

# Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees

Oguz Kaya, Bora Uçar

► **To cite this version:**

Oguz Kaya, Bora Uçar. Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees. *SIAM Journal on Scientific Computing*, Society for Industrial and Applied Mathematics, 2018, 40 (1), pp.C99 - C130. 10.1137/16M1102744 . hal-01397464v2

**HAL Id: hal-01397464**

**<https://hal.inria.fr/hal-01397464v2>**

Submitted on 17 Oct 2018

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

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

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

3                           OGUZ KAYA<sup>†</sup> AND BORA UÇAR<sup>‡</sup>

4       **Abstract.** CANDECOMP/PARAFAC (CP) decomposition of sparse tensors has been success-  
5 fully applied to many problems in web search, graph analytics, recommender systems, health care  
6 data analytics, and many other domains. In these applications, efficiently computing the CP de-  
7 composition of sparse tensors is essential in order to be able to process and analyze data of massive  
8 scale. For this purpose, we investigate an efficient computation of the CP decomposition of sparse  
9 tensors and its parallelization. We propose a novel computational scheme for reducing the cost of  
10 a core operation in computing the CP decomposition with the traditional alternating least squares  
11 (CP-ALS) based algorithm. We then effectively parallelize this computational scheme in the context  
12 of CP-ALS in shared and distributed memory environments, and propose data and task distribution  
13 models for better scalability. We implement parallel CP-ALS algorithms and compare our imple-  
14 mentations with an efficient tensor factorization library using tensors formed from real-world and  
15 synthetic datasets. With our algorithmic contributions and implementations, we report up to 5.96x,  
16 5.65x, and 3.9x speedup in sequential, shared memory parallel, and distributed memory parallel  
17 executions over the state of the art, and achieve strong scalability up to 4096 cores on an IBM  
18 BlueGene/Q supercomputer.

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

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

21       **1. Introduction.** With growing features and dimensionality of data, tensors, or  
22 multi-dimensional arrays, have been increasingly used in many fields including the  
23 analysis of Web graphs [28], knowledge bases [10], recommender systems [36, 37, 43],  
24 signal processing [30], computer vision [46], health care [34], and many others [29].  
25 Tensor decomposition algorithms are used as an effective tool for analyzing data in  
26 order to extract latent information within the data, or predict missing data elements.  
27 There have been considerable efforts in designing numerical algorithms for different  
28 tensor decomposition problems (see the survey [29]), and algorithmic and software  
29 contributions go hand in hand with these efforts [2, 5, 16, 22, 26, 27, 42, 40].

30       One of the well known tensor decompositions is the CANDECOMP/PARAFAC  
31 (CP) formulation, which approximates a given tensor as a sum of rank-one tensors.  
32 Among the commonly used algorithms for computing a CP decomposition is CP-  
33 ALS [11, 19], which is based on the alternating least squares method, though other  
34 variants also exist [1, 44]. These algorithms are iterative, in which the computa-  
35 tional core of each iteration involves a special operation called matricized tensor-times  
36 Khatri-Rao product (MTTKRP). When the input tensor is sparse and  $N$  dimensional,  
37 MTTKRP operation amounts to element-wise multiplication of  $N - 1$  row vectors from  
38  $N - 1$  matrices and their scaled sum reduction according to the nonzero structure of the  
39 tensor. As the dimensionality of the tensor increases, this operation gets computationally  
40 more expensive; hence, efficiently carrying out MTTKRP for higher dimensional  
41 tensors is of our particular interest in emerging applications [34]. This operation  
42 has received recent interest for efficient execution in different settings such as MAT-  
43 LAB [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].

<sup>†</sup>INRIA and LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France ([oguz.kaya@ens-lyon.fr](mailto:oguz.kaya@ens-lyon.fr)).

<sup>‡</sup>CNRS and LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France ([bora.ucar@ens-lyon.fr](mailto:bora.ucar@ens-lyon.fr))

44 We are interested in a fast computation of MTTKRP as well as CP-ALS for sparse  
 45 tensors using efficient computational schemes and effective parallelization in shared  
 46 and distributed memory environments.

47 Our contributions in this paper are as follows. We investigate the parallelization  
 48 of CP-ALS algorithm for sparse tensors in shared and distributed memory systems.  
 49 For the shared-memory computations, we propose a novel computational scheme that  
 50 significantly reduces the computational cost while offering an effective parallelism. We  
 51 then perform theoretical analyses corresponding to the computational gains and the  
 52 memory utilization, which are also validated with the experiments. We propose a fine-  
 53 grain distributed memory parallel algorithm, and compare it against a medium-grain  
 54 variant [40]. Finally, we discuss effective partitioning routines for these algorithms.  
 55 Even though the discussion is confined to CP-ALS in the paper, the contributions  
 56 apply to any other algorithm involving MTTKRP in its core [1].

57 The organization of the rest of the paper is as follows. In the next section, we  
 58 introduce our notation, describe the CP-ALS method, and present a data structure  
 59 called dimension tree which enables efficient CP-ALS computations. Next, in [sec-](#)  
 60 [tion 3](#), we explain how to use dimension trees to carry out MTTKRPs within CP-ALS.  
 61 Afterwards, we discuss an effective shared and distributed memory parallelization of  
 62 CP-ALS iterations in [section 4](#). We discuss partitioning methods pertaining to dis-  
 63 tributed memory parallel performance, and use these partitioning methods in our  
 64 experiments using real-world tensors. In [section 5](#), we give an overview of the existing  
 65 literature. Finally, we present experimental results in [section 6](#) to demonstrate per-  
 66 formance gains using our algorithms with shared and distributed memory parallelism  
 67 over an efficient state of the art implementation, and then conclude the paper.

## 68 2. Background and notation.

69 **2.1. Tensors and CP-ALS.** We denote the set  $\{1, \dots, M\}$  of integers as  $\mathbb{N}_M$   
 70 for  $M \in \mathbb{Z}^+$ . For vectors, we use bold lowercase Roman letters, as in  $\mathbf{x}$ . For matrices,  
 71 we use bold uppercase Roman letters, e.g.,  $\mathbf{X}$ . For tensors, we generally follow the  
 72 notation in Kolda and Bader’s survey [29]; particularly, we use bold calligraphic fonts,  
 73 e.g.,  $\mathcal{X}$ , to represent tensors. The *order* of a tensor is defined as the number of its  
 74 *dimensions*, or equivalently, *modes*, which we denote by  $N$ . A *slice* of a tensor in the  
 75  $n$ th mode is a set of tensor elements obtained by fixing the index only along the  $n$ th  
 76 mode. We use the MATLAB notation to refer to matrix rows and columns as well  
 77 as tensors slices, e.g.,  $\mathbf{X}(i, :)$  and  $\mathbf{X}(:, j)$  are the  $i$ th row and the  $j$ th column of  $\mathbf{X}$ ,  
 78 whereas  $\mathcal{X}(:, :, k)$  represents the  $k$ th slice of  $\mathcal{X}$  in the third dimension. We use italic  
 79 lowercase letters with subscripts to represent vector, matrix, and tensor elements,  
 80 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  $\mathcal{X}$ .  
 81 For the column vectors of a matrix, we use the same letter in lowercase and with a  
 82 subscript corresponding to the column index, e.g.,  $\mathbf{x}_i$  to denote  $\mathbf{X}(:, i)$ , whereas a row  
 83 of a matrix is always expressed in MATLAB notation, as in  $\mathbf{X}(i, :)$ . Sets, lists, trees,  
 84 and hypergraphs are expressed in non-bold calligraphic fonts.

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

$$88 \quad (1) \quad y_{i_1, \dots, i_{n-1}, 1, i_{n+1}, \dots, i_N} = \sum_{j=1}^{I_n} v_j x_{i_1, \dots, i_{n-1}, j, i_{n+1}, \dots, i_N}.$$

89 This operation is called tensor-times-vector multiply (TTV) and is denoted by  $\mathcal{Y} =$

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

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

102 The *Hadamard product* of two vectors  $\mathbf{u}, \mathbf{v} \in \mathbb{R}^I$  is a vector  $\mathbf{w} = \mathbf{u} * \mathbf{v}$ ,  $\mathbf{w} \in \mathbb{R}^I$ ,  
 103 where  $w_i = u_i \cdot v_i$ . The *outer product* of  $K > 1$  vectors  $\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(K)}$  of corresponding  
 104 sizes  $I_1, \dots, I_K$  is denoted by  $\mathcal{X} = \mathbf{u}^{(1)} \circ \dots \circ \mathbf{u}^{(K)}$  where  $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_K}$  is a  $K$ -  
 105 dimensional tensor with elements  $x_{i_1, \dots, i_K} = \prod_{k \in \mathbb{N}_K} u_{i_k}^{(k)}$ . The *Kronecker product* of  
 106 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

$$107 \quad \mathbf{w} = \mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} u_1 \mathbf{v} \\ u_2 \mathbf{v} \\ \vdots \\ u_I \mathbf{v} \end{bmatrix}.$$

108 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

$$109 \quad (2) \quad \mathbf{W} = \mathbf{U} \odot \mathbf{V} = [\mathbf{u}_1 \otimes \mathbf{v}_1, \dots, \mathbf{u}_K \otimes \mathbf{v}_K],$$

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

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

116 **2.2. CP decomposition.** The rank- $R$  CP-decomposition of a tensor  $\mathcal{X}$  ex-  
 117 presses or approximates  $\mathcal{X}$  as a sum of  $R$  rank-1 tensors. For instance, for  $\mathcal{X} \in$   
 118  $\mathbb{R}^{I \times J \times K}$ , we obtain  $\mathcal{X} \approx \sum_{r=1}^R \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$  where  $\mathbf{a}_r \in \mathbb{R}^I$ ,  $\mathbf{b}_r \in \mathbb{R}^J$ , and  $\mathbf{c}_r \in \mathbb{R}^K$ .  
 119 This decomposition results in the element-wise approximation (or equality)  $x_{i,j,k} \approx$   
 120  $\sum_{r=1}^R a_{ir} b_{jr} c_{kr}$ . The minimum  $R$  value rendering this approximation an equality is  
 121 called as the *rank* (or CP-rank) of the tensor  $\mathcal{X}$ , and computing this value is NP-  
 122 hard [21]. Here, the matrices  $\mathbf{A} = [\mathbf{a}_1, \dots, \mathbf{a}_R]$ ,  $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_R]$ , and  $\mathbf{C} = [\mathbf{c}_1, \dots, \mathbf{c}_R]$   
 123 are called the *factor matrices*, or *factors*. For  $N$ -mode tensors, we use  $\mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}$   
 124 to refer to the factor matrices having  $I_1, \dots, I_N$  rows and  $R$  columns, and  $\mathbf{u}_j^{(i)}$  to refer  
 125 to the  $j$ th column of  $\mathbf{U}^{(i)}$ . The standard algorithm for computing a CP decomposition  
 126 is the alternating least squares (CP-ALS) method, which establishes a good trade-off  
 127 between the number of iterations and the cost per iteration [29]. It is an iterative  
 128 algorithm, shown in Algorithm 1, that progressively updates the factors  $\mathbf{U}^{(n)}$  in an  
 129 alternating fashion starting from an initial guess. CP-ALS runs until it can no longer  
 130 improve the solution, or it reaches the allowed maximum number of iterations. The  
 131 initial factor matrices can be randomly set, or computed using the truncated SVD of

132 the matricizations of  $\mathcal{X}$  [29]. Each iteration of CP-ALS consists of  $N$  *subiterations*,  
 133 where in the  $n$ th subiteration  $\mathbf{U}^{(n)}$  is updated using  $\mathcal{X}$  and the current values of all  
 134 other factor matrices.

