core with different partitions.

#nodes×#cores	Delicious			Flickr		
	med-gd	fine-lb	fine-hp	med-gd	fine-lb	fine-hp
8×1	45.078	46.741	45.256	29.870	25.674	26.021
8×16	3.192	3.603	2.487	2.394	2.011	1.512
16×16	2.589	2.297	1.454	1.603	1.091	0.862
32×16	1.612	1.458	0.873	1.120	0.694	0.479
64×16	1.315	0.899	0.539	1.013	0.353	0.286
128×16	0.822	0.533	0.351	0.652	0.222	0.179
256×16	0.603	0.318	0.246	0.554	0.173	0.144

#nodes×#cores	Netflix			NELL		
	med-gd	fine-lb	fine-hp	med-gd	fine-lb	fine-hp
4×1	27.656	28.118	29.336	19.540	20.215	19.967
4×16	1.095	1.191	1.185	1.191	1.205	1.109
8×16	0.617	0.697	0.688	0.749	0.700	0.681
16×16	0.360	0.421	0.420	0.448	0.433	0.444
32×16	0.222	0.267	0.261	0.287	0.282	0.318
64×16	0.138	0.178	0.173	0.179	0.201	0.239
128×16	0.097	0.117	0.102	0.127	0.153	0.155
256×16	0.086	0.117	0.106	0.099	0.124	0.119

#nodes×#cores	Amazon	
	med-gd	fine-lb
64×1	-	36.303
64×16	-	1.874
128×16	1.141	1.007
256×16	1.056	0.570

measuring the influence of medium- and fine-grain algorithms and associated partitionings on parallel scalability.

We investigate the performance with two tables. In Table 4, we give the run time results of the medium- and the fine-grain algorithms up to 256 MPI ranks using 4096 cores. Since we achieved the maximum scalability in most tensors with 256 MPI ranks and 4096 cores, the discussion is mostly confined to this case. Especially, in Table 5, we give the detailed load balance and communication cost metrics just for this case.

In Table 4, we compare the execution of Algorithm 5 with three partitioning methods. The **fine-hp** and **fine-lb** methods correspond, respectively, to the standard hypergraph partitioning and the label-propagation-like heuristic of Section 4.2.4. For **fine-hp**, we used PaToH [12] with the default settings. On Amazon tensor, we could not obtain results for **fine-hp** as the tensor is too big for PaToH. For **fine-lb**, we ran the three alternatives, each for three passes, and chose the partitioning with the smallest cut. The **med-gd** method corresponds to the medium-grain partitioning heuristic [37]. The **fine-rd** method refers to the random partitioning of the fine-grain hypergraph. Due to memory constraints, we were not able to execute Algorithm 5 on a single node, as the original tensors are large. Therefore, for each tensor, we give the results starting from the minimum number of nodes needed, and for the same instance we also give the single threaded results. To measure the overall speedup of the algorithm, we estimate its sequential run time on

this cluster as follows. For each data, we take the fastest single threaded run time among all partitions using the minimum number of required nodes to execute the algorithm. By multiplying this timing with the number of nodes that are used, we obtain an estimated sequential run time. In practice, this is a reasonable estimate when only a few nodes are used, as in this case the cost of communication is very small and near perfect speedups can be observed with a good partition. In passing from one core per node to 16 core per node, we see some speedups over 16 in Table 4; this is because we use two threads per core (see Section B).

In order to be able to discuss the speed up results, we give the number of tensor nonzeros per part, computational load (the number of Hadamard products), communication volume, and the number of messages incurred by these three partitionings, using 256 MPI ranks in Table 5. For the four performance metrics, we give the maximum and the average value observed across all processes. We see in all instances except **fine-hp** on NELL that balancing the number of nonzeros per part gracefully translates into balancing the actual computational load.

