Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees

Oguz Kaya, Bora Uçar

PARALLEL CANDECOMP/PARAFAC DECOMPOSITION OF SPARSE TENSORS USING DIMENSION TREES*

OGUZ KAYA† AND BORA UÇAR‡

Abstract. CANDECOMP/PARAFAC (CP) decomposition of sparse tensors has been successfully applied to many problems in web search, graph analytics, recommender systems, health care data analytics, and many other domains. In these applications, efficiently computing the CP decomposition of sparse tensors is essential in order to be able to process and analyze data of massive scale. For this purpose, we investigate an efficient computation of the CP decomposition of sparse tensors and its parallelization. We propose 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 5.96x, 5.65x, and 3.9x speedup in sequential, shared memory parallel, and distributed memory parallel executions over the state of the art, and achieve strong scalability up to 4096 cores on an IBM BlueGene/Q supercomputer.

Key words. sparse tensors, CP decomposition, dimension tree, parallel algorithms

AMS subject classifications. 15-04, 05C70, 15A69, 15A83

1. Introduction.

With growing features and dimensionality of data, tensors, or multi-dimensional arrays, have been increasingly used in many fields including the analysis of Web graphs [28], knowledge bases [10], recommender systems [36, 37, 43], signal processing [30], computer vision [46], health care [34], and many others [29]. 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 [29]), and algorithmic and software contributions go hand in hand with these efforts [2, 5, 16, 22, 26, 27, 42, 40].

One of the well known tensor decompositions is the CANDECOMP/PARAFAC (CP) formulation, which approximates a given tensor as a sum of rank-one tensors. Among the commonly used algorithms for computing a CP decomposition is CP-ALS [11, 19], which is based on the alternating least squares method, though other variants also exist [1, 44]. These algorithms are iterative, in which the computational core of each iteration involves a special operation called matricized tensor-times Khatri-Rao product (MTTKRP). When the input tensor is sparse and N dimensional, MTTKRP operation amounts to element-wise multiplication of $N-1$ row vectors from $N-1$ matrices and their scaled sum reduction according to the nonzero structure of the tensor. As 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 in emerging applications [34]. This operation has received recent interest for efficient execution in different settings such as MATLAB [2, 5], MapReduce [22], shared memory [42], and distributed memory [16, 26, 40].

*Submitted to the editors November 8, 2016. A preliminary version appeared in SC'15 [26].

†INRIA and LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France (oguz.kaya@ens-lyon.fr).

‡CNRS and LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France (bora.ucar@ens-lyon.fr)

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 for sparse tensors in shared and distributed memory systems. 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 the experiments. We propose a fine-grain distributed memory parallel algorithm, and compare it against a medium-grain variant [40]. Finally, we discuss effective partitioning routines for these algorithms. Even though the discussion is confined to CP-ALS in the paper, the contributions apply to any other algorithm involving MTTKRP in its core [1].

The organization of the rest of the paper is as follows. In the next section, we introduce our notation, describe the 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, we discuss an effective shared and distributed memory parallelization of CP-ALS iterations in section 4. We discuss partitioning methods 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, we present experimental results in section 6 to demonstrate performance gains using our algorithms with shared and distributed memory parallelism over an efficient state of the art implementation, and then conclude the paper.

2. Background and notation.

2.1. Tensors and CP-ALS. We denote the set $\{1, \ldots, M\}$ of integers as $\mathbb{N}_M$ for $M \in \mathbb{Z}^+$. 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 [29]; particularly, we use bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{X}}$, to represent tensors. The *order* of a tensor is defined as the number of its *dimensions*, or equivalently, *modes*, which we denote by N. 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 third dimension. We use italic lowercase letters with subscripts to represent vector, matrix, and tensor elements, e.g., x_i for a vector $\mathbf{x}$, $x_{i,j}$ for a matrix $\mathbf{X}$, and $x_{i,j,k}$ 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 $\mathbf{X}(:,i)$, whereas a row of a matrix is always expressed in MATLAB notation, as in $\mathbf{X}(i,:)$. Sets, lists, trees, and hypergraphs are expressed in non-bold calligraphic fonts.

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 the mode n with a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ 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 is denoted by $\boldsymbol{\mathcal{Y}} =$

$\boldsymbol{\mathcal{X}} \times_n \mathbf{v}$. The order of a series of TTVs is irrelevant, i.e., $\boldsymbol{\mathcal{X}} \times_i \mathbf{u} \times_j \mathbf{v} = \boldsymbol{\mathcal{X}} \times_j \mathbf{v} \times_i \mathbf{u}$ for $\mathbf{u} \in \mathbb{R}^{I_i}$, $\mathbf{v} \in \mathbb{R}^{I_j}$, $i \neq j$, and $i, j \in \mathbb{N}_N$.

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 with the rows of $\mathbf{X}$, and the rest of the modes with 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 the columns of the resulting matrix. We use $\mathbf{X}_{(d)}$ to denote the matricization along 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 $x_{i_1,\ldots,i_K} = \prod_{k \in \mathbb{N}_K} u_{i_k}^{(k)}$. The *Kronecker product* of vectors $\mathbf{u} \in \mathbb{R}^I$ and $\mathbf{v} \in \mathbb{R}^J$ results in a vector $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}, \mathbf{w} \in \mathbb{R}^{IJ}$ 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 $\mathbf{W} \in \mathbb{R}^{IJ \times K}$.

For the operator $\circ$, we use the shorthand notation $\circ_{i \neq n} \mathbf{U}^{(i)}$ to denote the operation $\mathbf{U}^{(1)} \circ \cdots \circ \mathbf{U}^{(n-1)} \circ \mathbf{U}^{(n+1)} \circ \cdots \circ \mathbf{U}^{(N)}$ over a set $\{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\}$ of matrices (and similarly for vectors). Similarly, $\boldsymbol{\mathcal{X}} \times_{i \in \mathcal{I}} \mathbf{u}^{(i)}$ denotes the operation $\boldsymbol{\mathcal{X}} \times_{i_1} \mathbf{u}^{(i_1)} \times_{i_2} \cdots \times_{i_{|\mathcal{I}|}} \mathbf{u}^{(i_{|\mathcal{I}|})}$ over a set $\mathcal{I} = \{i_1, \ldots, i_{|\mathcal{I}|}\}$ of dimensions using a set $\{\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(N)}\}$ of vectors.

2.2. CP decomposition. The rank-R CP-decomposition of a tensor $\boldsymbol{\mathcal{X}}$ expresses or approximates $\boldsymbol{\mathcal{X}}$ as a sum of R rank-1 tensors. For instance, for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$, we obtain $\boldsymbol{\mathcal{X}} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ 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 $\boldsymbol{\mathcal{X}}$, and computing this value is NP-hard [21]. 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 a CP decomposition is the alternating least squares (CP-ALS) method, which establishes a good trade-off between the number of iterations and the cost per iteration [29]. It is an iterative algorithm, shown in Algorithm 1, that progressively updates the factors $\mathbf{U}^{(n)}$ in an alternating fashion starting from an initial guess. CP-ALS runs until it can no longer improve the solution, or it reaches the allowed maximum number of iterations. The initial factor matrices can be randomly set, or computed using the truncated SVD of

the matricizations of $\mathcal{X}$ [29]. Each iteration of CP-ALS consists of N *subiterations*, where in the nth subiteration $\mathbf{U}^{(n)}$ is updated using $\mathcal{X}$ and 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
$\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}$: Initial factor matrices
Output: $[\![\boldsymbol{\lambda}; \mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$: A rank-$R$ CP decomposition of $\mathcal{X}$
1: **repeat**
2: **for** $n = 1,\ldots,N$ **do**
3: $\mathbf{M}^{(n)} \leftarrow \mathbf{X}_{(n)} \odot_{i\neq n} \mathbf{U}^{(i)}$
4: $\mathbf{H}^{(n)} \leftarrow *_{i\neq n}(\mathbf{U}^{(i)^T}\mathbf{U}^{(i)})$
5: $\mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)}\mathbf{H}^{(n)^\dagger}$ ▶ $\mathbf{H}^{(n)^\dagger}$ is the pseudoinverse of $\mathbf{H}^{(n)}$.
6: $\boldsymbol{\lambda} \leftarrow$ Column-Normalize($\mathbf{U}^{(n)}$)
7: **until** convergence is achieved or the maximum number of iterations is reached.
8: **return** $[\![\boldsymbol{\lambda}; \mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$

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, for both sparse and dense tensors. The operation $\mathbf{X}_{(n)} \odot_{i\neq n} \mathbf{U}^{(i)}$ is called *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 Khatri-Rao product. One such formulation [4], also used in Tensor Toolbox [5], 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 $\mathbf{M}^{(n)}(:,r) \leftarrow \mathcal{X} \times_{i\neq n} \mathbf{u}_r^{(i)}$, or equivalently,

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

Once $\mathbf{M}^{(n)}$ is obtained, the Hadamard product of matrices $\mathbf{U}^{(i)^T}\mathbf{U}^{(i)}$ of size $R \times R$ is computed 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 nth subiteration, 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 $i \in \mathbb{N}_N$, then update $\mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$ once $\mathbf{U}^{(n)}$ changes. As in many cases the rank R of approximation is chosen as a small constant in practice for sparse tensors (less than 50) [47], 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 when R is small. Finally, $\mathbf{U}^{(n)}$ is normalized column-wise, and the column vector norms are stored in a vector $\boldsymbol{\lambda} \in \mathbb{R}^R$.

The convergence is achieved when the relative reduction in the norm of the error, 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.

2.3. Hypergraphs and hypergraph partitioning. We partition the data and the computation using the standard hypergraph partitioning tools. Necessary definitions follow.

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a set $\mathcal{V}$ of vertices and a set $\mathcal{E}$ of hyperedges. Each hyperedge is a subset of $\mathcal{V}$. Weights, denoted with $w[\cdot]$, and costs, denoted with $c[\cdot]$, can be associated with, respectively, the vertices and the hyperedges of $\mathcal{H}$. For a given integer $K \geq 2$, a K-way vertex partition of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is denoted by $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$, where the parts are non-empty, i.e., $\mathcal{V}_k \neq \emptyset$ for $k \in \mathbb{N}_K$; mutually exclusive, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for $k \neq \ell$; and collectively exhaustive, i.e., $\mathcal{V} = \bigcup \mathcal{V}_k$.