---

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

---

**Input:**  $\mathcal{X}$ : An  $N$ -mode tensor,  $\mathcal{X} \in \mathbb{R}^{I_1, \dots, I_N}$

$R$ : The rank of CP decomposition

$\mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}$ : Initial factor matrices

**Output:**  $[\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}]$ : A rank- $R$  CP decomposition of  $\mathcal{X}$

1: **repeat**

2:   **for**  $n = 1, \dots, N$  **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:      $\boldsymbol{\lambda} \leftarrow \text{COLUMN-NORMALIZE}(\mathbf{U}^{(n)})$

7: **until** convergence is achieved or the maximum number of iterations is reached.

8: **return**  $[\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}]$

---

135       Computing the matrix  $\mathbf{M}^{(n)} \in \mathbb{R}^{I_n \times R}$  at **Line 3** of **Algorithm 1** is the sole part  
 136 involving the tensor  $\mathcal{X}$ , and it is the most expensive computational step, for both  
 137 sparse and dense tensors. The operation  $\mathbf{X}_{(n)} \odot_{i \neq n} \mathbf{U}^{(i)}$  is called *matricized tensor-*  
 138 *times Khatri-Rao product* (MTTKRP). The Khatri-Rao product of the involved  $\mathbf{U}^{(n)}$ s  
 139 defines a matrix of size  $(\prod_{i \neq n} I_i) \times R$  according to (2), and can get very costly in  
 140 terms of computational and memory requirements when  $I_i$  or  $N$  is large—which is  
 141 the case for many real-world sparse tensors. To alleviate this, various methods are  
 142 proposed in the literature that enable performing MTTKRP without forming Khatri-  
 143 Rao product. One such formulation [4], also used in Tensor Toolbox [5], expresses  
 144 MTTKRP in terms of a series of TTVs, and computes the resulting matrix  $\mathbf{M}^{(n)}$   
 145 column by column. With this formulation, the  $r$ th column of  $\mathbf{M}^{(n)}$  can be computed  
 146 using  $N - 1$  TTVs as in  $\mathbf{M}^{(n)}(:, r) \leftarrow \mathcal{X} \times_{i \neq n} \mathbf{u}_r^{(i)}$ , or equivalently,

$$147 \quad (3) \quad \mathbf{M}^{(n)}(:, r) \leftarrow \mathcal{X} \times_1 \mathbf{u}_r^{(1)} \times_2 \cdots \times_{n-1} \mathbf{u}_r^{(n-1)} \times_{n+1} \mathbf{u}_r^{(n+1)} \times_{n+2} \cdots \times_N \mathbf{u}_r^{(N)}.$$

148       Once  $\mathbf{M}^{(n)}$  is obtained, the Hadamard product of matrices  $\mathbf{U}^{(i)T} \mathbf{U}^{(i)}$  of size  
 149  $R \times R$  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  
 150 that within the  $n$ th subiteration, only  $\mathbf{U}^{(n)}$  is updated among all factor matrices.  
 151 Therefore, for efficiency, one can precompute all matrices  $\mathbf{U}^{(i)T} \mathbf{U}^{(i)}$  of size  $R \times R$   
 152 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  
 153  $R$  of approximation is chosen as a small constant in practice for sparse tensors (less  
 154 than 50) [47], performing these Hadamard products to compute  $\mathbf{H}^{(n)}$  and the matrix-  
 155 matrix multiplication to compute  $\mathbf{U}^{(n)T} \mathbf{U}^{(n)}$  become relatively cheap compared with  
 156 the TTV step. Once both  $\mathbf{M}^{(n)}$  and  $\mathbf{H}^{(n)}$  are computed, another matrix-matrix  
 157 multiplication is performed using  $\mathbf{M}^{(n)}$  and the pseudoinverse of  $\mathbf{H}^{(n)}$  in order to  
 158 update the matrix  $\mathbf{U}^{(n)}$ , which is not expensive when  $R$  is small. Finally,  $\mathbf{U}^{(n)}$  is  
 159 normalized column-wise, and the column vector norms are stored in a vector  $\boldsymbol{\lambda} \in \mathbb{R}^R$ .

160       The convergence is achieved when the relative reduction in the norm of the error,  
 161 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  
 162 insignificant.

163 **2.3. Hypergraphs and hypergraph partitioning.** We partition the data and  
 164 the computation using the standard hypergraph partitioning tools. Necessary defini-  
 165 tions follow.

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

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

$$174 \quad (4) \quad W_k \leq W_{avg}(1 + \varepsilon) \quad \text{for } k \in \mathbb{N}_K,$$

175 we say that  $\Pi$  is *balanced* where  $\varepsilon$  represents the allowed maximum imbalance ratio.

176 In a partition  $\Pi$ , a hyperedge that has at least one vertex in a part is said to  
 177 *connect* that part. The number of parts connected by a hyperedge  $h$  is called its  
 178 *connectivity*, and is denoted by  $\kappa_h$ . Given a vertex partition  $\Pi$  of a hypergraph  
 179  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ , one can measure the *cutsizes* metric induced by  $\Pi$  as

$$180 \quad (5) \quad \chi(\Pi) = \sum_{h \in \mathcal{E}} c[h](\kappa_h - 1).$$

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

182 Given  $\varepsilon \geq 0$  and an integer  $K > 1$ , the standard hypergraph partitioning problem  
 183 is defined as the task of finding a balanced partition  $\Pi$  with  $K$  parts such that  $\chi(\Pi)$   
 184 is minimized. Hypergraph partitioning problem is NP-hard [31].

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

$$190 \quad (6) \quad W_{k,i} \leq W_{avg,i}(1 + \varepsilon) \quad \text{for } k \in \mathbb{N}_K \text{ and } i \in \mathbb{N}_C,$$

191 where the  $i$ th weight  $W_{k,i}$  of a part  $\mathcal{V}_k$  is defined as the sum of the  $i$ th weights of the  
 192 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  
 193 weight for the  $i$ th weight of all vertices, i.e.,  $W_{avg,i} = \sum_{v \in \mathcal{V}} w[v, i]/K$ .

194 **2.4. Dimension tree.** A *dimension tree* is a data structure that partitions the  
 195 mode indices of an  $N$ -dimensional tensor in a hierarchical manner for computing  
 196 tensor decompositions efficiently. It was first used in the hierarchical Tucker format  
 197 representing the hierarchical Tucker decomposition of a tensor [18], which was intro-  
 198 duced as a computationally feasible alternative to the original Tucker decomposition  
 199 for higher order tensors. We provide the formal definition of a dimension tree along  
 200 with some basic properties as follows.

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

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

- 206        2. For each non-leaf node  $t \in \mathcal{T}$ , the mode sets of its children partition  $\mu(t)$ .  
 207        3. The  $n$ th leaf node, denoted by  $l_n \in \text{LEAVES}(\mathcal{T})$ , has  $\mu(l_n) = \{n\}$ .

208        For the simplicity of the discussion, we assume without loss of generality that the  
 209        sequence  $l_1, \dots, l_N$  corresponds to a relative ordering of the leaf nodes in a post-order  
 210        traversal of the dimension tree. If this is not the case, we can relabel tensor modes  
 211        accordingly. We define the *inverse mode set* of a node  $t$  as  $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$ . For each  
 212        node  $t$  with a parent  $\mathcal{P}(t)$ ,  $\mu(t) \subset \mu(\mathcal{P}(t))$  holds due to the second property, which in  
 213        turn yields  $\mu'(t) \supset \mu'(\mathcal{P}(t))$ . If a dimension tree has the height  $\lceil \log(N) \rceil$  with its first  
 214         $\lceil \log(N) \rceil$  levels forming a complete binary tree, we call it a balanced binary dimension  
 215        tree (BDT). In Figure 1, we show a BDT for 4 dimensions (associated with a sparse  
 216        tensor described later).

217        **3. Computing CP decomposition using dimension trees.** In this section,  
 218        we propose a novel way of using dimension trees for computing the *standard* CP  
 219        decomposition of tensors with a formulation that asymptotically reduces the compu-  
 220        tational cost. In doing so, we do not alter the original CP decomposition in any way.  
 221        The reduction in the computational cost is made possible by storing partial TTV re-  
 222        sults, and hence by trading off more memory. A similar idea of reusing partial results  
 223        without the use of a tree framework was moderately explored by Baskaran et al. [6]  
 224        for computing the Tucker decomposition of sparse tensors, and by Phan et al. for  
 225        computing the CP decomposition of dense tensors [35]. We generalized the approach  
 226        of Baskaran et al. [6] using dimension trees for better computational gains [25] in the  
 227        standard algorithm for sparse Tucker decomposition. Here, we adopt the same data  
 228        structure for reducing the cost of MTTKRP operations.

229        **3.1. Using dimension trees to perform successive tensor-times-vector**  
 230        **multiplies.** At each subiteration of the CP-ALS algorithm,  $\mathcal{X}$  is multiplied with  
 231        the column vectors of matrices in  $N - 1$  modes using (3) in performing MTTKRP.  
 232        Some of these TTVs involve the same matrices as the preceding subiterations. As a  
 233        series of TTVs can be done in any order, this opens up the possibility to factor out  
 234        and reuse TTV results that are common in consecutive subiterations for reducing the  
 235        computational cost. For instance, in the first subiteration of CP-ALS using a 4-mode  
 236        tensor  $\mathcal{X}$ , we compute  $\mathcal{X} \times_2 \mathbf{u}_r^{(2)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$  and eventually update  $\mathbf{u}_r^{(1)}$  for each  
 237         $r \in \mathbb{N}_R$ , whereas in the second subiteration we compute  $\mathcal{X} \times_1 \mathbf{u}_r^{(1)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$   
 238        and update  $\mathbf{u}_r^{(2)}$ . In these two subiterations, the matrices  $\mathbf{U}^{(3)}$  and  $\mathbf{U}^{(4)}$  remain  
 239        unchanged, and both TTV steps involve the TTV of  $\mathcal{X}$  with  $\mathbf{u}_r^{(3)}$  and  $\mathbf{u}_r^{(4)}$ . Hence,  
 240        we can compute  $\mathcal{Y}_r = \mathcal{X} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ , then reuse it in the first and the second  
 241        subiterations as  $\mathcal{Y}_r \times_2 \mathbf{u}_r^{(2)}$  and  $\mathcal{Y}_r \times_1 \mathbf{u}_r^{(1)}$  to obtain the required TTV results.