As seen in Table 4, using 256 MPI ranks on Delicious, **fine-hp** and **fine-lb** yield 2.5x and 1.9x speedups, respectively, over **med-gd**. We observe in Table 5 that this is due to better minimization of the total and the maximum communication volume. On Flickr, **fine-hp** is 3.9x faster than **med-gd** at 256 MPI ranks with 14.7x and 10.6x less total and maximum communication volume, while **fine-lb** shows a speedup of 3.2x over **med-gd** with 6.4x and 5.4x less total and maximum communication cost. In both tensors, **med-gd** results in about the half of the communication cost metrics than **fine-rd**. In overall, on Delicious **fine-hp** and **fine-lb** obtain 1466x and 1134x projected speedups over 4096 cores, whereas **med-gd** gives 598x speedup for the same tensor. On Flickr, **fine-hp** and **fine-lb** similarly yield 1426x and 1187x projected speedups using 4096 cores, while **med-gd** can achieve at most 370x projected speedup. For the Delicious and Flickr tensors, while passing from 8 nodes (with 8×16 cores) to 256 nodes (with 256×16 cores), **med-gd** results in 5.29x and 4.32x speedups. The **fine-hp** and **fine-lb** partitioning result in 11.33x and 10.11x speedups for Delicious, and 11.62x and 10.50x speedups for Flickr in the same scenario.

On Netflix and NELL, **med-gd** yields 1286x and 790x projected speedups using 4096 cores. **fine-hp** shows a comparable performance with 1044x and 657x speedups, whereas **fine-lb** is slightly slower than **fine-hp** (945x and 630x speedups). We first note that in passing from 4 nodes (with 4×16 cores) to 256 nodes (with 256×16 cores), **med-gd** results in 12.73x and 12.03x speedups, respectively for Netflix and NELL. The **fine-hp** and **fine-lb** partitioning result in 10.18x and 11.18x speedups for Netflix, and 9.72x and 9.32x speedups for NELL in the same scenario. On average, on Netflix and NELL, **med-gd** gets 23% and 20% faster than **fine-hp**, and 36% and 25% faster than **fine-lb**. In Table 5, we see that this is due to **med-gd** incurring smaller maximum communication volume. We investigated this outcome and observed that when a tensor is long in one mode and short in all others, the communication due to the long mode dominates the overall cost, and the communication for the small modes remains negligible in comparison. On Netflix with $P = 256$ MPI ranks, using **med-gd** with $P = p \times q \times r$ topology, the worst case communication volume for the first mode is upper-bounded by $480K(qr - 1)$, as $I_1 = 480K$ indices are distributed to p process "slices" each with qr processes. Similarly, for the second and the third modes, the worst case communication volumes are $17K(pr - 1)$ and $2K(pq - 1)$. In such cases, choosing a large p and smaller q and r significantly reduces the worst-case communication cost in the first mode, while the cost in other modes stays low. The medium-grain heuristic achieves this. Specifically, on Netflix, the medium-grain heuristic chooses a grid size of $64 \times 4 \times 1$. This advantage is lost when there are at least two long dimensions (see Delicious and Flickr), as using more processes in one

Table 5 – Load balance and communication statistics for 256-way partitioning.

Partitioning	Nnz		Comp. Load		Comm. Vol.		Num. Msg	
	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.
Delicious								
fine-hp	547K	547K	1947K	1807K	199K	137K	2039	2018
fine-lb	564K	547K	1849K	1737K	309K	265K	2040	2040
fine-rd	550K	547K	2747K	2737K	1083K	1080K	2040	2040
medium-gd	598K	547K	2353K	2214K	624K	571K	1096	1096
Flickr								
fine-hp	441K	441K	1334K	1221K	54K	38K	1827	1588
fine-lb	454K	441K	1335K	1257K	107K	87K	2040	2030
fine-rd	443K	441K	2318K	2308K	1042K	1038K	2040	2040
medium-gd	443K	441K	1826K	1806K	576K	558K	1152	1152
Netflix								
fine-hp	392K	392K	1439K	1211K	49K	19K	1380	1158
fine-lb	404K	392K	1252K	1208K	48K	39K	1530	1528
fine-rd	394K	392K	1681K	1674K	412K	411K	1530	1530
medium-gd	394K	393K	1184K	1177K	26K	24K	642	642
NELL								
fine-hp	307K	307K	1219K	787K	46K	23K	1513	1402
fine-lb	316K	307K	920K	888K	89K	84K	1522	1502
fine-rd	309K	307K	1211K	1205K	271K	269K	1530	1520
medium-gd	310K	307K	871K	855K	58K	51K	583	570
Amazon								
fine-lb	5104K	4955K	16871K	16241K	211K	203K	1503	1503
fine-rd	4962K	4955K	20964K	20940K	4656K	4651K	1530	1530
medium-gd	19984K	4955K	50023K	16255K	230K	170K	550	514