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

$$W_k \leq W_{avg}(1 + \varepsilon) \quad \text{for} \quad k \in \mathbb{N}_K \ , \tag{4}$$

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 κ_h. Given a vertex partition Π of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, one can measure the *cutsize* metric induced by Π as

$$\chi(\Pi) = \sum_{h \in \mathcal{E}} c[h](\kappa_h - 1) \ . \tag{5}$$

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

Given $\varepsilon \geq 0$ 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. Hypergraph partitioning problem is NP-hard [31].

A common variant of the above problem is the *multi-constraint hypergraph partitioning* [15, 24]. In this variant, each vertex has an associated vector of weights. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint for each weight. Let $w[v, i]$ denote the C weights of a vertex v for $i \in \mathbb{N}_C$. In this variant, the balance criterion (4) is rewritten as

$$W_{k,i} \leq W_{avg,i}\,(1 + \varepsilon) \text{ for } k \in \mathbb{N}_K \text{ and } i \in \mathbb{N}_C \ , \tag{6}$$

where the ith weight $W_{k,i}$ of a part $\mathcal{V}_k$ is defined as the sum of the ith weights of the vertices in that part, i.e., $W_{k,i} = \sum_{v \in \mathcal{V}_k} w[v, i]$, and $W_{avg,i}$ represents the average part weight for the ith weight of all vertices, i.e., $W_{avg,i} = \sum_{v \in \mathcal{V}} w[v, i]/K$.

2.4. Dimension tree. A *dimension tree* is a data structure that 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 of a tensor [18], which was introduced as a computationally feasible alternative to the original Tucker decomposition for higher order tensors. We provide the formal definition of a dimension tree along with some basic properties as follows.

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

1. $\mu(\text{ROOT}(\mathcal{T})) = \mathbb{N}_N$.

2. *For each non-leaf node $t \in \mathcal{T}$, the mode sets of its children partition $\mu(t)$.*
3. *The nth leaf node, denoted by $l_n \in \textsc{Leaves}(\mathcal{T})$, has $\mu(l_n) = \{n\}$.*

For the simplicity of the discussion, we assume without loss of generality that the sequence $l_1, \ldots, l_N$ corresponds to a relative ordering of the leaf nodes in a post-order traversal of the dimension tree. If this is not the case, we can relabel tensor modes accordingly. We define the *inverse mode set* of a node t as $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$. For each node t with a parent $\mathcal{P}(t)$, $\mu(t) \subset \mu(\mathcal{P}(t))$ holds due to the second property, which in turn yields $\mu'(t) \supset \mu'(\mathcal{P}(t))$. 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 Figure 1, we show a BDT for 4 dimensions (associated with a sparse tensor described later).

3. Computing CP decomposition using dimension trees. In this section, 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 without the use of a tree framework was moderately explored by Baskaran et al. [6] for computing the Tucker decomposition of sparse tensors, and by Phan et al. for computing the CP decomposition of dense tensors [35]. We generalized the approach of Baskaran et al. [6] using dimension trees for better computational gains [25] in the standard algorithm for sparse Tucker decomposition. Here, we adopt the same data structure for reducing the cost of MTTKRP operations.

3.1. Using dimension trees to perform successive tensor-times-vector multiplies. At each subiteration of the CP-ALS algorithm, $\boldsymbol{\mathcal{X}}$ is multiplied with the column vectors of matrices in $N-1$ modes using (3) in performing MTTKRP. Some of these TTVs involve the same matrices as the preceding subiterations. As a series of TTVs can be done in any order, 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 $\boldsymbol{\mathcal{X}}$, we compute $\boldsymbol{\mathcal{X}} \times_2 \mathbf{u}_r^{(2)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and eventually update $\mathbf{u}_r^{(1)}$ for each $r \in \mathbb{N}_R$, whereas in the second subiteration we compute $\boldsymbol{\mathcal{X}} \times_1 \mathbf{u}_r^{(1)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ 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 $\boldsymbol{\mathcal{X}}$ with $\mathbf{u}_r^{(3)}$ and $\mathbf{u}_r^{(4)}$. Hence, we can compute $\boldsymbol{\mathcal{Y}}_r = \boldsymbol{\mathcal{X}} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$, then reuse it in the first and the second subiterations as $\boldsymbol{\mathcal{Y}}_r \times_2 \mathbf{u}_r^{(2)}$ and $\boldsymbol{\mathcal{Y}}_r \times_1 \mathbf{u}_r^{(1)}$ to obtain the required TTV results.

We use dimension trees to systematically detect and reuse such partial results by associating a tree $\mathcal{T}$ with an N-dimensional tensor $\boldsymbol{\mathcal{X}}$ as follows. With each node $t \in \mathcal{T}$, we associate R tensors $\boldsymbol{\mathcal{X}}_1^{(t)}, \ldots, \boldsymbol{\mathcal{X}}_R^{(t)}$. $\boldsymbol{\mathcal{X}}_r^{(t)}$ corresponds to the TTV result $\boldsymbol{\mathcal{X}}_r^{(t)} = \boldsymbol{\mathcal{X}} \times_{d \in \mu'(t)} \mathbf{u}_r^{(d)}$. Therefore, the inverse mode set $\mu'(t)$ corresponds to the set of modes in which TTV is performed on $\boldsymbol{\mathcal{X}}$ to form $\boldsymbol{\mathcal{X}}_r^{(t)}$. For the root of the tree, $\mu'(\textsc{Root}(\mathcal{T})) = \emptyset$; thus, all tensors of the root node correspond to the original tensor $\boldsymbol{\mathcal{X}}$, i.e., $\boldsymbol{\mathcal{X}}_r^{(\textsc{Root}(\mathcal{T}))} = \boldsymbol{\mathcal{X}}$ for $r \in \mathbb{N}_R$. Since $\mu'(t) \supset \mu'(\mathcal{P}(t))$ for a node t and its parent $\mathcal{P}(t)$, tensors of $\mathcal{P}(t)$ can be used as partial results to update the tensors of t. Let $\delta(t) = \mu'(t) \setminus \mu'(\mathcal{P}(t))$. We can then compute each tensor of t from its parent's as $\boldsymbol{\mathcal{X}}_r^{(t)} = \boldsymbol{\mathcal{X}}_r^{(\mathcal{P}(t))} \times_{d \in \delta(t)} \mathbf{u}_r^{(d)}$. 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 in 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 $\mathcal{P}(t)$ is called first to make sure that $\mathcal{P}(t)$'s tensors are up-to-date. Then, each $\boldsymbol{\mathcal{X}}_r^{(t)}$ is computed by performing a TTV on the corresponding tensor $\boldsymbol{\mathcal{X}}_r^{(\mathcal{P}(t))}$ of the parent. We use the notation $\boldsymbol{\mathcal{X}}_:^{(t)}$ to denote all R tensors of a node t.

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

Input: t: A node of the dimension tree
Output: Tensors $\boldsymbol{\mathcal{X}}_:^{(t)}$ of t are computed.

```
1: if EXISTS(X_:^(t)) then
2:     return                                  ▶ Tensors X_:^(t) are already computed.
3: DTREE-TTV(P(t))                             ▶ Compute the parent's tensors X_:^(P(t)) first.
4: for r = 1, ..., R do                        ▶ Now update all tensors of t using parent's.
5:     X_r^(t) ← X_r^(P(t)) ×_{d∈δ(t)} u_r^(d)
```

3.2. Dimension tree-based CP-ALS algorithm. At the nth subiteration of Algorithm 1, we need to compute $\boldsymbol{\mathcal{X}} \times_{i \neq n} \mathbf{u}_r^{(i)}$ for all $r \in \mathbb{N}_R$ in order to form $\mathbf{M}^{(n)}$. Using a dimension tree $\mathcal{T}$ with leaves $l_1, \ldots, l_N$, we can perform this simply by executing DTREE-TTV(l_n), after which the rth tensor of 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 the leaf order $l_1, \ldots, l_N$ obtained from a post-order traversal of $\mathcal{T}$. 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 $\mu'(t)$ involve n, as in this case its tensors involve multiplication using the old value of $\mathbf{U}^{(n)}$ which is about to change. Note that this step destroys the tensors of all nodes not lying on a path from l_n to the root. Afterwards, DTREE-TTV is called at Line 9 for the leaf node l_n to compute its tensors. This step computes (or reuses) the tensors of all nodes from the path from l_n to the root. Next, the rth column of $\mathbf{M}^{(n)}$ is formed using $\boldsymbol{\mathcal{X}}_r^{(l_n)}$ 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 2. *Let $\boldsymbol{\mathcal{X}}$ 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 $l_1, \ldots, l_N$ is obtained from a post-order traversal of $\mathcal{T}$, for each internal node t, the subtree rooted at t has the leaves $l_i, l_{i+1}, \ldots, l_{i+k-1}$ corresponding to k consecutive mode indices for some positive integers i and k. As we have $\mu(l_i) = i$ 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 $l_{i+k'}$ for $0 \leq k' < k$, we have $i + k' \in \mu(t)$; hence $i + k' \notin \mu'(t)$ as $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$. 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

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: $[\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\mathcal{X}$

$\mathcal{T} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\mathcal{X}$) ▶ The tree has leaves $\{l_1, \ldots, l_N\}$.
for $n = 2 \ldots N$ **do**
  $\mathbf{W}^{(n)} \leftarrow \mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$
repeat
  for $n = 1, \ldots, N$ **do**
    for all $t \in \mathcal{T}$ **do**
      if $n \in \mu'(t)$ **then**
        DESTROY($\mathcal{X}_{:}^{(t)}$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
    DTREE-TTV(l_n) ▶ Perform the TTVs for the leaf node tensors.
    for $r = 1, \ldots, R$ **do** ▶ Form $\mathbf{M}^{(n)}$ column-by-column (done implicitly).
      $\mathbf{M}^{(n)}(:, r) \leftarrow \mathcal{X}_r^{(l_n)}$ ▶ $\mathcal{X}_r^{(l_n)}$ is a vector of size I_n.
    $\mathbf{H}^{(n)} \leftarrow *_{i \neq n}\mathbf{W}^{(i)}$
    $\mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)}\mathbf{H}^{(n)^\dagger}$
    $\boldsymbol{\lambda} \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
    $\mathbf{W}^{(n)} \leftarrow \mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$
until converge is achieved or the maximum number of iterations is reached.
return $[\![\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