242        We use dimension trees to systematically detect and reuse such partial results by  
 243        associating a tree  $\mathcal{T}$  with an  $N$ -dimensional tensor  $\mathcal{X}$  as follows. With each node  
 244         $t \in \mathcal{T}$ , we associate  $R$  tensors  $\mathcal{X}_1^{(t)}, \dots, \mathcal{X}_R^{(t)}$ .  $\mathcal{X}_r^{(t)}$  corresponds to the TTV result  
 245         $\mathcal{X}_r^{(t)} = \mathcal{X} \times_{d \in \mu'(t)} \mathbf{u}_r^{(d)}$ . Therefore, the inverse mode set  $\mu'(t)$  corresponds to the set  
 246        of modes in which TTV is performed on  $\mathcal{X}$  to form  $\mathcal{X}_r^{(t)}$ . For the root of the tree,  
 247         $\mu'(\text{ROOT}(\mathcal{T})) = \emptyset$ ; thus, all tensors of the root node correspond to the original tensor  
 248         $\mathcal{X}$ , i.e.,  $\mathcal{X}_r^{(\text{ROOT}(\mathcal{T}))} = \mathcal{X}$  for  $r \in \mathbb{N}_R$ . Since  $\mu'(t) \supset \mu'(\mathcal{P}(t))$  for a node  $t$  and its  
 249        parent  $\mathcal{P}(t)$ , tensors of  $\mathcal{P}(t)$  can be used as partial results to update the tensors of  $t$ .  
 250        Let  $\delta(t) = \mu'(t) \setminus \mu'(\mathcal{P}(t))$ . We can then compute each tensor of  $t$  from its parent's  
 251        as  $\mathcal{X}_r^{(t)} = \mathcal{X}_r^{(\mathcal{P}(t))} \times_{d \in \delta(t)} \mathbf{u}_r^{(d)}$ . This procedure is called DTREE-TTV and is shown  
 252        in Algorithm 2. DTREE-TTV first checks if the tensors of  $t$  are already computed,

253 and immediately returns if so. This happens, for example, when two children of  $t$  call  
 254 DTREE-TTV on  $t$  consecutively, in which case in the second DTREE-TTV call, the  
 255 tensors of  $t$  would be already computed. If the tensors of  $t$  are not already computed,  
 256 DTREE-TTV on  $\mathcal{P}(t)$  is called first to make sure that  $\mathcal{P}(t)$ 's tensors are up-to-date.  
 257 Then, each  $\mathcal{X}_r^{(t)}$  is computed by performing a TTV on the corresponding tensor  
 258  $\mathcal{X}_r^{(\mathcal{P}(t))}$  of the parent. We use the notation  $\mathcal{X}_r^{(t)}$  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  $\mathcal{X}_r^{(t)}$  of  $t$  are computed.

```

1: if EXISTS( $\mathcal{X}_r^{(t)}$ ) then
2:   return                                     ▶ Tensors  $\mathcal{X}_r^{(t)}$  are already computed.
3: DTREE-TTV( $\mathcal{P}(t)$ )                             ▶ Compute the parent's tensors  $\mathcal{X}_r^{(\mathcal{P}(t))}$  first.
4: for  $r = 1, \dots, R$  do                       ▶ Now update all tensors of  $t$  using parent's.
5:    $\mathcal{X}_r^{(t)} \leftarrow \mathcal{X}_r^{(\mathcal{P}(t))} \times_{d \in \delta(t)} \mathbf{u}_r^{(d)}$ 

```

---

259 **3.2. Dimension tree-based CP-ALS algorithm.** At the  $n$ th subiteration  
 260 of Algorithm 1, we need to compute  $\mathcal{X} \times_{i \neq n} \mathbf{u}_r^{(i)}$  for all  $r \in \mathbb{N}_R$  in order to form  
 261  $\mathbf{M}^{(n)}$ . Using a dimension tree  $\mathcal{T}$  with leaves  $l_1, \dots, l_N$ , we can perform this simply by  
 262 executing DTREE-TTV( $l_n$ ), after which the  $r$ th tensor of  $l_n$  provides the  $r$ th column of  
 263  $\mathbf{M}^{(n)}$ . Once  $\mathbf{M}^{(n)}$  is formed, the remaining steps follow as before. We show the whole  
 264 CP-ALS using a dimension tree in Algorithm 3. At Line 1, we construct a dimension  
 265 tree  $\mathcal{T}$  with the leaf order  $l_1, \dots, l_N$  obtained from a post-order traversal of  $\mathcal{T}$ . This  
 266 tree can be constructed in any way that respects the properties of a dimension tree  
 267 described in section 3; but for our purposes we assume that it is formed as a BDT.  
 268 At Line 8 within the  $n$ th subiteration, we destroy all tensors of a node  $t$  if its set  
 269 of multiplied modes  $\mu'(t)$  involve  $n$ , as in this case its tensors involve multiplication  
 270 using the old value of  $\mathbf{U}^{(n)}$  which is about to change. Note that this step destroys the  
 271 tensors of all nodes not lying on a path from  $l_n$  to the root. Afterwards, DTREE-TTV  
 272 is called at Line 9 for the leaf node  $l_n$  to compute its tensors. This step computes (or  
 273 reuses) the tensors of all nodes from the path from  $l_n$  to the root. Next, the  $r$ th  
 274 column of  $\mathbf{M}^{(n)}$  is formed using  $\mathcal{X}_r^{(l_n)}$  for  $r \in \mathbb{N}_R$ . Once  $\mathbf{M}^{(n)}$  is ready,  $\mathbf{H}^{(n)}$  and  $\mathbf{U}^{(n)}$   
 275 are computed as before, after which  $\mathbf{U}^{(n)}$  is normalized.

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

279 **THEOREM 2.** *Let  $\mathcal{X}$  be an  $N$ -mode tensor. The total number of TTVs at each*  
 280 *iteration of Algorithm 3 using a BDT is at most  $RN \lceil \log N \rceil$ .*

281 *Proof.* As we assume that the sequence  $l_1, \dots, l_N$  is obtained from a post-order  
 282 traversal of  $\mathcal{T}$ , for each internal node  $t$ , the subtree rooted at  $t$  has the leaves  
 283  $l_i, l_{i+1}, \dots, l_{i+k-1}$  corresponding to  $k$  consecutive mode indices for some positive  
 284 integers  $i$  and  $k$ . As we have  $\mu(l_i) = i$  for the  $i$ th leaf node, we obtain  $\mu(t) =$   
 285  $\{i, i+1, \dots, i+k-1\}$  due to the second property of dimension trees. As a result,  
 286 for each leaf node  $l_{i+k'}$  for  $0 \leq k' < k$ , we have  $i+k' \in \mu(t)$ ; hence  $i+k' \notin \mu'(t)$   
 287 as  $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$ . Therefore, within an iteration of Algorithm 3, the tensors of  $t$   
 288 get computed at the  $i$ th subiteration, stay valid (not destroyed) and get reused until  
 289 the  $i+k-1$ th subiteration, and finally get destroyed in the following subiteration.  
 290 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:**  $[\lambda; \mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}]$ : Rank- $R$  CP decomposition of  $\mathcal{X}$ 

```

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

```

291 the modes associated with the leaf tensors in its subtree are processed (which are the  
292 only nodes that can reuse the tensors of  $t$ ). As a result, in one CP-ALS iteration,  
293 tensors of every tree node (except the root) get to be computed and destroyed exactly  
294 once. Therefore, the total number of TTVs in one iteration becomes the sum of the  
295 number of TTVs performed to compute the tensors of each node in the tree once. In  
296 computing its tensors, every node  $t$  has  $R$  tensors, and for each tensor it performs  
297 TTVs for each dimension in the set  $\delta(t)$  in [Algorithm 2](#), except the root node, whose  
298 tensors are all equal to  $\mathcal{X}$  and never change. Therefore, we can express the total  
299 number of TTVs performed within a CP-ALS iteration due to one of these  $R$  tensors  
300 as

$$301 \quad (7) \quad \sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\delta(t)| = \sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\mu(\mathcal{P}(t)) \setminus \mu(t)|.$$

302 Since in a BDT every non-leaf node  $t$  has exactly two children, say  $t_1$  and  $t_2$ , we  
303 obtain  $|\mu(t) \setminus \mu(t_1)| + |\mu(t) \setminus \mu(t_2)| = |\mu(t)|$ , as  $\mu(t)$  is partitioned into two sets  $\mu(t_1)$   
304 and  $\mu(t_2)$ . With this observation, we can reformulate (7) as

$$305 \quad (8) \quad \sum_{t \in \mathcal{T} \setminus \{\text{ROOT}(\mathcal{T})\}} |\mu(\mathcal{P}(t)) \setminus \mu(t)| = \sum_{t \in \mathcal{T} \setminus \{\text{LEAVES}(\mathcal{T})\}} |\mu(t)|.$$

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

311 iterating over nodes by levels of the BDT yields

$$\begin{aligned}
 312 \quad \sum_{t \in \mathcal{T} \setminus \text{LEAVES}(\mathcal{T})} |\mu(t)| &= \sum_{k=1}^{\lceil \log N \rceil} \sum_{t \in \mathcal{T} \setminus \text{LEAVES}(\mathcal{T}), \text{LEVEL}(t)=k} |\mu(t)| \\
 313 \quad &\leq \sum_{k=1}^{\lceil \log N \rceil} N = N \lceil \log N \rceil . \\
 314
 \end{aligned}$$

315 As there are  $R$  tensors in each BDT tree node, the overall cost becomes  $RN \lceil \log N \rceil$   
 316 TTVs for a CP-ALS iteration.  $\square$