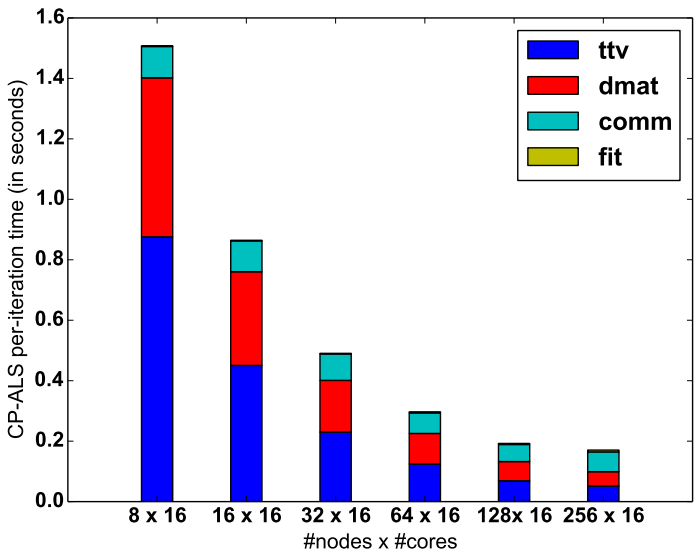


Figure 3 – Running time dissection of a parallel CP-ALS iteration using **fine-hp** and hp-tree scheme. The legends "ttv", "dmat", "comm", and "fit" correspond to the time spent for TTVs, dense matrix operations following the TTVs, communication, and fit computation. The fit computation's time is not discernible in the plot.

long mode can increase the communication significantly in the other long modes.

On Amazon tensor, **med-gd** starts to lose scalability at 256 MPI ranks. In Table 5 we observe that this is due to the load imbalance. Amazon tensor has some relatively "dense" slices that make load balancing difficult for the medium-grain heuristic. This problem never arises in the fine-grain partitioning due to finer granularity of tasks; as a result, **fine-lb** runs 1.85x faster than **med-gd** using 256 MPI ranks. We do not provide projected speedups for Amazon as we needed at least 64 nodes for the single-threaded execution, which renders the approximation somewhat inaccurate. In this tensor, from 128 nodes to 256 nodes, **med-gd** displays a speedup of 1.08. With **fine-lb**, the parallel algorithm enjoys 1.86x speedup in passing from 64 nodes to 128 nodes, and 3.2x speedup in passing from 64 to 256 nodes.

In Fig. 3 we present the dissection of the parallel run time for a CP-ALS iteration on Flickr tensor using 256 MPI ranks. We choose Flickr as representative, as it includes the highest proportion of dense matrix operations in comparison to all other tensors. Despite this fact and using a BDT for faster TTVs, the TTV step still remains to be the dominant computational cost. First, we observe that the workload due to TTV and dense matrix computations decrease with the increasing number of processes. This is despite the fact that CP-ALS on Flickr involves the highest proportion of dense matrix operations in comparison to all other tensors in our dataset. Second, we expect in general that having more processes increases the total communication volume; yet we observe in the plot that the communication cost declines until 128 MPI ranks. This is because a good partitioning can reduce the communication volume per process (while increasing the total communication volume). At 256 MPI ranks, however, communication cost starts to increase and become the bottleneck. The fit computations take negligible time and are not discernible in the plot.

On Flickr tensor, the three variants of the **fine-lb** took 58.38, 89.66, and 65.04 seconds to partition the hypergraph, **med-gd** took 190 seconds, and **fine-hp** took 207 minutes. In all data instances, **fine-lb** gives good results while being a fast partitioning heuristic. **fine-hp** consistently provides better partitions than **fine-lb** in all instances, yet the partitioning cost might render it impractical to use in real-world scenarios. **med-gd** heuristic is only effective when the tensor nonzeros are homogeneously distributed, and the tensor has only one large dimension. One might consider reducing the communication volume on a medium-grain topology using hypergraph partitioning, yet the high number of constraints prevents this approach from being amenable. Therefore, we believe that **fine-lb** serves well in most practical situations.