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) get 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 are 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

$$\sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\delta(t)| = \sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\mu(\mathcal{P}(t)) \setminus \mu(t)|. \tag{7}$$

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 (7) as

$$\sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\mu(\mathcal{P}(t)) \setminus \mu(t)| = \sum_{t \in \mathcal{T} \setminus \text{LEAVES}(\mathcal{T})} |\mu(t)| \,. \tag{8}$$

Note that in constructing a BDT, at the root node we start with the mode set $\mu(\text{ROOT}(\mathcal{T})) = \mathbb{N}_N$. Then, at each level $k > 0$, we form the mode sets of the nodes at level k by partitioning the mode sets of their parents at level $k-1$. As a result, at each level k, each dimension $n \in \mathbb{N}_N$ can appear in only one set $\mu(t)$ for a node t belonging to the level k of the BDT. With this observation in mind, rewriting (8) by

iterating over nodes by levels of the BDT yields

$$\sum_{t\in\mathcal{T}\setminus\textsc{Leaves}(\mathcal{T})} |\mu(t)| = \sum_{k=1}^{\lceil \log N\rceil} \sum_{t\in\mathcal{T}\setminus\textsc{Leaves}(\mathcal{T}),\textsc{Level}(t)=k} |\mu(t)|$$

$$\leq \sum_{k=1}^{\lceil \log N\rceil} N = N\lceil \log N\rceil \ .$$

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. ∎

In comparison, the traditional scheme [42] incurs $R(N-1)$ TTVs in each mode, and $RN(N-1)$ TTVs in total in an iteration. This yields a factor of $(N-1)/\log N$ reduction in the number performed of TTVs using dimension trees. We note that in terms of the actual computational cost, this corresponds to a lower bound on the expected speedup for the following reason. As the tensor is multiplied in different dimensions, resulting tensors are expected to have many index overlaps, effectively reducing their number of nonzeros and rendering subsequent TTVs significantly cheaper. This renders using a BDT much more effective as it avoids repeating such expensive TTVs at the higher levels of the dimension tree by reusing partial results. That is, the formula for the potential gain is a complicated function, depending on the sparsity of the tensor. On one extreme, multiplying the tensor in certain dimensions might create no or few index overlaps, which makes the cost of each TTV approximately equal, yielding the stated speedup. On the other extreme, the first TTVs performed the original tensor may drastically reduce the number of tensor elements so that the cost of the subsequent TTVs becomes negligible. The traditional scheme multiplies the original tensor N times, once per dimension, whereas a BDT suffices with 2 such TTVs as the root node has only two children, yielding a speedup factor of $N/2$. Therefore, in practice the actual speedup is expected to be between these two extremes depending on the sparsity of the tensor, and having more speedup with higher index overlap after multiplications.

For sparse tensors, one key idea we use for obtaining high performance is performing TTVs for all R tensors $\boldsymbol{\mathcal{X}}_{:}^{(t)}$ of a node $t \in \mathcal{T}$ in a *vectorized* manner. We illustrate this on a 4-dimensional tensor $\boldsymbol{\mathcal{X}}$ 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 $\boldsymbol{\mathcal{X}}_r^{(t_{1234})} = \boldsymbol{\mathcal{X}}$ for all $r \in \mathbb{N}_R$. The two children of t_{1234} are t_{12} and t_{34} with the corresponding tensors $\boldsymbol{\mathcal{X}}_r^{(t_{12})} = \boldsymbol{\mathcal{X}}_r^{(t_{1234})} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and $\boldsymbol{\mathcal{X}}_r^{(t_{34})} = \boldsymbol{\mathcal{X}}_r^{(t_{1234})} \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$, respectively. Since $\boldsymbol{\mathcal{X}}_r^{(t_{1234})}$ are identical for all $r \in \mathbb{N}_R$, the nonzero pattern of tensors $\boldsymbol{\mathcal{X}}_r^{(t_{12})}$ are also identical. This is also the case for $\boldsymbol{\mathcal{X}}_r^{(t_{34})}$, and 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 $\boldsymbol{\mathcal{Y}}_r \leftarrow \boldsymbol{\mathcal{X}}_r \times_d \mathbf{v}_r$ for $\mathbf{V} = [\mathbf{v}_1|\cdots|\mathbf{v}_R] \in \mathbb{R}^{I_d \times R}$ as

$$\mathbf{y}^{(:)}_{i_1,\ldots,i_{d-1},1,i_{d+1},\ldots,i_N} = \sum_{j=1}^{I_d} \mathbf{V}(j,:) * \mathbf{x}^{(:)}_{i_1,\ldots,i_{d-1},j,i_{d+1},\ldots,i_N}, \tag{9}$$

where $\mathbf{y}^{(:)}_{i_1,\ldots,i_{d-1},1,i_{d+1},\ldots,i_N}$ and $\mathbf{x}^{(:)}_{i_1,\ldots,i_{d-1},j,i_{d+1},\ldots,i_N}$ are vectors of size R with elements $y^{(r)}_{i_1,\ldots,i_{d-1},1,i_{d+1},\ldots,i_N}$ and $x^{(r)}_{i_1,\ldots,i_{d-1},j,i_{d+1},\ldots,i_N}$ in $\boldsymbol{\mathcal{Y}}_r$ and $\boldsymbol{\mathcal{X}}_r$, for all $r \in \mathbb{N}_R$. We call this operation 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 $\boldsymbol{\mathcal{Z}}_r \leftarrow \boldsymbol{\mathcal{Y}}_r \times_e \mathbf{w}_r = (\boldsymbol{\mathcal{X}}_r \times_d \mathbf{v}_r) \times_e \mathbf{w}_r$ in two modes d and e, $d < e$, with matrices $\mathbf{V} \in \mathbb{R}^{I_d \times R}$ and $\mathbf{W} \in \mathbb{R}^{I_e \times R}$ as

$$\mathbf{z}^{(:)}_{i_1,\ldots,i_{d-1},1,i_{d+1},\ldots,i_{e-1},1,i_{e+1},\ldots,i_N} = \sum_{j_2=1}^{I_e} \mathbf{W}(j,:) * \mathbf{y}^{(:)}_{i_1,\ldots,i_{d-1},1,i_{d+1},\ldots,i_{e-1},j_2,i_{e+1},\ldots,i_N}$$

$$= \sum_{(j_1,j_2)=(1,1)}^{(I_d,I_e)} \mathbf{V}(j_1,:) * \mathbf{W}(j_2,:) * \mathbf{x}^{(:)}_{i_1,\ldots,i_{d-1},j_1,i_{d+1},\ldots,i_{e-1},j_2,i_{e+1},\ldots,i_N}. \tag{10}$$

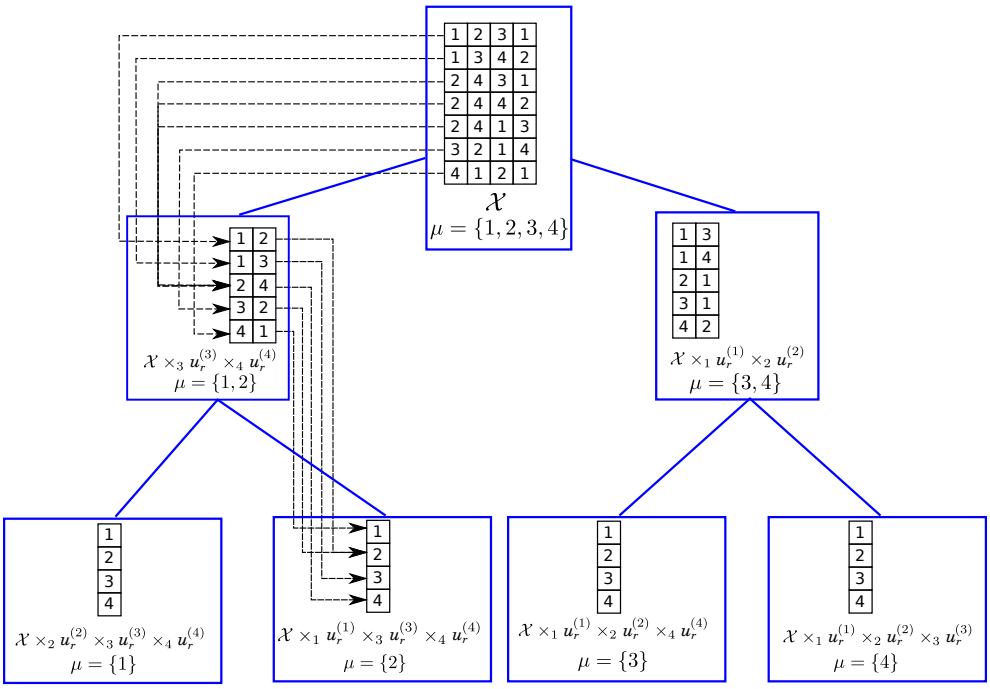
The formula similarly generalizes to any number of dimensions, where each dimension adds another Hadamard product with a corresponding matrix row. This "thick" mode of operation provides a significant performance gain thanks to the increase in locality. Also, performing TTVs in this manner in CP-ALS effectively reduces the $RN\lceil \log N \rceil$ TTVs required in Theorem 2 to $N\lceil \log N \rceil$ TTMV calls within an iteration. In our approach, for each node $t \in \mathcal{T}$, we store a single list $\mathcal{I}_t$ containing, for each $k \in \mathbb{N}_{|\mathcal{I}_t|}$, an index tuple of the form $\mathcal{I}_t(k) = (i_1, \ldots, i_N)$, to represent the nonzeros $x^{(r)}_{i_1,\ldots,i_N} \in \boldsymbol{\mathcal{X}}^{(t)}_r$ 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 $\mathbf{V}_t(k,:)$, where $\mathbf{V}_t \in \mathbb{R}^{|\mathcal{I}_t| \times R}$ is called the *value matrix* of t. Finally, carrying out the summation (9) requires a "mapping" $\mathcal{R}_t(k)$ indicating the set of elements of $\boldsymbol{\mathcal{X}}^{(P(t))}_{:}$ contributing to this particular element of $\boldsymbol{\mathcal{X}}^{(t)}_{:}$. Altogether, we represent the sparse tensors $\boldsymbol{\mathcal{X}}^{(t)}_{:}$ of each tree node t with the tuple $(\mathcal{I}_t, \mathcal{R}_t, \mathbf{V}_t)$ whose computational details follow next.