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

337 For sparse tensors, one key idea we use for obtaining high performance is per-  
 338 forming TTVs for all  $R$  tensors  $\mathcal{X}_r^{(t)}$  of a node  $t \in \mathcal{T}$  in a *vectorized* manner. We  
 339 illustrate this on a 4-dimensional tensor  $\mathcal{X}$  and a BDT, and for clarity, we put the  
 340 mode set  $\mu(t)$  of each tree node  $t$  in the subscript, as in  $t_{1234}$ . Let  $t_{1234}$  represent  
 341 the root of the BDT with  $\mathcal{X}_r^{(t_{1234})} = \mathcal{X}$  for all  $r \in \mathbb{N}_R$ . The two children of  $t_{1234}$   
 342 are  $t_{12}$  and  $t_{34}$  with the corresponding tensors  $\mathcal{X}_r^{(t_{12})} = \mathcal{X}_r^{(t_{1234})} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$  and  
 343  $\mathcal{X}_r^{(t_{34})} = \mathcal{X}_r^{(t_{1234})} \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$ , respectively. Since  $\mathcal{X}_r^{(t_{1234})}$  are identical for all  
 344  $r \in \mathbb{N}_R$ , the nonzero pattern of tensors  $\mathcal{X}_r^{(t_{12})}$  are also identical. This is also the case  
 345 for  $\mathcal{X}_r^{(t_{34})}$ , and the same argument applies to the children  $t_1$  and  $t_2$  of  $t_{12}$ , as well  
 346 as  $t_3$  and  $t_4$  of  $t_{34}$ . As a result, each node in the tree involves  $R$  tensors with iden-  
 347 tical nonzero patterns. This opens up two possibilities in terms of efficiency. First,  
 348 it is sufficient to compute only one set of nonzero indices for each node  $t \in \mathcal{T}$  to  
 349 represent the nonzero structure of all of its tensors, which reduces the computational  
 350 and memory cost by a factor of  $R$ . Second, we can perform the TTVs for all tensors  
 351 at once in a “vectorized” manner by modifying (1) to perform  $R$  TTVs of the form  
 352  $\mathcal{Y}_r \leftarrow \mathcal{X}_r \times_d \mathbf{v}_r$  for  $\mathbf{V} = [\mathbf{v}_1 | \cdots | \mathbf{v}_R] \in \mathbb{R}^{I_d \times R}$  as

$$353 \quad (9) \quad \mathbf{y}_{i_1, \dots, i_{d-1}, 1, i_{d+1}, \dots, i_N}^{(\cdot)} = \sum_{j=1}^{I_d} \mathbf{V}(j, \cdot) * \mathbf{x}_{i_1, \dots, i_{d-1}, j, i_{d+1}, \dots, i_N}^{(\cdot)},$$

354 where  $\mathbf{y}_{i_1, \dots, i_{d-1}, 1, i_{d+1}, \dots, i_N}^{(\cdot)}$  and  $\mathbf{x}_{i_1, \dots, i_{d-1}, j, i_{d+1}, \dots, i_N}^{(\cdot)}$  are vectors of size  $R$  with ele-  
 355 ments  $y_{i_1, \dots, i_{d-1}, 1, i_{d+1}, \dots, i_N}^{(r)}$  and  $x_{i_1, \dots, i_{d-1}, j, i_{d+1}, \dots, i_N}^{(r)}$  in  $\mathcal{Y}_r$  and  $\mathcal{X}_r$ , for all  $r \in \mathbb{N}_R$ . We  
 356 call this operation tensor-times-multiple-vector multiplication (TTMV) as  $R$  column  
 357 vectors of  $\mathbf{V}$  are multiplied simultaneously with  $R$  tensors of identical nonzero pat-  
 358 terns. We can similarly extend this formula to the multiplication  $\mathcal{Z}_r \leftarrow \mathcal{Y}_r \times_e \mathbf{w}_r =$   
 359  $(\mathcal{X}_r \times_d \mathbf{v}_r) \times_e \mathbf{w}_r$  in two modes  $d$  and  $e$ ,  $d < e$ , with matrices  $\mathbf{V} \in \mathbb{R}^{I_d \times R}$  and  
 360  $\mathbf{W} \in \mathbb{R}^{I_e \times R}$  as

$$361 \quad \mathbf{z}_{i_1, \dots, i_{d-1}, 1, i_{d+1}, \dots, i_{e-1}, 1, i_{e+1}, \dots, i_N}^{(\cdot)} = \sum_{j_2=1}^{I_e} \mathbf{W}(j_2, \cdot) * \mathbf{y}_{i_1, \dots, i_{d-1}, 1, i_{d+1}, \dots, i_{e-1}, j_2, i_{e+1}, \dots, i_N}^{(\cdot)}$$

$$362 \quad (10) \quad = \sum_{(j_1, j_2)=(1,1)}^{(I_d, I_e)} \mathbf{V}(j_1, \cdot) * \mathbf{W}(j_2, \cdot) * \mathbf{x}_{i_1, \dots, i_{d-1}, j_1, i_{d+1}, \dots, i_{e-1}, j_2, i_{e+1}, \dots, i_N}^{(\cdot)}$$

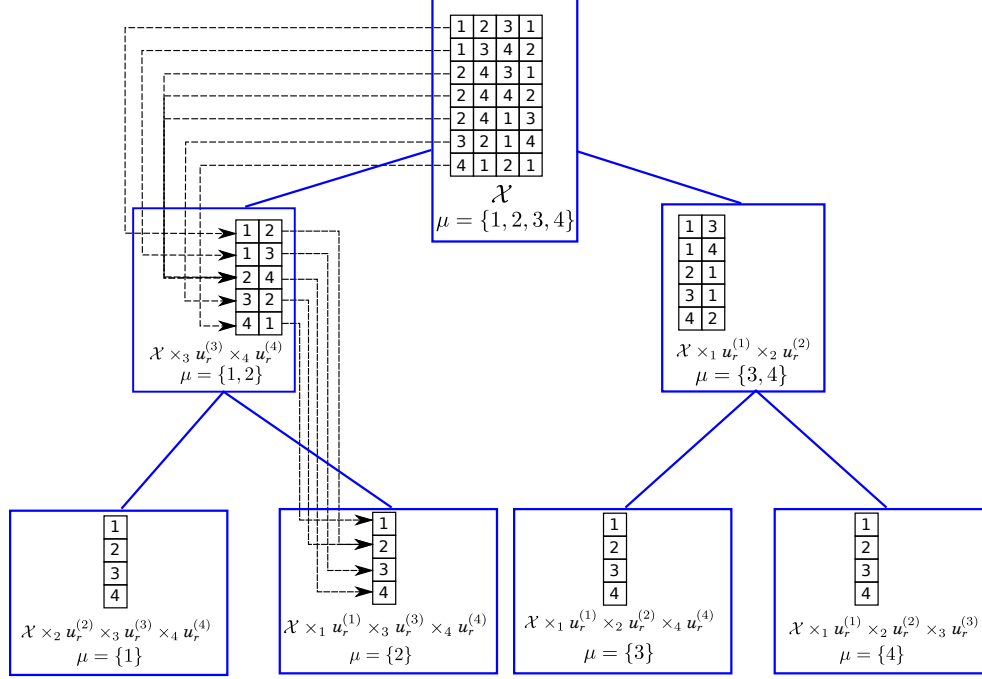
363 The formula similarly generalizes to any number of dimensions, where each dimension  
 364 adds another Hadamard product with a corresponding matrix row. This “thick” mode  
 365 of operation provides a significant performance gain thanks to the increase in locality.  
 366 Also, performing TTVs in this manner in CP-ALS effectively reduces the  $RN[\log N]$   
 367 TTVs required in [Theorem 2](#) to  $N[\log N]$  TTMV calls within an iteration. In our  
 368 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  
 369 index tuple of the form  $\mathcal{I}_t(k) = (i_1, \dots, i_N)$ , to represent the nonzeros  $x_{i_1, \dots, i_N}^{(r)} \in \mathcal{X}_r^{(t)}$   
 370 for all  $r \in \mathbb{N}_R$ . Also, for each such index tuple we hold a vector of size  $R$  corresponding  
 371 to the values of this nonzero in each one of  $R$  tensors, and we denote this value vector  
 372 as  $\mathbf{V}_t(k, \cdot)$ , where  $\mathbf{V}_t \in \mathbb{R}^{|\mathcal{I}_t| \times R}$  is called the *value matrix* of  $t$ . Finally, carrying  
 373 out the summation (9) requires a “mapping”  $\mathcal{R}_t(k)$  indicating the set of elements of  
 374  $\mathcal{X}^{(P(t))}$  contributing to this particular element of  $\mathcal{X}^{(t)}$ . Altogether, we represent the  
 375 sparse tensors  $\mathcal{X}^{(t)}$  of each tree node  $t$  with the tuple  $(\mathcal{I}_t, \mathcal{R}_t, \mathbf{V}_t)$  whose computational  
 376 details follow next.

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

384 **3.2.1. Symbolic TTV.** For simplicity, we proceed with describing how to per-  
 385 form the symbolic TTV using the same 4-dimensional tensor  $\mathcal{X}$  and the BDT. The  
 386 approach naturally generalizes to any  $N$ -dimensional tensor and dimension tree.

387 The first information we need is the list of nonzero indices  $\mathcal{I}_t$  for each node  $t$  in  
 388 a dimension tree, which we determine as follows. As mentioned, the two children  $t_{12}$   
 389 and  $t_{34}$  of  $t_{1234}$  have the corresponding tensors  $\mathcal{X}_r^{(t_{12})} = \mathcal{X}_r^{(t_{1234})} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$  and  
 390  $\mathcal{X}_r^{(t_{34})} = \mathcal{X}_r^{(t_{1234})} \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$ , respectively. Using (10), the nonzero indices of  $\mathcal{X}_r^{(t_{12})}$   
 391 and  $\mathcal{X}_r^{(t_{34})}$  take the form  $(i, j, 1, 1)$  and  $(1, 1, k, l)$ , respectively (tensor indices in the  
 392 multiplied dimensions are always 1; hence we omit storing them in practice), and such  
 393 a nonzero index exists in these tensors only if there exists a nonzero  $x_{i,j,k,l} \in \mathcal{X}_r^{(t_{1234})}$ .  
 394 Determining the list  $\mathcal{I}_{t_{12}}$  (or  $\mathcal{I}_{t_{34}}$ ) can simply be done by starting with a list  $\mathcal{I}_{t_{1234}}$   
 395 of tuples, then replacing each tuple  $(i, j, k, l)$  in the list with the tuple  $(i, j)$  (or with  
 396  $(k, l)$  for  $\mathcal{I}_{t_{34}}$ ). This list may contain duplicates, which can be efficiently eliminated

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



397 by sorting. Once the list of nonzeros for  $t_{12}$  (or  $t_{34}$ ) is determined, we proceed to  
 398 detecting the nonzero patterns of its children  $t_1$  and  $t_2$  (or,  $t_3$  and  $t_4$ ).