7 Conclusion

We investigated an efficient computation of successive tensor-times-vector multiplication in the context of the well-known CP-ALS algorithm for sparse tensor factorization. We introduced a computational scheme using dimension trees that asymptotically reduces the computational cost of the TTV operations for higher order tensors while using a reasonable amount of memory. Our technique provides performance benefits for lower order tensors, and gets progressively better as the dimensionality of the tensor increases in comparison to the state of the art. We proposed an effective shared memory parallelization of this method with a pre-computation step in order to efficiently carry out numerical computations within the CP-ALS iterations. We introduced a fine-grain parallelization approach in the distributed memory setting, compared it against a recently proposed medium-grain variant, discussed good partitionings for both approaches, and validated these findings with experiments on real-world tensors. The proposed computational scheme can be applied to both dense and sparse tensors as well as other tensor decomposition algorithms involving successive tensor-times-vector and -matrix multiplications. We are planning to investigate this potential in our future work.

Appendix

Here we collect some definitions regarding hypergraphs and explain the real-world tensors used in the experiments.

A Hypergraphs and hypergraph partitioning

A hypergraph $H = (V, E)$ consists of a set V of vertices and a set E of hyperedges. Each hyperedge is a subset of V. 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 by $\Pi = \{V_1, \ldots, V_K\}$, where the parts are non-empty, i.e., $V_k \neq \emptyset$ for $k \in \mathbb{N}_K$; mutually exclusive, i.e., $V_k \cap V_\ell = \emptyset$ for $k \neq \ell$; and collectively exhaustive, i.e., $V = \bigcup V_k$.

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

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k \in \mathbb{N}_K \tag{8}$$

we say that Π is *balanced* where ε represents the allowed maximum 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 is called its *connectivity*, and is denoted by λ_h. Given a vertex partition Π of a hypergraph $H = (V, E)$, one can measure the *cutsize* metric induced by Π as

$$\chi(\Pi) = \sum_{h \in E} \mathsf{c}[h](\lambda_h - 1) \ . \tag{9}$$

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 [30].

A common variant of the above problem is the *multi-constraint hypergraph partitioning* [15, 23]. 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 $i \in \mathbb{N}_C$. In this variant, the balance criterion (8) is rewritten as

$$W_{k,i} \leq W_{avg,i}\,(1 + \varepsilon) \text{ for } k \in \mathbb{N}_K \text{ and } i \in \mathbb{N}_C \tag{10}$$

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., $W_{k,i} = \sum_{v \in V_k} \mathsf{w}[v, i]$, and $W_{avg,i}$ represents the average part weight for the ith weight of all vertices, i.e., $W_{avg,i} = \sum_{v \in V} \mathsf{w}[v, i]/K$.

B Data set and the environment

Netflix tensor has user × movie × time dimensions, which we formed from the data of the Netflix Prize competition [8]. In this tensor, nonzeros correspond to the user reviews for movies, and the 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. We obtained the NELL tensor from the Never Ending Language Learning (NELL) knowledge database of the "Read the Web" project [10], 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 are formed by Görlitz et al. [17]. These datasets consist of tuples of the form (time × users × resources × tags); hence, we form 4-mode tensors out of these tuples. We obtained the Amazon review dataset from SNAP [31], which contains product review texts by users. We then processed this dataset with the standard text processing routines. We used the `nltk` package [9] in Python to tokenize the review text, to discard the stop words, to apply Porter stemmer, and to keep the words that are in the US, GB, or CA dictionaries. Afterwards, we retained only the words with at least five occurrences in the whole review set. We then set the numerical values by measuring the frequency of a word in a review.

We conducted experiments on a shared memory and a separate distributed memory system. The shared memory system has two Haswell CPUs sockets (Intel(R) Xeon(R) CPU E5-2695 v3) each having 14 cores at a clock speed of 2.30GHz with Turbo Boost disabled. The system has a total memory of size 768GB and L1, L2, L3 caches of sizes of 32KB, 256KB, and 35MB, respectively.