The sparsity of input tensor $\boldsymbol{\mathcal{X}}$ determines the sparsity structure of tensors in the dimension tree, i.e., $\mathcal{I}_t$ and $\mathcal{R}_t$, for each tree node t. Computing this sparsity structure for each TTV can get very expensive, and is redundant. As $\boldsymbol{\mathcal{X}}$ stays fixed, we can compute the sparsity of each tree 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.

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

The first information we need is the list 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 $\boldsymbol{\mathcal{X}}^{(t_{12})}_r = \boldsymbol{\mathcal{X}}^{(t_{1234})}_r \times_3 \mathbf{u}^{(3)}_r \times_4 \mathbf{u}^{(4)}_r$ and $\boldsymbol{\mathcal{X}}^{(t_{34})}_r = \boldsymbol{\mathcal{X}}^{(t_{1234})}_r \times_1 \mathbf{u}^{(1)}_r \times_2 \mathbf{u}^{(2)}_r$, respectively. Using (10), the nonzero indices of $\boldsymbol{\mathcal{X}}^{(t_{12})}_r$ and $\boldsymbol{\mathcal{X}}^{(t_{34})}_r$ take the form $(i,j,1,1)$ and $(1,1,k,l)$, respectively (tensor indices in the multiplied dimensions are always 1; hence we omit storing them in practice), and such a nonzero index exists in these tensors only if there exists a nonzero $x_{i,j,k,l} \in \boldsymbol{\mathcal{X}}^{(t_{1234})}_r$. Determining the list $\mathcal{I}_{t_{12}}$ (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 be efficiently eliminated

Fig. 1: BDT of a 4-dimensional sparse tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{4\times4\times4\times4}$ 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.



by sorting. Once the list of nonzeros for t_{12} (or t_{34}) is determined, we proceed to detecting 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 (9) and (10) for each nonzero index at a node t, we need to identify the set of nonzeros of the parent node $\mathcal{P}(t)$'s tensors that contribute to this nonzero. Specifically, at t_{12}, for each nonzero index $\mathcal{I}_{t_{12}}(m) = (i,j)$ we need to bookmark all nonzero index tuples of t_{1234} of the form $\mathcal{I}_{t_{1234}}(n) = (i,j,k,l)$, as this is this set of nonzeros of $\boldsymbol{\mathcal{X}}_{:}^{(t_{1234})}$ that contribute to the nonzero of $\boldsymbol{\mathcal{X}}_{:}^{(t_{12})}$ with index $(i,j,1,1)$ in (10). Therefore, for each such index tuple of t_{12} we need a *reduction set* $\mathcal{R}_{t_{12}}(m)$ which contains all such index tuples of the parent, i.e., $n \in \mathcal{R}_{t_{12}}(m)$. We determine these sets simultaneously with $\mathcal{I}_t$. In Figure 1, we illustrate a sample 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 memory cost of symbolic TTV using a BDT for sparse tensors.

THEOREM 3. *Let $\boldsymbol{\mathcal{X}}$ be an N-mode sparse tensor. The total number of index arrays in a BDT of $\boldsymbol{\mathcal{X}}$ 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 $\mu(t)$. As stated in the proof of Theorem 2, 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. ∎

Theorem 3 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 $\text{nnz}(\mathcal{X})$), yielding the overall worst-case memory cost $\text{nnz}(\mathcal{X})N(\lceil \log N \rceil + 1)$. Assuming that the tensor does not contain any empty slices (which can otherwise be removed in a preprocessing step), the index array of a leaf node (whose tensors are vectors) corresponding to mode n is simply $[1, \ldots, I_n]$ hence need not be explicitly stored, which effectively reduces this cost to $\text{nnz}(\mathcal{X})N\lceil \log N \rceil$. This cost is indeed a pessimistic estimate for real-world tensors, as with significant index overlap in non-root tree tensors, the total memory cost could reduce to as low as $O(\text{nnz}(\mathcal{X})N)$. This renders the approach very suitable for higher dimensional tensors. Another minor cost is the storage of reduction pointers, which necessitates one array per non-root tree node, taking $2N - 2$ arrays in total. The size of these arrays similarly diminishes towards the leaves with a potential index overlap.

In computing the symbolic TTV, we sort $|\mu(t)|$ index arrays for each node $t \in \mathcal{T}$. In addition, for each non-root 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 $\text{nnz}(\mathcal{X})$ elements. Therefore, combining with the number of index arrays as given in Theorem 3, the overall worst case cost of sorting becomes $O((N\lceil \log N \rceil + 2N - 2)\text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X})) = O(N \log N \ \text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X}))$. 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 2.57 and 5.5 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 towards the leaves. In comparison, existing approaches [39] sort the original tensor once with a cost of $O(N\text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X})))$ 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 factor matrices; therefore, it is often executed with multiple initializations [29]. In all of these use cases, the tensor $\mathcal{X}$ is fixed; therefore, the symbolic TTV is required only once. Moreover, CP-ALS usually has 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 this parallelism in our implementation.

After symbolic TTV is performed, index arrays of all nodes in the tree stay fixed and are kept throughout CP-ALS iterations. However, at each subiteration n, only the value matrices $\mathbf{V}_t$ of tree tensors which are necessary to compute Dtree-Ttv(l_n) are kept. The following theorem provides an upper bound on the number of such value matrices, which gives an upper bound on the memory usage for tensor values.

Theorem 4. *For an N-mode tensor $\mathcal{X}$, the total number of tree nodes whose value matrices are allocated is at most $\lceil \log N \rceil$ 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 $t \in \mathcal{T}$ involving n in $\mu'(t)$ are destroyed at Line 8. These are exactly the tensors that do not lie in the path from the leaf l_n to the root, as they do not involve n in their mode set μ. The TTMV result for l_n depends only on the nodes on this path from 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, the number of nodes whose value matrices are not destroyed cannot exceed $\lceil \log N \rceil$ at any instant of Algorithm 3. ∎

Theorem 4 puts an upper bound of $\lceil \log N \rceil$ on the maximum number of allocated value matrices, thus on the maximum memory utilization due to tensor values, in Dtree-Cp-Als. In the worst case, each value matrix may have up to $\text{nnz}(\boldsymbol{\mathcal{X}})$ elements, requiring $\text{nnz}(\boldsymbol{\mathcal{X}})R\lceil \log N \rceil$ memory in total to store the values. Combining with Theorems 2 and 3, in the worst case, the intermediate results increase the memory requirement by a factor of $O(\log N(1 + R/N))$ (with respect to the storage of $\boldsymbol{\mathcal{X}}$ in coordinate format), while the computational cost is reduced by a factor of $O(N/\log N)$. These are the largest increase in the memory and the smallest decrease in the computational cost. In practice, we expect less increase in the memory and more gains in the computational cost thanks to the overlap of indices towards the leaves.

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 TTMVs in subsection 4.1. Later, in subsection 4.2, we present distributed memory parallel algorithms that use this shared memory parallelization.

Algorithm 4 Smp-Dtree-Ttv

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

```
1: if Exists(V_t) then
2:    return                          ▶ Numerical values V_t are already computed.
3: Smp-Dtree-Ttv(P(t))                ▶ Compute the parent's values V_{P(t)} first.
4: parallel for 1 ≤ i ≤ |I_t| do      ▶ Process each I_t(i) = (i_1, ..., i_N) in parallel.
5:    V_t(i,:) ← 0                    ▶ Initialize with a zero vector of size 1 × R.
6:    for all j ∈ R_t(i) do           ▶ Reduce from the element with index I_{P(t)}(j) = (j_1, ..., j_N).
7:       r ← V_{P(t)}(j,:)
8:       for all d ∈ δ(t) do          ▶ Multiply the vector r with corresponding matrix rows.
9:          r ← r * U^(d)(j_d,:)
10:      V_t(i,:) ← V_t(i,:) + r      ▶ Add the update due to the parent's jth element.
```

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

update, for each element $\mathbf{V}_{\mathcal{P}(t)}(j,:)$ of the parent, the algorithm multiplies this vector with the rows of the corresponding matrices of the TTMV in $\delta(t)$ of t, then adds it to $\mathbf{V}_t(i,:)$.

For shared-memory parallel CP-ALS, we replace Line 9 of Algorithm 3 with a call to Smp-Dtree-Ttv(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 and factor matrices to all available processes. We discuss 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 straightforward lemma that enables us to distribute tensor nonzeros for parallelization.

Lemma 5 (Distributive property of TTVs). *Let $\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}}$, and $\boldsymbol{\mathcal{Z}}$ be tensors in $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{Y}} + \boldsymbol{\mathcal{Z}}$. Then, for any $n \in \mathbb{N}_N$ and $\boldsymbol{u} \in \mathbb{R}^{I_n}$ $\boldsymbol{\mathcal{X}} \times_n \boldsymbol{u} = \boldsymbol{\mathcal{Y}} \times_n \boldsymbol{u} + \boldsymbol{\mathcal{Z}} \times_n \mathbf{u}$ holds.*

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

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

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

Corollary 6. *Let $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_P$ be tensors in $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with $\sum_{i=1}^{P} \boldsymbol{\mathcal{X}}_i = \boldsymbol{\mathcal{X}}$. 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 $\boldsymbol{\mathcal{X}} \times_{i \neq n} = \boldsymbol{\mathcal{X}}_1 \times_{i \neq n} \boldsymbol{u}^{(i)} + \cdots + \boldsymbol{\mathcal{X}}_P \times_{i \neq n} \boldsymbol{u}^{(i)})$.*

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

4.2.1. Fine-grain parallelism. Corollary 6 allows us to partition the tensor $\boldsymbol{\mathcal{X}}$ in the sum form $\boldsymbol{\mathcal{X}}_1 + \cdots + \boldsymbol{\mathcal{X}}_P$ for any $P > 1$, then perform TTMVs on each tensor part $\boldsymbol{\mathcal{X}}_p$ independently, finally sum up these results to obtain the TTMV result for $\boldsymbol{\mathcal{X}}$ multiplied in $N-1$ modes. As $\boldsymbol{\mathcal{X}}$ 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 TTMV of $\boldsymbol{\mathcal{X}}_p$ 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 assembly 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 $\sigma_i^{(n)} = p$ implies that the final value of $\mathbf{M}^{(n)}(i,:)$ resides at the process p. We assume the same partition given by $\boldsymbol{\sigma}^{(n)}$ 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.