399 To be able to carry out the numerical calculations (9) and (10) for each nonzero  
 400 index at a node  $t$ , we need to identify the set of nonzeros of the parent node  $\mathcal{P}(t)$ 's  
 401 tensors that contribute to this nonzero. Specifically, at  $t_{12}$ , for each nonzero index  
 402  $\mathcal{I}_{t_{12}}(m) = (i, j)$  we need to bookmark all nonzero index tuples of  $t_{1234}$  of the form  
 403  $\mathcal{I}_{t_{1234}}(n) = (i, j, k, l)$ , as this is this set of nonzeros of  $\mathcal{X}^{(t_{1234})}$  that contribute to the  
 404 nonzero of  $\mathcal{X}^{(t_{12})}$  with index  $(i, j, 1, 1)$  in (10). Therefore, for each such index tuple of  
 405  $t_{12}$  we need a *reduction set*  $\mathcal{R}_{t_{12}}(m)$  which contains all such index tuples of the parent,  
 406 i.e.,  $n \in \mathcal{R}_{t_{12}}(m)$ . We determine these sets simultaneously with  $\mathcal{I}_t$ . In Figure 1, we  
 407 illustrate a sample BDT for a 4-dimensional sparse tensor with  $\mathcal{I}_t$ , shown with arrays,  
 408 and  $\mathcal{R}_t$ , shown using arrows.

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

411 **THEOREM 3.** *Let  $\mathcal{X}$  be an  $N$ -mode sparse tensor. The total number of index*  
 412 *arrays in a BDT of  $\mathcal{X}$  is at most  $N(\lceil \log N \rceil + 1)$ .*

413 *Proof.* Each node  $t$  in the dimension tree holds an index array for each mode in  
 414 its mode set  $\mu(t)$ . As stated in the proof of Theorem 2, the total size of mode sets  
 415 at each tree level is at most  $N$ . Therefore, the total number of index arrays cannot  
 416 exceed  $(\lceil \log N \rceil + 1)N$  in a BDT.  $\square$

417 **Theorem 3** shows that the storage requirement for the tensor indices of a BDT  
 418 cannot exceed  $(\lceil \log N \rceil + 1)$ -times the size of the original tensor in the coordinate

419 format (which has  $N$  index arrays of size  $\text{nnz}(\mathcal{X})$ ), yielding the overall worst-case  
 420 memory cost  $\text{nnz}(\mathcal{X})N(\lceil \log N \rceil + 1)$ . Assuming that the tensor does not contain any  
 421 empty slices (which can otherwise be removed in a preprocessing step), the index  
 422 array of a leaf node (whose tensors are vectors) corresponding to mode  $n$  is simply  
 423  $[1, \dots, I_n]$  hence need not be explicitly stored, which effectively reduces this cost to  
 424  $\text{nnz}(\mathcal{X})N\lceil \log N \rceil$ . This cost is indeed a pessimistic estimate for real-world tensors, as  
 425 with significant index overlap in non-root tree tensors, the total memory cost could  
 426 reduce to as low as  $O(\text{nnz}(\mathcal{X})N)$ . This renders the approach very suitable for higher  
 427 dimensional tensors. Another minor cost is the storage of reduction pointers, which  
 428 necessitates one array per non-root tree node, taking  $2N - 2$  arrays in total. The size  
 429 of these arrays similarly diminishes towards the leaves with a potential index overlap.

430 In computing the symbolic TTV, we sort  $|\mu(t)|$  index arrays for each node  $t \in \mathcal{T}$ .  
 431 In addition, for each non-root node  $t$  of a BDT, we sort an extra array to determine  
 432 the reduction set  $\mathcal{R}_t$ . In the worst case, each array can have up to  $\text{nnz}(\mathcal{X})$  elements.  
 433 Therefore, combining with the number of index arrays as given in [Theorem 3](#), the over-  
 434 all worst case cost of sorting becomes  $O((N\lceil \log N \rceil + 2N - 2)\text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X})) =$   
 435  $O(N \log N \text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X})))$ . We note, however, that both the total index array  
 436 size and sorting cost are pessimistic overestimates, since the nonzero structure of real-  
 437 world tensors exhibits significant locality in indices. For example, on two tensors from  
 438 our experiments (Delicious and Flickr), we observed a reduction factor of 2.57 and 5.5  
 439 in the number of nonzeros of the children of the root of the BDT. Consequently, the  
 440 number of nonzeros in a node's tensors reduces dramatically as we approach towards  
 441 the leaves. In comparison, existing approaches [\[39\]](#) sort the original tensor once with  
 442 a cost of  $O(N\text{nnz}(\mathcal{X}) \log(\text{nnz}(\mathcal{X})))$  at the expense of computing TTVs from scratch  
 443 in each CP-ALS iteration.

444 Symbolic TTV is a one-time computation whose cost is amortized. Normally,  
 445 choosing an appropriate rank  $R$  for a sparse tensor  $\mathcal{X}$  requires several executions of  
 446 CP-ALS. Also, CP-ALS is known to be sensitive to the initialization of factor matrices;  
 447 therefore, it is often executed with multiple initializations [\[29\]](#). In all of these use cases,  
 448 the tensor  $\mathcal{X}$  is fixed; therefore, the symbolic TTV is required only once. Moreover,  
 449 CP-ALS usually has a number of iterations which involve many costly numeric TTV  
 450 calls. As a result, the cost of the subsequent numeric TTV calls over many iterations  
 451 and many CP-ALS executions easily amortizes that of this symbolic preprocessing.  
 452 Nevertheless, in case of need, this step can efficiently be parallelized in multiple ways.  
 453 First, symbolic TTV is essentially a sorting of multiple index arrays; hence, one can  
 454 use parallel sorting methods. Second, the BDT structure naturally exposes a coarser  
 455 level of parallelism; once a node's symbolic TTV is computed, one can proceed with  
 456 those of its children in parallel, and process the whole tree in this way. Finally, in a  
 457 distributed memory setting where we partition the tensor to multiple processes, each  
 458 process can perform the symbolic TTV on its local tensor in parallel. We benefit only  
 459 from this parallelism in our implementation.

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

465 **THEOREM 4.** *For an  $N$ -mode tensor  $\mathcal{X}$ , the total number of tree nodes whose*  
 466 *value matrices are allocated is at most  $\lceil \log N \rceil$  at any instant of [Algorithm 3](#) using a*  
 467 *BDT.*

468 *Proof.* Note that at the beginning of the  $n$ th subiteration of [Algorithm 3](#), the  
 469 tensors of each node  $t \in \mathcal{T}$  involving  $n$  in  $\mu'(t)$  are destroyed at [Line 8](#). These are  
 470 exactly the tensors that do not lie in the path from the leaf  $l_n$  to the root, as they  
 471 do not involve  $n$  in their mode set  $\mu$ . The TTMV result for  $l_n$  depends only on the  
 472 nodes on this path from  $l_n$  to the root; therefore, at the end of the  $n$ th subiteration,  
 473 only the tensors of the nodes on this path will be computed using [Algorithm 2](#). As  
 474 this path length cannot exceed  $\lceil \log N \rceil$  in a BDT, the number of nodes whose value  
 475 matrices are not destroyed cannot exceed  $\lceil \log N \rceil$  at any instant of [Algorithm 3](#).  $\square$

476 [Theorem 4](#) puts an upper bound of  $\lceil \log N \rceil$  on the maximum number of allo-  
 477 cated value matrices, thus on the maximum memory utilization due to tensor values,  
 478 in DTREE-CP-ALS. In the worst case, each value matrix may have up to  $\text{nnz}(\mathcal{X})$   
 479 elements, requiring  $\text{nnz}(\mathcal{X})R\lceil \log N \rceil$  memory in total to store the values. Combin-  
 480 ing with [Theorems 2](#) and [3](#), in the worst case, the intermediate results increase the  
 481 memory requirement by a factor of  $O(\log N(1 + R/N))$  (with respect to the storage  
 482 of  $\mathcal{X}$  in coordinate format), while the computational cost is reduced by a factor of  
 483  $O(N/\log N)$ . These are the largest increase in the memory and the smallest decrease  
 484 in the computational cost. In practice, we expect less increase in the memory and  
 485 more gains in the computational cost thanks to the overlap of indices towards the  
 486 leaves.