All the codes in this shared memory system are compiled with gcc/g++-5.3.0 using OpenMP directives and compiler options of -O3, -ffast-math, -funroll-loops, -ftree-vectorize, -fstrict-aliasing. The distributed memory system is an IBM Blue Gene/Q cluster. This system consists of 6 racks of 1024 nodes with each node having 16 GBs of memory and a 16-core IBM PowerPC A2 processor running at 1.6 GHz. We ran our experiments up to 256 nodes (4096 cores). Each core of PowerPC A2 can handle one arithmetic and memory operation simultaneously; therefore, we assigned 32 threads per node (2 threads per core) for better performance. On this system, all codes were compiled using the Clang C++ compiler (version 3.5.2) with IBM MPI wrapper using the same optimization flags, and linked against IBM ESSL library for LAPACK and BLAS routines.

Acknowledgments

Some preliminary experiments were carried out using the workstations and the PSMN cluster at ENS Lyon. This work was performed using HPC resources from GENCI-[TGCC/CINES/IDRIS] (Grant 2016 - i2016067501).

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] W. Austin, G. Ballard, and T. G. Kolda. Parallel tensor compression for large-scale scientific data. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, May 23–27*, pages 912–922, 2016.

[4] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006.

[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 `http://www.sandia.gov/ tgkolda/TensorToolbox/`, February 2015.

[7] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–6, Sept 2012.

[8] J. Bennett and S. Lanning. The Netflix Prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[9] S. Bird, E. Loper, and E. Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

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

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

[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. `http://bmi.osu.edu/ umit/software.htm`, 1999.

[13] Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Supercomputing, ACM/IEEE 2001 Conference*, page 42, Denver, Colorado, 2001.

[14] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 32(2):656–683, 2010.

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

[16] J. H. Choi and S. V. N. Vishwanathan. DFacTo: Distributed factorization of tensors. In *27th Advances in Neural Information Processing Systems*, pages 1296–1304, Montreal, Quebec, Canada, 2014.

[17] 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*, page 19, Berkeley, CA, USA, 2008. USENIX Association.

[18] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.

[19] 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.

[20] J. Håstad. Tensor rank is NP-complete. *Journal of Algorithms*, 11(4):644–654, 1990. ISSN 0196-6774.

[21] 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*, pages 316–324, New York, NY, USA, 2012. ACM.

[22] L. Karlsson, D. Kressner, and A. Uschmajew. Parallel algorithms for tensor completion in the CP format. *Parallel Computing*, 57(C):222–234, Sept. 2016.

[23] 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.

[24] O. Kaya and B. Uçar. High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors. Technical Report RR-8801, Inria, Oct 2015.

[25] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 77:1–77:11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6.

[26] O. Kaya and B. Uçar. High performance parallel algorithms for the Tucker decomposition of sparse tensors. In *45th International Conference on Parallel Processing (ICPP '16)*, pages 103–112, Aug 2016.

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

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

[29] L. D. Lathauwer and B. D. 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.

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

[31] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[32] I. Perros, R. Chen, R. Vuduc, and J. Sun. Sparse hierarchical Tucker factorization and its application to healthcare. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 943–948, Nov 2015.

[33] S. Rendle and T. S. Lars. Pairwise interaction tensor factorization for personalized tag recommendation. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 81–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-889-6.

[34] S. Rendle, B. M. Leandro, A. Nanopoulos, and L. Schmidt-Thieme. Learning optimal ranking with tensor factorization for tag recommendation. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 727–736, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9.

[35] G. M. Slota, K. Madduri, and S. Rajamanickam. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *Proc. 2nd IEEE Int'l. Conf. on Big Data (BigData)*, pages 481–490. IEEE, Oct. 2014.

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

[37] S. Smith and G. Karypis. A medium-grained algorithm for sparse tensor factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 902–911, 2016.

[38] 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*, pages 61–70, Hyderabad, India, May 2015. IEEE Computer Society.

[39] S. Smith, J. Park, and G. Karypis. An exploration of optimization algorithms for high performance tensor completion. *Proceedings of the 2016 ACM/IEEE conference on Supercomputing*, 2016.

[40] P. Symeonidis, A. Nanopoulos, and Y. Manolopoulos. Tag recommendations based on tensor dimensionality reduction. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, pages 43–50, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-093-7.

[41] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

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

[43] N. Zheng, Q. Li, S. Liao, and L. Zhang. Flickr group recommendation based on tensor decomposition. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 737–738, NY, USA, 2010. ACM.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399