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

Input: $\boldsymbol{\mathcal{X}}_p$: A part of an N-mode tensor $\boldsymbol{\mathcal{X}}$
R: The rank of CP decomposition
For all $n \in \mathbb{N}_N$:
$\boldsymbol{\sigma}^{(n)}$: The vector indicating the owner process of each row of $\mathbf{U}^{(n)}$
$\mathcal{F}_p^{(n)}$: The index set with elements $i \in \mathcal{F}_p^{(n)}$ where $\sigma_i^{(n)} = p$
$\mathcal{G}_p^{(n)}$: The set of all unique indices of the nonzeros of $\boldsymbol{\mathcal{X}}_p$ in mode n
$\mathbf{U}^{(n)}(\mathcal{G}_p^{(n)},:)$: Distributed initial matrix (rows needed by process p)
Output: $[\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\boldsymbol{\mathcal{X}}$ with distributed $\mathbf{U}^{(n)}$

$\mathcal{T} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\boldsymbol{\mathcal{X}}_p$) ▶ The tree has leaves $\{l_1, \ldots, l_N\}$.
for $n = 2 \ldots N$ **do**
  $\mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)^T \; \mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)$)
repeat
  for $n = 1 \ldots N$ **do**
    for all $t \in \mathcal{T}$ **do** ▶ Invalidate tree tensors that are multiplied in mode n.
      if $n \in \mu'(t)$ **then**
        DESTROY($\mathbf{V}_t$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
    SMP-DTREE-TTV(l_n) ▶ Perform the TTMVs for the leaf node tensors.
    $\mathbf{M}^{(n)}(\mathcal{G}_p^{(n)},:) \leftarrow \mathbf{V}_{l_n}$ ▶ Form $\mathbf{M}^{(n)}$ using leaf tensors (done implicitly).
    COMM-FACTOR-MATRIX($\mathbf{M}^{(n)}$, "*fold*", $\mathcal{G}_p^{(d)}, \mathcal{F}_p^{(d)}, \boldsymbol{\sigma}^{(d)}$) ▶ Assemble $\mathbf{M}^{(n)}(\mathcal{F}_p^{(n)},:)$.
    $\mathbf{H}^{(n)} \leftarrow *_{i \neq n} \mathbf{W}^{(i)}$
    $\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:) \leftarrow \mathbf{M}^{(n)}(\mathcal{F}_p^{(n)},:)\mathbf{H}^{(n)\dagger}$
    $\boldsymbol{\lambda} \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
    COMM-FACTOR-MATRIX($\mathbf{U}^{(n)}$, "*expand*", $\mathcal{G}_p^{(d)}, \mathcal{F}_p^{(d)}, \boldsymbol{\sigma}^{(d)}$) ▶ Send/Receive $\mathbf{U}^{(n)}$.
    $\mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($[\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)]^T \; \mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)$)
until converge is achieved or the maximum number of iterations is reached.
return $[\![\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

This approach amounts to a fine-grain parallelism where each fine-grain computational task corresponds to performing TTMV operations due to a nonzero element $x_{i_1,\ldots,i_N} \in \boldsymbol{\mathcal{X}}$. Specifically, according to (10), 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 $\boldsymbol{\mathcal{X}}_p$ in order to perform its local TTMVs. For each dimension n, we represent the union of all these "required" row indices for the process p by $\mathcal{G}_p^{(n)}$. Similarly, we represent the set of "owned" rows by the process p by $\mathcal{F}_p^{(n)}$. In this situation, the set $\mathcal{G}_p^{(n)} \setminus \mathcal{F}_p^{(n)}$ correspond to the rows of $\mathbf{M}^{(n)}$ for which the process p generates a partial TTMV 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 TTMVs according to (10). These rows of $\mathbf{U}^{(n)}$ are similarly to be received from their owners in order to carry out the TTMVs at process p. Hence, a "good" partition in general involves a significant overlap of $\mathcal{G}_p^{(n)}$ and $\mathcal{F}_p^{(n)}$ 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 $\boldsymbol{\mathcal{X}}_p$, $\mathcal{F}_p^{(n)}$, and $\mathcal{G}_p^{(n)}$ are determined in the partitioning phase, and are provided as input to the algorithm. Each process starts with the subset $\mathcal{G}_p^{(n)}$ 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 $\boldsymbol{\mathcal{X}}_p$. 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 $[\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)]^T\ \mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)$ due to its owned rows. Afterwards, we perform an All-Reduce communication to sum up these local results to 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 Smp-Dtree-Ttv on the leaf node l_n,

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.
$\boldsymbol{\sigma}$: The ownership of each row of $\mathbf{M}$
$\mathcal{G}_p$: The rows used by process p
$\mathcal{F}_p$: The rows owned by process p
Output: Rows of $\mathbf{M}$ are properly communicated.

```
1:  if comm = "fold" then
2:     for all i ∈ G_p \ F_p do                ▶ Send non-owned rows to their owners.
3:        Send M(i,:) to the process σ_i.
4:     for all i ∈ F_p do                      ▶ Gather all partial results of owned rows together.
5:        Receive and sum up all partial results for M(i,:).
6:  else if comm = "expand" then
7:     for all i ∈ F_p do                      ▶ Send owned rows to all processes in need.
8:        Send M(i,:) to the all processes p' with i ∈ F_p'.
9:     for all i ∈ G_p \ F_p do                ▶ Receive rows that are needed locally.
10:       Receive M(i,:) from the process σ_i.
```

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 $\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)},:)$ using the recently assembled $\mathbf{M}^{(n)}(\mathcal{F}_p^{(n)},:)$. Once the new distributed $\mathbf{U}^{(n)}$ is computed, we normalize it column-wise and obtain the vector $\boldsymbol{\lambda}$ 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 $\mathcal{G}_p \setminus \mathcal{F}_p$ of rows to their owner processes, while similarly receiving all partial results for its set $\mathcal{F}_p$

of owned rows and summing them up. Symmetrically, in the fold communication, the process p sends the rows with indices $\mathcal{F}_p$, and receives the rows with indices $\mathcal{G}_p \setminus \mathcal{F}_p$. The exact set of row indices that needs to be communicated in fold and expand steps depends on the partitioning of $\boldsymbol{\mathcal{X}}$ 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. Another advantage of determining this set of rows to be communicated in advance is that it eliminates the need to store the vector $\boldsymbol{\sigma}^{(n)} \in \mathbb{R}^{I_n}$ which could otherwise be costly for a large tensor dimension.

4.2.2. Medium-grain parallelism. For an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ 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 $\boldsymbol{\mathcal{X}}$. Specifically, for each dimension n, the index set $\mathbb{N}_{I_n}$ is partitioned into P_n sets $\mathcal{S}_1^{(n)}, \ldots, \mathcal{S}_{P_n}^{(n)}$. With this partition, the process with the index $(p_1, \ldots, p_N) \in P_1 \times \cdots \times P_N$ gets $\boldsymbol{\mathcal{X}}(\mathcal{S}_{p_1}^{(1)}, \ldots, \mathcal{S}_{p_N}^{(n)})$ as its local tensor. Each factor matrix $\mathbf{U}^{(n)}$ is also partitioned following this topology where the set of rows $\mathbf{U}^{(n)}(\mathcal{S}_j^{(n)}, :)$ is owned by the processes with index $(p_1, \ldots, p_N)$ where $p_n = j, j \in \mathbb{N}_{P_n}$, 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 messages by a factor of P_n in the nth dimension.

One can design an algorithm specifically for the medium-grain decomposition [40]. 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 to capture 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 $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ 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 $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ with the vertex set $\mathcal{V}$ and the hyperedge set $\mathcal{E}$. The generalization of the model to higher order tensors should be clear from this construction.

The vertex set $\mathcal{V} = \mathcal{V}^{(1)} \cup \mathcal{V}^{(2)} \cup \mathcal{V}^{(3)} \cup \mathcal{V}^{(\boldsymbol{\mathcal{X}})}$ 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 $v_i^{(1)} \in \mathcal{V}^{(1)}$ for $i \in \mathbb{N}_{I_1}$, $v_j^{(2)} \in \mathcal{V}^{(2)}$ for $j \in \mathbb{N}_{I_2}$, and $v_k^{(3)} \in \mathcal{V}^{(3)}$ 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 $v_{i,j,k}^{(\boldsymbol{\mathcal{X}})}$, which we define for each nonzero $x_{i,j,k} \in \boldsymbol{\mathcal{X}}$.

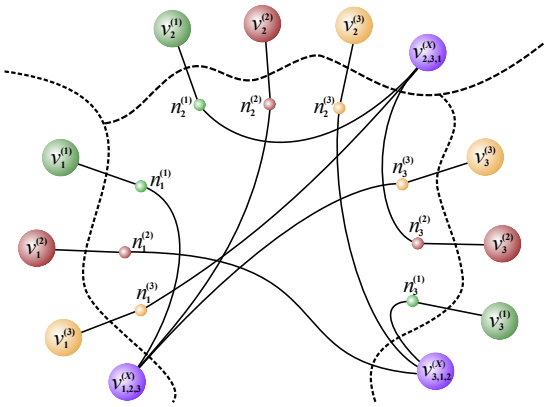


Fig. 2: Fine-grain hypergraph model for the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$ and a 3-way partition of the hypergraph. The objective is to minimize the cutsize of the partition while maintaining a balance on the total part weights corresponding to each vertex type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

This vertex type relates to the number of operations performed in TTMV due to the nonzero element $x_{i,j,k} \in \mathcal{X}$ in using (10) in all modes. In the N-dimensional case, this includes up to N vector 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 are carried out, and cannot be determined before a partitioning takes place. In our earlier work [26], this cost was exactly N Hadamard products per nonzero, as the MTTKRPs were computed without reusing partial results and without index compression after each TTMV. In the current case, we assign a cost of N to each vertex $v^{(\mathcal{X})}_{i,j,k}$ to represent an upper bound on the computational cost, and expect this to lead to a good load balance in practice. With these vertex definitions, one can use multi-constraint partitioning (6) with one constraint per vertex type. In this case, the first, the second, and the third types have unit weights in the first, second, and third constraints, respectively, and zero weight in all other constraints. The fourth vertex type also gets a unit weight (N, or equivalently, 1) in the fourth constraint, and zero weight for others. Here, balancing the first three constraints 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 TTMVs.