487 **4. Parallel CP-ALS for sparse tensors using dimension trees.** We first  
 488 present shared memory parallel algorithms involving efficient parallelization of the  
 489 dimension tree-based TTMVs in [subsection 4.1](#). Later, in [subsection 4.2](#), we present  
 490 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( $\mathbf{V}_t$ ) then
2:   return ▶ Numerical values  $\mathbf{V}_t$  are already computed.
3: SMP-DTREE-TTV( $\mathcal{P}(t)$ ) ▶ Compute the parent's values  $\mathbf{V}_{\mathcal{P}(t)}$  first.
4: parallel for  $1 \leq i \leq |\mathcal{I}_t|$  do ▶ Process each  $\mathcal{I}_t(i) = (i_1, \dots, i_N)$  in parallel.
5:    $\mathbf{V}_t(i, :) \leftarrow 0$  ▶ Initialize with a zero vector of size  $1 \times R$ .
6:   for all  $j \in \mathcal{R}_t(i)$  do ▶ Reduce from the element with index  $\mathcal{I}_{\mathcal{P}(t)}(j) = (j_1, \dots, j_N)$ .
7:      $\mathbf{r} \leftarrow \mathbf{V}_{\mathcal{P}(t)}(j, :)$ 
8:     for all  $d \in \delta(t)$  do ▶ Multiply the vector  $\mathbf{r}$  with corresponding matrix rows.
9:        $\mathbf{r} \leftarrow \mathbf{r} * \mathbf{U}^{(d)}(j_d, :)$ 
10:     $\mathbf{V}_t(i, :) \leftarrow \mathbf{V}_t(i, :) + \mathbf{r}$  ▶ Add the update due to the parent's  $j$ th element.

```

---

491 **4.1. Shared memory parallelism.** For the given tensor  $\mathcal{X}$ , after forming the  
 492 dimension tree  $\mathcal{T}$  with symbolic structures  $\mathcal{I}_t$  and  $\mathcal{R}_t$  for all tree nodes, we can  
 493 perform numeric TTMV computations in parallel. In [Algorithm 4](#), we provide the  
 494 shared memory parallel TTMV algorithm, called SMP-DTREE-TTV, for a node  $t$  of a  
 495 dimension tree. The goal of SMP-DTREE-TTV is to compute the tensor values  $\mathbf{V}_t$  for  
 496 a given node  $t$ . Similar to [Algorithm 2](#), it starts by checking if  $\mathbf{V}_t$  is already computed,  
 497 and returns immediately in that case. Otherwise, it calls SMP-DTREE-TTV on the  
 498 parent node  $\mathcal{P}(t)$  to make sure that parent's tensor values  $\mathbf{V}_{\mathcal{P}(t)}$  are available. Once  
 499  $\mathbf{V}_{\mathcal{P}(t)}$  is ready, the algorithm proceeds with computing  $\mathbf{V}_t$  for each nonzero index  
 500  $\mathcal{I}_t(i) = (i_1, \dots, i_N)$ . As for each such index the reduction set is defined during the  
 501 symbolic TTV,  $\mathbf{V}_t(i, :)$  can be independently updated in parallel. In performing this

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

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

511 **4.2. Distributed memory parallelism.** Parallelizing CP-ALS in a distributed  
 512 memory setting involves defining unit parallel tasks, data elements, and their interde-  
 513 pendencies. Following to this definition, we partition and distribute tensor elements  
 514 and factor matrices to all available processes. We discuss a *fine-grain* and a *medium-*  
 515 *grain* parallel task model together with the associated distributed memory parallel  
 516 algorithms.

517 We start the discussion with the following straightforward lemma that enables us  
 518 to distribute tensor nonzeros for parallelization.

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

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

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

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

529 **COROLLARY 6.** *Let  $\mathcal{X}$  and  $\mathcal{X}_1, \dots, \mathcal{X}_P$  be tensors in  $\mathbb{R}^{I_1 \times \dots \times I_N}$  with  $\sum_{i=1}^P \mathcal{X}_i =$*   
 530  $\mathcal{X}$ . *Then, for any  $n \in \mathbb{N}_N$  and  $\mathbf{u}^{(i)} \in \mathbb{R}^{I_i}$  for  $i \in \mathbb{N}_N \setminus \{n\}$  we obtain  $\mathcal{X} \times_{i \neq n} =$*   
 531  $\mathcal{X}_1 \times_{i \neq n} \mathbf{u}^{(i)} + \dots + \mathcal{X}_P \times_{i \neq n} \mathbf{u}^{(i)}$ .

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

536 **4.2.1. Fine-grain parallelism.** **Corollary 6** allows us to partition the tensor  
 537  $\mathcal{X}$  in the sum form  $\mathcal{X}_1 + \dots + \mathcal{X}_P$  for any  $P > 1$ , then perform TTMVs on each  
 538 tensor part  $\mathcal{X}_p$  independently, finally sum up these results to obtain the TTMV result  
 539 for  $\mathcal{X}$  multiplied in  $N - 1$  modes. As  $\mathcal{X}$  is sparse, an intuitive way to achieve this  
 540 decomposition is by partitioning its nonzeros to  $P$  tensors where  $P$  is the number of  
 541 available distributed processes. This way, for any dimension  $n$ , we can perform the  
 542 TTMV of  $\mathcal{X}_p$  with the columns of the set of factor matrices  $\{\mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}\}$  in all  
 543 modes except  $n$ . This yields a “local” matrix  $\mathbf{M}_p^{(n)}$  at each process  $p$ , and all these local  
 544 matrices must subsequently be “assembled” by summing up their rows corresponding

545 to the same row indices. In order to perform this assembly of rows, we also partition  
 546 the rows of matrices  $\mathbf{M}^{(n)}$  so that each row is “owned” by a process that is responsible  
 547 for holding the final row sum. We represent this partition with a vector  $\boldsymbol{\sigma}^{(n)} \in \mathbb{R}^{I_n}$   
 548 where  $\sigma_i^{(n)} = p$  implies that the final value of  $\mathbf{M}^{(n)}(i, :)$  resides at the process  $p$ . We  
 549 assume the same partition given by  $\boldsymbol{\sigma}^{(n)}$  on the corresponding factor matrices  $\mathbf{U}^{(n)}$ ,  
 550 as this enables each process to compute the rows of  $\mathbf{U}^{(n)}$  that it owns using the rows  
 551 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:**  $\mathcal{X}_p$ : A part of an  $N$ -mode tensor  $\mathcal{X}$

$R$ : 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  $\mathcal{X}_p$  in mode  $n$

$\mathbf{U}^{(n)}(\mathcal{G}_p^{(n)}, :)$ : Distributed initial matrix (rows needed by process  $p$ )

**Output:**  $[\boldsymbol{\lambda}; \mathbf{U}^{(1)}, \dots, \mathbf{U}^{(N)}]$ : Rank- $R$  CP decomposition of  $\mathcal{X}$  with distributed  $\mathbf{U}^{(n)}$

```

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

```

---

552 This approach amounts to a fine-grain parallelism where each fine-grain compu-  
 553 tational task corresponds to performing TTMV operations due to a nonzero element  
 554  $x_{i_1, \dots, i_N} \in \mathcal{X}$ . Specifically, according to (10), the process  $p$  needs the matrix rows  
 555  $\mathbf{U}^{(1)}(i_1, :), \dots, \mathbf{U}^{(N)}(i_N, :)$  for each nonzero  $x_{i_1, \dots, i_N}$  in its local tensor  $\mathcal{X}_p$  in order  
 556 to perform its local TTMVs. For each dimension  $n$ , we represent the union of all  
 557 these “required” row indices for the process  $p$  by  $\mathcal{G}_p^{(n)}$ . Similarly, we represent the  
 558 set of “owned” rows by the process  $p$  by  $\mathcal{F}_p^{(n)}$ . In this situation, the set  $\mathcal{G}_p^{(n)} \setminus \mathcal{F}_p^{(n)}$   
 559 correspond to the rows of  $\mathbf{M}^{(n)}$  for which the process  $p$  generates a partial TTMV  
 560 result, which need to be sent to their owner processes. Equally, it represents the set of  
 561 rows of  $\mathbf{U}^{(n)}$  that are not owned by the process  $p$  and are needed in its local TTMVs  
 562 according to (10). These rows of  $\mathbf{U}^{(n)}$  are similarly to be received from their owners  
 563 in order to carry out the TTMVs at process  $p$ . Hence, a “good” partition in general  
 564 involves a significant overlap of  $\mathcal{G}_p^{(n)}$  and  $\mathcal{F}_p^{(n)}$  to minimize the cost of communication.

565 In Algorithm 5, we describe the fine-grain parallel algorithm that operates in



566 this manner at process  $p$ . The elements  $\mathcal{X}_p$ ,  $\mathcal{F}_p^{(n)}$ , and  $\mathcal{G}_p^{(n)}$  are determined in the  
 567 partitioning phase, and are provided as input to the algorithm. Each process starts  
 568 with the subset  $\mathcal{G}_p^{(n)}$  of rows of each factor matrix  $\mathbf{U}^{(n)}$  that it needs for its local  
 569 computations. Similar to Algorithm 3, at Line 1 we start by forming the dimension  
 570 tree for the local tensor  $\mathcal{X}_p$ . We then compute the matrices  $\mathbf{W}^{(n)}$  corresponding to  
 571  $\mathbf{U}^{(n)T} \mathbf{U}^{(n)}$  using the initial factor matrices. We do this step in parallel in which each  
 572 process computes the local contribution  $[\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)}, :)]^T \mathbf{U}^{(n)}(\mathcal{F}_p^{(n)}, :)$  due to its owned  
 573 rows. Afterwards, we perform an ALL-REDUCE communication to sum up these local  
 574 results to obtain a copy of  $\mathbf{W}^{(n)}$  at each process. The cost of this communication  
 575 is typically negligible as  $\mathbf{W}^{(n)}$  is a small matrix of size  $R \times R$ . The main CP-ALS  
 576 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.

$\sigma$ : 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  $comm = \text{“fold”}$  then
2:   for all  $i \in \mathcal{G}_p \setminus \mathcal{F}_p$  do           ▶ Send non-owned rows to their owners.
3:     Send  $\mathbf{M}(i, :)$  to the process  $\sigma_i$ .
4:   for all  $i \in \mathcal{F}_p$  do                 ▶ Gather all partial results of owned rows together.
5:     Receive and sum up all partial results for  $\mathbf{M}(i, :)$ .
6: else if  $comm = \text{“expand”}$  then
7:   for all  $i \in \mathcal{F}_p$  do                 ▶ Send owned rows to all processes in need.
8:     Send  $\mathbf{M}(i, :)$  to the all processes  $p'$  with  $i \in \mathcal{F}_{p'}$ .
9:   for all  $i \in \mathcal{G}_p \setminus \mathcal{F}_p$  do       ▶ Receive rows that are needed locally.
10:    Receive  $\mathbf{M}(i, :)$  from the process  $\sigma_i$ .
```

---

577 and obtain the “local” matrix  $\mathbf{M}^{(n)}$ . Then, the partial results for the rows of  $\mathbf{M}^{(n)}$   
 578 are communicated to be assembled at their owner processes. We name this as the  
 579 *fold* communication step following the convention from the fine-grain parallel sparse  
 580 matrix computations. Afterwards, we form the matrix  $\mathbf{H}^{(n)}$  locally at each process  
 581  $p$  in order to compute the owned part  $\mathbf{U}^{(n)}(\mathcal{F}_p^{(n)}, :)$  using the recently assembled  
 582  $\mathbf{M}^{(n)}(\mathcal{F}_p^{(n)}, :)$ . Once the new distributed  $\mathbf{U}^{(n)}$  is computed, we normalize it column-  
 583 wise and obtain the vector  $\boldsymbol{\lambda}$  of norms. The computational and the communication  
 584 costs of this step are negligible. The new  $\mathbf{U}^{(n)}$  is finalized after the normalization,  
 585 and we then perform an *expand* communication step in which we send the rows of  
 586  $\mathbf{U}^{(n)}$  from the owner processes to all others in need. This is essentially the inverse  
 587 of the *fold* communication step in the sense that each process  $p$  that sends a partial  
 588 row result of  $\mathbf{M}^{(n)}(i, :)$  to another process  $q$  in the *fold* step receives the final result  
 589 for the corresponding row  $\mathbf{U}^{(n)}(i, :)$  from the process  $q$  in the *expand* communication.  
 590 Finally, we update the matrix  $\mathbf{W}^{(n)}$  using the new  $\mathbf{U}^{(n)}$  in parallel.

591 The expand and the fold communications at Lines 11 and 15 constitute the most  
 592 expensive communication steps. We outline these communications in Algorithm 6. In  
 593 the expand communication, the process  $p$  sends the partial results for the set  $\mathcal{G}_p \setminus \mathcal{F}_p$  of  
 594 rows to their owner processes, while similarly receiving all partial results for its set  $\mathcal{F}_p$   
 595

596 of owned rows and summing them up. Symmetrically, in the fold communication, the  
 597 process  $p$  sends the rows with indices  $\mathcal{F}_p$ , and receives the rows with indices  $\mathcal{G}_p \setminus \mathcal{F}_p$ .  
 598 The exact set of row indices that needs to be communicated in fold and expand steps  
 599 depends on the partitioning of  $\mathcal{X}$  and the factor matrices. As this partition does not  
 600 change once determined, the communicated rows between  $p$  and  $q$  stays the same in  
 601 CP-ALS iterations. Therefore, in our implementation we determine this row set once  
 602 outside the main CP-ALS iteration, and reuse it at each iteration. Another advantage  
 603 of determining this set of rows to be communicated in advance is that it eliminates  
 604 the need to store the vector  $\sigma^{(n)} \in \mathbb{R}^{I_n}$  which could otherwise be costly for a large  
 605 tensor dimension.

606 **4.2.2. Medium-grain parallelism.** For an  $N$ -mode tensor  $\mathcal{X} \in \mathbb{R}^{I_1 \times \dots \times I_N}$   
 607 and using  $P = \prod_{i=1}^N P_i$  processes, the medium-grain decomposition imposes a parti-  
 608 tion with  $P_1 \times \dots \times P_N$  Cartesian topology on the dimensions of  $\mathcal{X}$ . Specifically,  
 609 for each dimension  $n$ , the index set  $\mathbb{N}_{I_n}$  is partitioned into  $P_n$  sets  $\mathcal{S}_1^{(n)}, \dots, \mathcal{S}_{P_n}^{(n)}$ .  
 610 With this partition, the process with the index  $(p_1, \dots, p_N) \in P_1 \times \dots \times P_N$  gets  
 611  $\mathcal{X}(\mathcal{S}_{p_1}^{(1)}, \dots, \mathcal{S}_{p_N}^{(N)})$  as its local tensor. Each factor matrix  $\mathbf{U}^{(n)}$  is also partitioned fol-  
 612 lowing this topology where the set of rows  $\mathbf{U}^{(n)}(\mathcal{S}_j^{(n)}, :)$  is owned by the processes  
 613 with index  $(p_1, \dots, p_N)$  where  $p_n = j, j \in \mathbb{N}_{P_n}$ , even though these rows are to be  
 614 further partitioned among the processes having  $p_n = j$ . As a result, one advantage  
 615 of the medium-grain partition is that only the processes with  $p_n = j$  need to com-  
 616 municate with each other in mode  $n$ . This does not necessarily reduce the volume of  
 617 communication, but it can reduce the number of messages by a factor of  $P_n$  in the  
 618  $n$ th dimension.

619 One can design an algorithm specifically for the medium-grain decomposition [40].  
 620 However, using the fine-grain algorithm on a medium-grain partition effectively pro-  
 621 vides a medium-grain algorithm. For this reason, we do not need nor provide a sepa-  
 622 rate algorithm for the medium-grain task model, and use the fine-grain algorithm with  
 623 a proper medium-grain partition instead, which equally benefits from the topology.

624 **4.2.3. Partitioning.** The distributed memory algorithms that we described re-  
 625 quire partitioning the data and the computations, as in any distributed memory algo-  
 626 rithm. In order to reason about their computational load balance and communication  
 627 cost, we use hypergraph models. Once the models are built, different hypergraph  
 628 partitioning methods can be used to partition the data and the computations. We  
 629 discuss a few partitioning alternatives.

630 **4.2.4. Partitioning for the fine-grain parallelism.** We propose a hyper-  
 631 graph model to capture the computational load and the communication volume of  
 632 the fine-grain parallelization given in Algorithm 5. For the simplicity of the discus-  
 633 sion, we present the model for a 3rd order tensor  $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$  and factor matrices  
 634  $\mathbf{U}^{(1)} \in \mathbb{R}^{I_1 \times R}$ ,  $\mathbf{U}^{(2)} \in \mathbb{R}^{I_2 \times R}$ , and  $\mathbf{U}^{(3)} \in \mathbb{R}^{I_3 \times R}$ . For these inputs, we construct a  
 635 hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$  with the vertex set  $\mathcal{V}$  and the hyperedge set  $\mathcal{E}$ . The general-  
 636 ization of the model to higher order tensors should be clear from this construction.

637 The vertex set  $\mathcal{V} = \mathcal{V}^{(1)} \cup \mathcal{V}^{(2)} \cup \mathcal{V}^{(3)} \cup \mathcal{V}^{(\mathcal{X})}$  of the hypergraph involves four types  
 638 of vertices. The first three types correspond to the rows of the matrices  $\mathbf{U}^{(1)}$ ,  $\mathbf{U}^{(2)}$ ,  
 639 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  
 640  $j \in \mathbb{N}_{I_2}$ , and  $v_k^{(3)} \in \mathcal{V}^{(3)}$  for  $k \in \mathbb{N}_{I_3}$ . These vertices represent the ‘‘ownership’’ of the  
 641 corresponding matrix rows, and we assign unit weight to each such vertex. The fourth  
 642 type of vertices are denoted by  $v_{i,j,k}^{(\mathcal{X})}$ , which we define for each nonzero  $x_{i,j,k} \in \mathcal{X}$ .

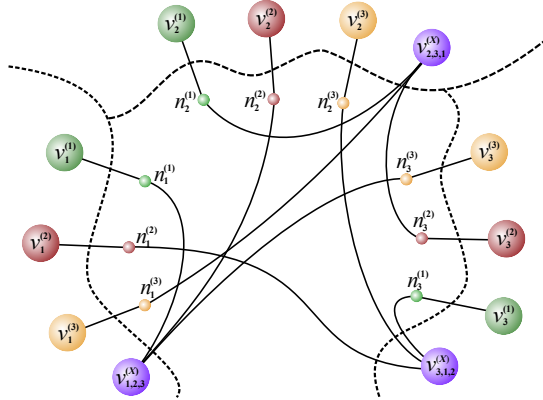


Fig. 2: Fine-grain hypergraph model for the  $3 \times 3 \times 3$  tensor  $\mathcal{X} = \{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}$  and a 3-way partition of the hypergraph. The objective is to minimize the cutsizes of the partition while maintaining a balance on the total part weights corresponding to each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

643 This vertex type relates to the number of operations performed in TTMV due to the  
 644 nonzero element  $x_{i,j,k} \in \mathcal{X}$  in using (10) in all modes. In the  $N$ -dimensional case,  
 645 this includes up to  $N$  vector Hadamard products involving the value of the nonzero  
 646  $x_{i,j,k}$ , and the corresponding matrix rows. The exact number of performed Hadamard  
 647 products depends on how nonzero indices coincide as TTVs are carried out, and cannot  
 648 be determined before a partitioning takes place. In our earlier work [26], this cost  
 649 was exactly  $N$  Hadamard products per nonzero, as the MTTKRPs were computed  
 650 without reusing partial results and without index compression after each TTMV. In  
 651 the current case, we assign a cost of  $N$  to each vertex  $v_{i,j,k}^{(\mathcal{X})}$  to represent an upper  
 652 bound on the computational cost, and expect this to lead to a good load balance in  
 653 practice. With these vertex definitions, one can use multi-constraint partitioning (6)  
 654 with one constraint per vertex type. In this case, the first, the second, and the third  
 655 types have unit weights in the first, second, and third constraints, respectively, and  
 656 zero weight in all other constraints. The fourth vertex type also gets a unit weight ( $N$ ,  
 657 or equivalently, 1) in the fourth constraint, and zero weight for others. Here, balancing  
 658 the first three constraints corresponds to balancing the number of matrix rows at each  
 659 process (which provides the memory balance as well as the computational balance in  
 660 dense matrix operations), whereas balancing the fourth type corresponds to balancing  
 661 the computational load due to TTMVs.

662 As TTMVs are carried out using (10), data dependencies to the rows of  $\mathbf{U}^{(1)}(i, :)$ ,  
 663  $\mathbf{U}^{(2)}(j, :)$ , and  $\mathbf{U}^{(3)}(k, :)$  take place when performing Hadamard products due to each  
 664 nonzero  $x_{i,j,k}$ . We introduce three types of hyperedges in  $\mathcal{E} = \mathcal{E}^{(1)} \cup \mathcal{E}^{(2)} \cup \mathcal{E}^{(3)}$   
 665 to represent these dependencies as follows:  $\mathcal{E}^{(1)}$  contains a hyperedge  $n_i^{(1)}$  for each  
 666 matrix row  $\mathbf{U}^{(1)}(i, :)$ ,  $\mathcal{E}^{(2)}$  contains a hyperedge  $n_j^{(2)}$  for each row  $\mathbf{U}^{(2)}(j, :)$ , and  $\mathcal{E}^{(3)}$   
 667 contains a hyperedge  $n_k^{(3)}$  for each row  $\mathbf{U}^{(3)}(k, :)$ . Initially,  $n_i^{(1)}$ ,  $n_j^{(2)}$  and  $n_k^{(3)}$  contain  
 668 the corresponding vertices  $v_i^{(1)}$ ,  $v_j^{(2)}$ , and  $v_k^{(3)}$ , as the owner of a matrix row has a  
 669 dependency to it by default. In computing the MTTKRP using (10), each nonzero  
 670  $x_{i,j,k}$  requires access to  $\mathbf{U}^{(1)}(i, :)$ ,  $\mathbf{U}^{(2)}(j, :)$ , and  $\mathbf{U}^{(3)}(k, :)$ . Therefore, we add the  
 671 vertex  $v_{i,j,k}^{(\mathcal{X})}$  to the hyperedges  $n_i^{(1)}$ ,  $n_j^{(2)}$  and  $n_k^{(3)}$  to model this dependency. In  
 672 Figure 2, we demonstrate this fine-grain hypergraph model on a sample tensor  $\mathcal{X} =$

673  $\{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}$ , yet we exclude the vertex weights for simplicity. Each  
 674 vertex type and hyperedge type is shown using a different color in the figure.

675 Consider now a  $P$ -way partition of the vertices of  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$  where each part is  
 676 associated with a unique process to obtain a  $P$ -way parallel execution of [Algorithm 5](#).  
 677 We consider the first subiteration of [Algorithm 5](#) that updates  $\mathbf{U}^{(1)}$ , and assume that  
 678 each process already has all data elements to carry out the local TTMVs at [Line 9](#).  
 679 Now suppose that the nonzero  $x_{i,j,k}$  is owned by the process  $p$  and the matrix row  
 680  $\mathbf{U}^{(1)}(i, \cdot)$  is owned by the process  $q$ . Then, the process  $p$  computes a partial result  
 681 for  $\mathbf{M}^{(1)}(i, \cdot)$  which needs to be sent to the process  $q$  at [Line 3](#) of [Algorithm 6](#). By  
 682 construction of the hypergraph, we have  $v_i^{(1)} \in n_i^{(1)}$  which resides at the process  
 683  $q$ , and due to the nonzero  $x_{i,j,k}$  we have  $v_{i,j,k}^{(\mathcal{X})} \in n_i^{(1)}$  which resides at the process  $p$ ;  
 684 therefore, this communication is accurately represented in the *connectivity*  $\kappa_{n_i^{(1)}}$  of the  
 685 hyperedge  $n_i^{(1)}$ . In general, the hyperedge  $n_i^{(1)}$  incurs  $\kappa_{n_i^{(1)}} - 1$  messages to transfer the  
 686 partial results for the matrix row  $\mathbf{M}^{(1)}(i, \cdot)$  to the process  $q$  at [Line 3](#). Therefore, the  
 687 *connectivity-1 cutsize* metric (5) over the hyperedges exactly encodes the total volume  
 688 of messages sent at [Line 3](#), if we set  $c[\cdot] = R$ . Since the send operations at [Line 8](#) are  
 689 duals of the send operations at [Line 3](#), the total volume of messages sent at [Line 8](#) for  
 690 the first mode is also equal to this number. By extending this reasoning to all other  
 691 modes, we obtain that the cumulative (over all modes) volume of communication  
 692 in one iteration of [Algorithm 5](#) equals to the connectivity-1 cut-size metric. As the  
 693 communication due to each mode take place in different stages, one might alternatively  
 694 use a multi-objective hypergraph model to minimize the communication volume due  
 695 to each mode (or equivalently, hyperedge type) independently.