As TTMVs are carried out using (10), 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 $\mathcal{E} = \mathcal{E}^{(1)} \cup \mathcal{E}^{(2)} \cup \mathcal{E}^{(3)}$ to represent these dependencies as follows: $\mathcal{E}^{(1)}$ contains a hyperedge $n^{(1)}_i$ for each matrix row $\mathbf{U}^{(1)}(i,:)$, $\mathcal{E}^{(2)}$ contains a hyperedge $n^{(2)}_j$ for each row $\mathbf{U}^{(2)}(j,:)$, and $\mathcal{E}^{(3)}$ contains a hyperedge $n^{(3)}_k$ for each row $\mathbf{U}^{(3)}(k,:)$. Initially, $n^{(1)}_i$, $n^{(2)}_j$ and $n^{(3)}_k$ contain the corresponding vertices $v^{(1)}_i$, $v^{(2)}_j$, and $v^{(3)}_k$, as the owner of a matrix row has a dependency to it by default. In computing the MTTKRP using (10), 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 $v^{(\mathcal{X})}_{i,j,k}$ to the hyperedges $n^{(1)}_i$, $n^{(2)}_j$ and $n^{(3)}_k$ to model this dependency. In Figure 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 simplicity. 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 $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ where each part is associated with a unique process to obtain a P-way parallel execution of Algorithm 5. We consider the first subiteration of Algorithm 5 that updates $\mathbf{U}^{(1)}$, and assume that each process already has all data elements to carry out the local TTMVs 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 Line 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 $v_{i,j,k}^{(\boldsymbol{\mathcal{X}})} \in n_i^{(1)}$ which resides at the process p; therefore, this communication is accurately represented in the *connectivity* $\kappa_{n_i^{(1)}}$ of the hyperedge $n_i^{(1)}$. In general, the hyperedge $n_i^{(1)}$ incurs $\kappa_{n_i^{(1)}} - 1$ messages to transfer the partial results for the matrix row $\mathbf{M}^{(1)}(i,:)$ to the process q at Line 3. Therefore, the *connectivity-1 cutsize* metric (5) over the hyperedges exactly encodes the total volume of messages sent at Line 3, if we set $c[\cdot] = R$. Since the send operations at Line 8 are duals of the send operations at Line 3, the total volume of messages sent at Line 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 $\mathcal{H} = (\mathcal{V}^{(\boldsymbol{\mathcal{X}})}, \mathcal{E})$ using just one load constraint due to the vertices in $\mathcal{V}^{(\boldsymbol{\mathcal{X}})}$, 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 well 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.

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 TTMV 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 TTMV 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 high computational and memory requirements.

Label propagation-like: Given that the standard partitioning approach is too costly in practice, we developed a fast hypergraph partitioning heuristic which has reasonable memory and computational costs. The method is based on the balanced

label-propagation algorithm [38, 45], and includes some additional adaptations to handle hypergraphs [9, 20]. 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 $2|\mathcal{V}| + |\mathcal{E}| + 4P$. Running the algorithm for a few iterations provides reasonably good partitions. This basic algorithm can have many 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 $P = P_1P_2P_3$, 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_1P_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_1P_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 [40] 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 [26, 40]. 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. We briefly mention these here and refer the reader to the original sources for details. In [4],

Bader and Kolda show how to efficiently carry out MTTKRP as well as other fundamental tensor operations on sparse tensors in MATLAB. GigaTensor [22] 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 [40] is an efficient parallelization of MTTKRP and CP-ALS both in shared [42] and distributed memory [40] 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 [41]. 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 using CP formulation [23] in which they replicate the entire factor matrix $\mathbf{U}^{(n)}$ among MPI processes unlike our approach, and report scalability results only up to 100 cores. For computing the CP decomposition of dense tensors, Phan et al. [35] proposes a scheme that divides the tensor modes into two sets, pre-computes the TTMVs for each mode set, and finally reuses these partial results to obtain the final MTTKRP result in each mode. This provides a factor of 2 improvement in the number of TTMVs over the traditional approach, and our dimension tree-based framework can be considered as the generalization of this approach that provides a factor of $N/\log N$ improvement.

While this paper was under evaluation, another paper appeared [33]. Li et al. use the same idea of storing intermediate tensors but use a different formulation based on tensor times tensor multiplication and a tensor times matrix through Hadamard products for shared memory systems. The overall approach is similar to that by Phan et al. [35], where the difference lies in the application of the method to sparse tensors and auto-tuning to better control memory use and gains in the operation counts.

Aside from CP decomposition, Baskaran et al. [6] provide a shared-memory parallel implementation for the Tucker decomposition of sparse tensors. We [27] 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. [34] investigate an 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 compare the medium- and the fine-grain distributed memory parallel algorithms.

6.1. Dataset and environment.

We experimented with five real-world tensors whose sizes are shown in Table 1. Netflix tensor has user $\times$ movie $\times$ time dimensions, which we formed from the data of the Netflix Prize competition [7]. 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 $(\text{entity}, \text{relation}, \text{entity})$ 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

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

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 form (time × users × resources × tags); hence, we form 4-mode tensors out of these tuples. We obtained the Amazon review dataset from SNAP [32], which contains product review texts by users. We first processed this dataset with the standard text processing routines. We used the `nltk` package [8] 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. Then, we created a three dimensional tensor whose dimensions correspond to the users, products, and retained words. Numerical values are set to 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 CPU sockets (Intel(R) Xeon(R) 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 each socket has L1, L2, L3 caches of sizes of 32KB, 256KB, and 35MB, respectively. All codes are compiled with gcc/g++-5.3.0 using OpenMP directives and compiler options -O3, -ffast-math, -funroll-loops, -ftree-vectorize, -fstrict-aliasing on this shared memory system. The distributed memory system is an IBM Blue Gene/Q cluster. This system consists of 6 racks of 1024 nodes with each node having 16GB of memory and a 16-core IBM PowerPC A2 processor running at 1.6GHz. 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.

We also used synthetic tensors created randomly having 4, 8, 16, and 32 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.

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 [40], a C code with OpenMP and MPI parallelizations. The "winner" for each test case in the results are highlighted with bold font.

6.2. Shared memory experiments. We compare the shared memory performance of the dimension tree-based CP-ALS algorithm with the state of the art. We experimented with four methods called **ht-tree2**, **ht-tree3**, **ht-tree**, and **splatt**. The **ht-tree** method, implemented in HyperTensor, uses a full BDT to carry out

TTMVs. The **ht-tree2** method is the same implementation as **ht-tree** except that it uses a 2-level dimension tree. In this tree, N leaf nodes are directly connected to the root, hence no intermediate results are generated. However, TTMVs are performed one mode at a time to benefit from the index compression to reduce the operation count. As a result, this method performs $N-1$ TTMVs for each mode in an iteration just as SPLATT; we thus expect comparable performance. The **ht-tree3** method uses a three level BDT whose second level has two nodes holding the partial TTMV results corresponding to the first and the second half of the set of dimensions. This is analogous to the approach by Phan et al. [35] for computing dense CP decompositions, but it also employs our data structure and shared memory parallelization for sparse tensors. Storing partial results in the second level reduces the number of TTMVs by a factor of 2, but for an N-mode tensor, this method still performs $O(N^2)$ TTMVs per iteration. Note that for 3- and 4-dimensional tensors, **ht-tree3** and **ht-tree** use identical trees, thus give the same results. These results for **ht-tree3** are indicated with an asterisk in the tables. Finally, **splatt** corresponds to the parallel CP-ALS implementation in SPLATT. We ran all algorithms for 20 iterations with the rank of approximation $R = 20$ (except for the sequential execution of 16- and 32-dimensional random tensors, which are run for 2 iterations due to their cost), and recorded the average time spent per CP-ALS iteration. Test instances in which a method gets out of memory are indicated with a dash symbol.

6.2.1. Sequential execution. In Table 2, we give the sequential per-iteration run time of all methods. We report the run time in seconds for **splatt**, and the relative speedup with respect to **splatt** for the other three methods. We first note that **ht-tree2** runs slightly slower than **splatt** on three dimensional Amazon and NELL tensors (0.99x and 0.87x), and notably slower on Netflix tensor (0.60x). This is so because SPLATT has a specially tuned implementation for 3-dimensional tensors, whereas we use a single code for all dimensions. On all higher dimensional tensors, **ht-tree2** 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. The gap between **splatt** and **ht-tree2** narrows using Random16D, as **ht-tree2** depletes the memory in one NUMA node, which is discussed more in subsection 6.2.4, and starts accessing the distant memory in the other NUMA node. **ht-tree2** gets out of memory using Random32D as it stores $O(N^2)$ index arrays.

We now measure the effect of dimension trees by comparing **ht-tree** with **ht-tree2** in Table 2. These two methods use the same TTMV implementation, whereas **ht-tree** uses a full BDT. On Delicious, Flickr, Netflix, and NELL, **ht-tree** obtains 1.78x, 1.61x, 1.63x, and 1.47x speedup over **ht-tree2** thanks to the BDT. Likewise, on random tensors, we observed 1.43x, 1.91x, 3.01x speedup on tensors Random4D, Random8D, and Random16D, respectively, using **ht-tree**. This validates our performance expectation (Theorem 2) that as the dimensionality of the tensor increases, a BDT results in significantly fewer TTMVs hence better performance.

Comparing **ht-tree** with **splatt** similarly yields a speedup of 1.98x, 1.98x, 1.28x, 2.05x, 3.97x, 3.94x, and 5.96x on tensors Delicious, Flickr, NELL, Random4D, Random8D, Random16D, and Random32D, respectively, which similarly meets our expectation of performance gain from Theorem 2. On Amazon, **ht-tree** was only 2% faster, whereas on Netflix, **splatt** was only 2% faster, which was the only instance in which **splatt** had a slight edge over **ht-tree**.

Finally, we note that the performance gap between **ht-tree** and **ht-tree3** widens significantly as the tensor gets higher dimensional. Using Random8D, Random16D,

Table 2: Sequential CP-ALS run time per iteration. Timings are in seconds for **splatt**, whereas we report the relative speedup with respect to **splatt** for other methods.

	splatt	**ht-tree2**	**ht-tree3**	**ht-tree**
Delicious	66.6	1.11	*	**1.98**
Flickr	43.6	1.23	*	**1.98**
Netflix	**8.2**	0.60	*	0.98
NELL	8.3	0.87	*	**1.28**
Amazon	214.6	0.99	*	**1.02**
Random4D	224.7	1.43	*	**2.05**
Random8D	1527.1	2.08	2.70	**3.97**
Random16D	4401.6	1.31	2.02	**3.94**
Random32D	19919.9	-	2.38	**5.96**

Table 3: Shared memory parallel CP-ALS run time per iteration (in seconds). Timings are in seconds for **splatt**, whereas we report the relative speedup with respect to **splatt** for other methods.

	splatt	**ht-tree2**	**ht-tree3**	**ht-tree**
Delicious	8.3	0.93	*	**2.00**
Flickr	5.8	1.09	*	**1.81**
Netflix	**0.7**	0.55	*	0.87
NELL	1.3	1.11	*	**1.46**
Amazon	**24.4**	0.86	*	0.95
Random4D	20.1	0.91	*	**1.47**
Random8D	86.9	0.81	1.69	**2.18**
Random16D	349.2	0.82	1.73	**3.47**
Random32D	1601.8	-	2.18	**5.65**

and Random32D, **ht-tree** is 1.47x, 1.95x, and 2.50x faster than **ht-tree3** as it incurs significantly fewer TTMVs. These results suggest that using a full BDT is indeed the ideal choice for performance.

6.2.2. Shared memory parallel execution. In Table 3, we give the 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 execution times. Using **ht-tree**, we obtain 2.15x, 1.66x, 1.58x, 1.32x, and 1.10x speedup over **ht-tree2** on Delicious, Flickr, Netflix, NELL, and Amazon tensors, respectively. For random tensors, we get 1.62x, 2.69x, and 4.23x speedup on Random4D, Random8D, and Random16D. Comparing **ht-tree** with **splatt**, we observe a speedup of 2.00x, 1.81x, 1.46x, 1.47x, 2.18x, 3.47x, and 5.65x on tensors Delicious, Flickr, NELL, Random4D, Random8D, Random16D, and Random32D, respectively. This demonstrates that the use of a BDT in CP-ALS computations can be effectively parallelized in a shared memory setting, on top of significantly reducing the amount of TTMV work. On three dimensional Amazon and Netflix tensors, **splatt** has a slight edge over **ht-tree** by 5% and 14% faster executions, respectively. Another point to note is that **splatt** has somewhat better parallel speedup in general (over its own sequential run time) than **ht-tree2** and **ht-tree**. This is mostly due to the fact that TTMV is a memory-bound computation; hence, once the memory bandwidth is fully utilized, one cannot expect further speedup through multi-threading. When performing TTMVs, our implementation makes slightly more memory accesses due to extra pointer arrays involved in the dimension tree nodes, which saturates the bandwidth earlier and

Table 4: Symbolic precomputation timings. We report the exact timing for **splatt** in seconds, and relative timing with respect to **splatt** for other methods, indicating the ratio at which **splatt** is faster than these methods this precomputation.

	splatt	**ht-tree2**	**ht-tree3**	**ht-tree**
Delicious	**87.3**	2.41	*	1.39
Flickr	**57.1**	2.70	*	1.32
Netflix	**51.6**	1.96	*	1.48
NELL	**37.7**	1.63	*	1.47
Amazon	**720.3**	1.85	*	1.58
Random4D	**62.9**	2.21	*	2.40
Random8D	**86.2**	6.36	4.35	3.97
Random16D	**233**	15.71	4.32	3.67
Random32D	**638.7**	-	4.85	3.07

thereby affects the parallel speedup to a certain extent. Nevertheless, using **ht-tree** we achieve up to 5.65x faster runs over **splatt** in a shared memory parallel execution.

Conformally with the sequential case, **ht-tree** gets significantly faster than **ht-tree3** as the tensor dimensionality increases, up to 2.59x using Random32D, which demonstrates the effectiveness of using a full BDT.

6.2.3. Preprocessing cost. In Table 4, we provide symbolic TTV costs of our methods as well as the precomputation cost of **splatt** for setting up its data structures. All runtimes are for a sequential execution; neither SPLATT nor HYPERTENSOR parallelizes this step in their current version. We first note that **ht-tree** incurs significantly less cost than **ht-tree2** in all instances. This is expected as **ht-tree2** has $O(N^2)$ index arrays to be sorted, whereas **ht-tree** has only $O(N \log N)$ of them. For the same reason, **ht-tree** gets notably faster than **ht-tree3** as the tensor dimensionality increases to 32. The cost is comparable between **splatt** and **ht-tree** for Delicious, Flickr, Netflix, NELL, and Amazon tensors, but **splatt** takes significantly less time for higher dimensional random tensors, up to 3.97x on Random8D, as it sorts only $O(N)$ arrays.

Comparing these timings with the iteration times in Table 2, we see that this precomputation is amortized in a few iterations, except for Netflix and NELL. In practice, CP-ALS is typically executed multiple times with different initial matrices and ranks of approximation using the same symbolic dimension tree construct, which should render this preprocessing cost less important even for these two tensors.

6.2.4. Memory usage. We provide the memory consumption of all methods in Table 5. The first column corresponds to the amount of memory used to store factor matrices, which is common to all methods. We give the memory usage for storing index arrays in GBs for **splatt**, and as the ratio to **splatt** for all other methods. We also give the memory consumption for storing the value matrices of intermediate tensors for **ht-tree3** and **ht-tree** in GBs. We first note that **ht-tree2** uses the highest amount of memory to store index arrays as expected, up to 8.32 times more than **splatt** on Random16D. In all tensors, the amount of index memory used by **ht-tree** is only slightly higher than **splatt**, the worst case being Flickr tensor for which **ht-tree** consumes 1.35 times more memory in comparison. There is a multitude of reasons for this observation. First, even though **splatt** uses only $O(N)$ index arrays, we realized upon inspecting the implementation that it uses two different representations of a tensor for faster execution, effectively doubling its memory requirements.

Table 5: Memory usage of different methods. Index usages of **ht-tree2**, **ht-tree3**, and **ht-tree** are reported with respect to that of **splatt**, whereas the memory usage of factors, value matrices, and **splatt** index are in GBs.

	factors	**splatt**	**ht-tree2**	**ht-tree3**		**ht-tree**	
		index	index	index	value	index	value
Delicious	3	**7**	2.44	*	*	1.21	5.7
Flickr	4.5	**4.8**	2.29	*	*	1.35	4.3
Netflix	0.1	**3.4**	1.85	*	*	1.29	1.4
NELL	0.6	**2.5**	1.88	*	*	1.28	0.5
Amazon	1.4	**49.6**	1.84	*	*	1.32	65.4
Random4D	6	**9.1**	2.21	*	*	1.12	10.2
Random8D	11.9	**21**	4.24	1.08	15.3	1.08	30.5
Random16D	23.8	**44.9**	8.32	1.47	15.3	1.23	45.7
Random32D	47.7	**94.3**	-	2.40	15.3	1.31	61

It also employs two arrays per dimension in its compressed sparse fiber (CSF) tensor storage [39], doubling the memory consumption. Finally, its memory efficiency depends heavily on the index overlaps after TTMVs, which happens rarely for high dimensional random tensors.

Aside from this, we note that the amount of memory used to store the values of intermediate tensors is reasonable, Random8D being the only exception in which the value matrix size exceeds the index size using **ht-tree**. Finally, **ht-tree3** uses more memory for index storage than **ht-tree** for high dimensional tensors, as it uses more index arrays similar to **ht-tree2**. It uses less space for value matrices, however, as it has only one level for intermediate tensors.

All in all, we conclude upon considering these results that **ht-tree** provides remarkable performance improvements with a reasonable increase in the memory usage.

6.3. Distributed memory experiments. 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 BDTs for local TTMVs, and is slower, comparing it against HyperTensor's fine-grain implementation which has faster local TTMVs would not be fair, nor would correctly reveal the effect of different partitioning strategies. Second, we were not able to get Splatt to 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 [40], 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 TTMV 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 measuring the influence of medium- and fine-grain algorithms and associated partitionings on parallel scalability.

We investigate the performance in two tables. In Table 6, we give the strong scalability 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

Table 6: Per iteration speedup results for distributed memory parallel CP-ALS using different partitions. The best single threaded execution is given in seconds (shaded cells), and the relative speedups are reported for all other cases. *#nodes* and *#cores* correspond to the number of nodes (and equivalently, MPI ranks) and cores per node used in each instance, respectively.

#nodes×*#cores*	Delicious				Flickr			
	med-gd	**fine-rd**	**fine-lb**	**fine-hp**	**med-gd**	**fine-rd**	**fine-lb**	**fine-hp**
8 × 1	**45.078**	0.65	1.00	1.00	0.86	-	**25.674**	0.99
8 × 16	14.12	4.64	12.51	**18.13**	10.72	-	12.77	**16.98**
16 × 16	17.41	6.30	19.62	**31.00**	16.02	2.71	23.53	**29.78**
32 × 16	27.96	9.02	30.92	**51.64**	23.30	4.03	36.99	**53.60**
64 × 16	34.28	15.51	50.14	**83.63**	25.34	7.08	72.73	**89.77**
128 × 16	54.84	25.83	84.57	**128.43**	39.38	13.43	115.65	**143.43**
256 × 16	74.76	40.43	141.75	**183.24**	46.34	23.49	148.40	**178.29**

#nodes×*#cores*	Netflix				NELL			
	med-gd	**fine-rd**	**fine-lb**	**fine-hp**	**med-gd**	**fine-rd**	**fine-lb**	**fine-hp**
4 × 1	**27.656**	0.86	0.98	0.94	**19.540**	0.87	0.97	0.98
4 × 16	**25.26**	16.41	23.22	24.18	16.41	10.14	16.22	**17.62**
8 × 16	**44.82**	22.18	39.68	40.20	26.09	13.43	27.91	**28.69**
16 × 16	**76.82**	27.99	65.69	65.85	43.52	19.58	**45.13**	44.01
32 × 16	**124.58**	34.40	103.58	105.96	68.08	26.66	**69.29**	61.45
64 × 16	**200.41**	41.90	155.37	159.86	**109.16**	36.39	97.21	81.76
128 × 16	264.49	52.68	236.37	**271.14**	**153.86**	49.10	127.71	126.06
256 × 16	**321.58**	67.45	236.37	260.91	**197.37**	61.84	157.58	164.20

#nodes×*#cores*	Amazon		
	med-gd	**fine-rd**	**fine-lb**
64 × 1	-	0.55	**36.303**
64 × 16	-	4.12	**19.37**
128 × 16	31.82	5.86	**36.05**
256 × 16	34.38	8.58	**63.69**

in Table 7, we give the detailed load balance and communication cost metrics just for this case.

In Table 6, we compare the execution of Algorithm 5 with three partitioning methods. The **fine-hp** and **fine-lb** correspond, respectively, to the standard hypergraph partitioning and the label-propagation-like heuristic of subsection 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 was too big for PaToH to handle. 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 [40]. The **fine-rd** method refers to the random partitioning of the fine-grain hypergraph, which is given as a reference to illustrate the impact of a good partitioning. 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. We use the run time of the single threaded execution of the fastest method as our baseline (highlighted with shaded cells) in computing the parallel speedup. In passing from one core per node to 16 cores per node, we see some speedup over 16 in Table 6; this is because we use two threads per core (see subsection 6.1).

In order to be able to analyze the speedup results of Table 6, 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 7. 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

Table 7: 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**

gracefully translates into balancing the actual computational load.

Using 256 MPI ranks on Delicious, **fine-hp** and **fine-lb** are 2.5x and 1.9x faster over **med-gd**. We observe in Table 7 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 the communication volume of **fine-rd**. In overall, on Delicious **fine-hp** and **fine-lb** obtain 183x and 142x speedup using 4096 cores, whereas **med-gd** and **fine-rd** give 75x and 40x speedup for the same tensor. On Flickr, **fine-hp** and **fine-lb** similarly yield 178x and 148x speedup using 4096 cores, while **med-gd** and **fine-rd** could achieve 46x and 23x 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 speedup. The **fine-hp** and **fine-lb** result in 11.33x and 10.11x speedup for Delicious, and 11.62x and 10.50x speedup for Flickr in the same scenario. **fine-rd** is significantly slower than the other methods, incurring the highest communication as shown in Table 7.

On Netflix and NELL, **med-gd** yields 321x and 197x speedup using 4096 cores. **fine-hp** shows a comparable performance with 261x and 164x speedup, whereas **fine-lb** is slightly slower than **fine-hp** with 236x and 158x speedup. 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 speedup for Netflix and NELL, respectively. The **fine-hp** and **fine-lb** partitioning result in 10.18x and 11.18x speedup for Netflix, and 9.72x and 9.32x speedup for NELL in the same scenario. **fine-rd** is similarly the slowest of all methods giving 67x and 62x speedup for these two tensors. Using Netflix and NELL, **med-**

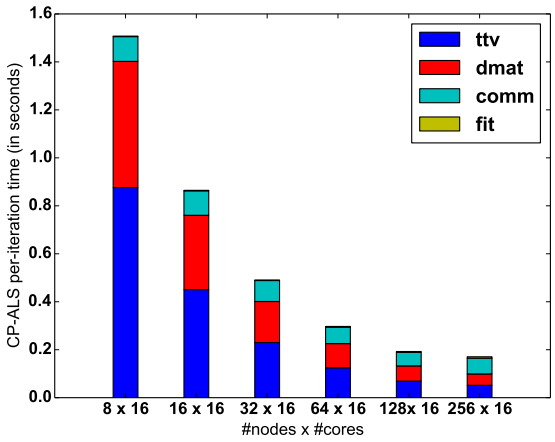


Fig. 3: Running time dissection of a parallel CP-ALS iteration using **fine-hp** and **ht-tree** scheme. The legends **ttv**, **dmat**, **comm**, and **fit** correspond to the time spent for TTMVs, dense matrix operations following the TTMVs, communication, and fit computation. The time for fit computation not discernible in the plot.

gd gets 23% and 20% faster than **fine-hp**, and 36% and 25% faster than **fine-lb**. In Table 7, we see that this is due to **med-gd** incurring smaller maximum communication volume and having fewer messages. 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 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. We observe in Table 7 that this is due to 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. 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 overall, **med-gd** gives 34x speedup over the baseline whereas **fine-lb** gets significantly faster with 63x speedup.

In Figure 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 TTMVs, the TTMV

step still remains to be the dominant computational cost. In this figure, we first observe that the workload due to TTMV and dense matrix computations decrease with the increasing number of processes. 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 computation takes a negligible amount of time hence is 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 TTMV 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.

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 (2011), pp. 67–86.
[2] C. A. Andersson and R. Bro, *The N-way toolbox for MATLAB*, Chemometrics and Intelligent Laboratory Systems, 52 (2000), pp. 1–4.
[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, 2016, pp. 912–922.
[4] B. W. Bader and T. G. Kolda, *Efficient MATLAB computations with sparse and factored tensors*, SIAM Journal on Scientific Computing, 30 (2007), pp. 205–231.

[5] B. W. Bader, T. G. Kolda, et al., *Matlab tensor toolbox version 2.6*. Available online http://www.sandia.gov/~tgkolda/TensorToolbox/, February 2015.
[6] 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), Sept 2012, pp. 1–6.
[7] J. Bennett and S. Lanning, *The Netflix Prize*, in Proceedings of KDD cup and workshop, vol. 2007, 2007, p. 35.
[8] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*, O'Reilly Media Inc., 2009.
[9] J. Buurlage, *Self-improving sparse matrix partitioning and bulk-synchronous pseudo-streaming*, master's thesis, Utrecht University, 2016.
[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, vol. 5, 2010, p. 3.
[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 (1970), pp. 283–319.
[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, Denver, Colorado, 2001, p. 42.
[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 (2010), pp. 656–683.
[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, Montreal, Quebec, Canada, 2014, pp. 1296–1304.
[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, Berkeley, CA, USA, 2008, USENIX Association, p. 19.
[18] L. Grasedyck, *Hierarchical singular value decomposition of tensors*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 2029–2054.
[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 (1970), pp. 1–84.
[20] V. Henne, *Label propagation for hypergraph partitioning*, master's thesis, Karsruhe Institute of Technology, Germany, 2015.
[21] J. Håstad, *Tensor rank is NP-complete*, Journal of Algorithms, 11 (1990), pp. 644–654.
[22] 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, New York, NY, USA, 2012, ACM, pp. 316–324.
[23] L. Karlsson, D. Kressner, and A. Uschmajew, *Parallel algorithms for tensor completion in the CP format*, Parallel Computing, 57 (2016), pp. 222–234.
[24] G. Karypis and V. Kumar, *Multilevel algorithms for multi-constraint hypergraph partitioning*, Tech. Report 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998.
[25] O. Kaya and B. Uçar, *High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors*, Tech. Report RR-8801, Inria, Oct 2015.
[26] 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, New York, NY, USA, 2015, ACM, pp. 77:1–77:11.
[27] 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), Aug 2016, pp. 103–112.
[28] T. G. Kolda and B. Bader, *The TOPHITS model for higher-order web link analysis*, in Proceedings of Link Analysis, Counterterrorism and Security, 2006.
[29] T. G. Kolda and B. Bader, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.

[30] 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, vol. 67, 1998, pp. 1–16.
[31] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley–Teubner, Chichester, U.K., 1990.
[32] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*. http://snap.stanford.edu/data, June 2014.
[33] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, *Model-driven sparse CP decomposition for higher-order tensors*, in IPDPS 2017, 31th IEEE International Symposium on Parallel and Distributed Processing, Orlando, FL, USA, May 2017, pp. 1048–1057.
[34] 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, Nov 2015, pp. 943–948.
[35] A. H. Phan, P. Tichavský, and A. Cichocki, *Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations*, IEEE Transactions on Signal Processing, 61 (2013), pp. 4834–4846.
[36] 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, New York, NY, USA, 2010, ACM, pp. 81–90.
[37] 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, New York, NY, USA, 2009, ACM, pp. 727–736.
[38] 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), IEEE, Oct. 2014, pp. 481–490.
[39] S. Smith and G. Karypis, *Tensor-matrix products with a compressed sparse tensor*, in Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, ACM, 2015, p. 7.
[40] 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, 2016, pp. 902–911.
[41] 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).
[42] 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, Hyderabad, India, May 2015, IEEE Computer Society, pp. 61–70.
[43] P. Symeonidis, A. Nanopoulos, and Y. Manolopoulos, *Tag recommendations based on tensor dimensionality reduction*, in Proceedings of the 2008 ACM Conference on Recommender Systems, New York, NY, USA, 2008, ACM, pp. 43–50.
[44] G. Tomasi and R. Bro, *A comparison of algorithms for fitting the parafac model*, Computational Statistics & Data Analysis, 50 (2006), pp. 1700 – 1734.
[45] 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, New York, NY, USA, 2013, ACM, pp. 507–516.
[46] M. A. O. Vasilescu and D. Terzopoulos, *Multilinear analysis of image ensembles: TensorFaces*, in Computer Vision—ECCV 2002, Springer, 2002, pp. 447–460.
[47] 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, NY, USA, 2010, ACM, pp. 737–738.