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

708 We now discuss three methods for partitioning the described hypergraph.

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

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

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

721 label-propagation algorithm [38, 45], and includes some additional adaptations to  
 722 handle hypergraphs [9, 20]. The heuristic starts with an initial assignment of vertices  
 723 to parts, and then proceeds with multiple passes over the hypergraph. At each pass,  
 724 the vertices are visited in an order, and are possibly moved to other parts in order to  
 725 reduce the cutsize while respecting the balance constraints.

726 For the heuristic to be efficient on hypergraphs, some adaptations are needed.  
 727 Each pass involves two types of updates. In the first step, each hyperedge chooses  
 728 a “preferred part” by considering the current part of its vertices. Next, each vertex  
 729 updates its part according to the preferred parts of the hyperedges that include the  
 730 vertex. In both steps, the most dominant part index is chosen for the update. The  
 731 heuristic runs in linear time on the size of the hypergraph per iteration, and requires  
 732 a memory of  $2|\mathcal{V}| + |\mathcal{E}| + 4P$ . Running the algorithm for a few iterations provides  
 733 reasonably good partitions. This basic algorithm can have many variants. In one  
 734 variant, we visit the vertices in an order imposed by an increasing ordering by size of  
 735 the hyperedges. This variant has an overhead of sorting the hyperedges. In another  
 736 variant, we reweigh the preference of a hyperedge of size  $s$  by the multiplier  $(1 - \frac{1}{P})^{s-1}$ .  
 737 This last variant has a memory overhead for storing the weights for efficiency purposes;  
 738 for each size  $s$ , the value  $(1 - \frac{1}{P})^{s-1}$  is needed.

739 **4.2.5. Partitioning for the medium-grain parallelism.** Similar to the fine-  
 740 grain model, one can use a hypergraph model for the medium-grain parallel compu-  
 741 tations to reduce the communication volume using hypergraph partitioners. How-  
 742 ever, medium-grain variant is analogous to checkerboard partitioning designed for  
 743 matrices [13, 14], and calls for a multi-constrained partitioning. Specifically, for a  
 744 3-dimensional tensor with a process topology  $P_1 \times P_2 \times P_3$  where  $P = P_1 P_2 P_3$ , the  
 745 hypergraph is to be partitioned in three phases; using one load constraint in the first  
 746 phase,  $P_1$  constraints (where each constraint is obtained from the first phase) in the  
 747 second phase, and  $P_1 P_2$  constraints (obtained from the second phase partitioning)  
 748 in the third phase. As  $P$  can be large, the number of constraints  $P_1$  and  $P_1 P_2$  can  
 749 similarly get large, and in this case the state of the art partitioners do not perform  
 750 well both in terms of partition quality and speed. For higher dimensional tensors, this  
 751 situation only gets worse. That is why explicit communication reduction using hyper-  
 752 graph partitioning for the medium-grain algorithm is not feasible in practice. Hence,  
 753 we use the partitioning heuristic by Smith and Karypis [40] to partition medium-grain  
 754 hypergraphs for load balance, and to expect a communication reduction due to par-  
 755 tition topology indirectly. We also determine the partition topology by choosing  $P_1$ ,  
 756  $P_2$ , and  $P_3$  proportional to the tensor dimensions  $I_1$ ,  $I_2$ , and  $I_3$ .

757 **4.2.6. Mode partitioning.** Once the nonzero partitioning is obtained for the  
 758 given fine- or medium-grain parallelism, we proceed with partitioning the mode in-  
 759 dices (or, equivalently, the rows of the factor matrices) using a similar heuristic com-  
 760 mon in similar work [26, 40]. For each matrix row  $i$  in dimension  $n$ , we identify the  
 761 processes that have a data dependency to that row. These are exactly the processes  
 762 which have at least one nonzero with index  $i$  in the  $n$ th dimension. Next, all row  
 763 indices are sorted in increasing order of the number of dependent processes. Finally,  
 764 each row is greedily assigned to the process having the minimum total communication  
 765 volume among all processes dependent to that row.

766 **5. Related work.** There has been many recent advances in the efficient compu-  
 767 tations of tensor factorizations in general, and CP decomposition in particular. We  
 768 briefly mention these here and refer the reader to the original sources for details. In [4],

769 Bader and Kolda show how to efficiently carry out MTTKRP as well as other funda-  
 770 mental tensor operations on sparse tensors in MATLAB. GigaTensor [22] is a parallel  
 771 implementation of CP-ALS using the Map-Reduce framework. DFacTo [16] is a C++  
 772 implementation with distributed memory parallelism using MPI, and it uses a partic-  
 773 ular formulation of MTTKRP using sparse matrix-vector multiplication. SPLATT [40]  
 774 is an efficient parallelization of MTTKRP and CP-ALS both in shared [42] and dis-  
 775 tributed memory [40] environments using OpenMP and MPI, and is implemented in  
 776 C. It uses a medium-grain distributed parallelism with a Cartesian partitioning of the  
 777 tensor, and generalizes this technique to the tensor completion problem [41]. It is the  
 778 fastest publicly available CP-ALS implementation in the existing literature, and their  
 779 approach translates to performing  $N(N-1)$  TTMVs in performing the MTTKRP in  
 780 the main CP-ALS iteration. Karlsson et al. similarly discuss a parallel computation  
 781 of the tensor completion problem using CP formulation [23] in which they replicate  
 782 the entire factor matrix  $\mathbf{U}^{(n)}$  among MPI processes unlike our approach, and report  
 783 scalability results only up to 100 cores. For computing the CP decomposition of dense  
 784 tensors, Phan et al. [35] proposes a scheme that divides the tensor modes into two  
 785 sets, pre-computes the TTMVs for each mode set, and finally reuses these partial  
 786 results to obtain the final MTTKRP result in each mode. This provides a factor of 2  
 787 improvement in the number of TTMVs over the traditional approach, and our dimen-  
 788 sion tree-based framework can be considered as the generalization of this approach  
 789 that provides a factor of  $N/\log N$  improvement.

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

796 Aside from CP decomposition, Baskaran et al. [6] provide a shared-memory par-  
 797 allel implementation for the Tucker decomposition of sparse tensors. We [27] provide  
 798 efficient shared and distributed memory parallelization of the Tucker decomposition  
 799 for sparse tensors using OpenMP and MPI. Austin et al. [3] discuss a high perfor-  
 800 mance distributed memory parallelization of dense Tucker factorization in the context  
 801 of data compression. Finally, Perros et al. [34] investigate an efficient computation of  
 802 hierarchical Tucker decomposition for sparse tensors.

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

808 **6.1. Dataset and environment.** We experimented with five real-world tensors  
 809 whose sizes are shown in Table 1. Netflix tensor has user  $\times$  movie  $\times$  time dimensions,  
 810 which we formed from the data of the Netflix Prize competition [7]. In this tensor,  
 811 nonzeros correspond to the user reviews for movies, and the review date extends the  
 812 data to the third dimension. The values of the nonzeros are determined by the cor-  
 813 responding review scores given by the users. We obtained the NELL tensor from  
 814 the Never Ending Language Learning (NELL) knowledge database of the “Read the  
 815 Web” project [10], which consists of tuples of the form (entity, relation, entity) such  
 816 as (“Chopin”, “plays musical instrument”, “piano”). The nonzeros of this tensor cor-  
 817 respond 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

818 be the “belief” scores given by the learning algorithms used in NELL. Delicious and  
819 Flickr are the datasets for the web-crawl of Delicious.com and Flickr.com during 2006  
820 and 2007, which are formed by Görlitz et al. [17]. These datasets consist of tuples of  
821 form (time  $\times$  users  $\times$  resources  $\times$  tags); hence, we form 4-mode tensors out of these  
822 tuples. We obtained the Amazon review dataset from SNAP [32], which contains  
823 product review texts by users. We first processed this dataset with the standard text  
824 processing routines. We used the `nltk` package [8] in Python to tokenize the review  
825 text, to discard the stop words, to apply Porter stemmer, and to keep the words that  
826 are in the US, GB, or CA dictionaries. Afterwards, we retained only the words with  
827 at least five occurrences in the whole review set. Then, we created a three dimen-  
828 sional tensor whose dimensions correspond to the users, products, and retained words.  
829 Numerical values are set to the frequency of a word in a review.

830 We conducted experiments on a shared memory and a separate distributed mem-  
831 ory system. The shared memory system has two CPU sockets (Intel(R) Xeon(R)  
832 E5-2695 v3) each having 14 cores at a clock speed of 2.30GHz with Turbo Boost dis-  
833 abled. The system has a total memory of size 768GB, and each socket has L1, L2,  
834 L3 caches of sizes of 32KB, 256KB, and 35MB, respectively. All codes are compiled  
835 with `gcc/g++-5.3.0` using OpenMP directives and compiler options `-O3`, `-ffast-math`,  
836 `-funroll-loops`, `-ftree-vectorize`, `-fstrict-aliasing` on this shared memory system. The  
837 distributed memory system is an IBM Blue Gene/Q cluster. This system consists of  
838 6 racks of 1024 nodes with each node having 16GB of memory and a 16-core IBM  
839 PowerPC A2 processor running at 1.6GHz. We ran our experiments up to 256 nodes  
840 (4096 cores). Each core of PowerPC A2 can handle one arithmetic and memory oper-  
841 ation simultaneously; therefore, we assigned 32 threads per node (2 threads per core)  
842 for better performance. On this system, all codes were compiled using the Clang C++  
843 compiler (version 3.5.2) with IBM MPI wrapper using the same optimization flags,  
844 and linked against IBM ESSL library for LAPACK and BLAS routines.

845 We also used synthetic tensors created randomly having 4, 8, 16, and 32 dimen-  
846 sions. In these random tensors, each dimension is of size 10M, and there are 100M  
847 nonzeros with a uniform random distribution of indices. Using these tensors, we  
848 measure the effect of tensor dimensionality on the performance.

849 We provide the dimension-tree based CP-ALS implementation in our tensor fac-  
850 torization library called HYPERTENSOR. It is a C++11 implementation providing  
851 shared and distributed memory parallelism through OpenMP and MPI libraries. We  
852 compared our code against SPLATT v1.1.1 [40], a C code with OpenMP and MPI  
853 parallelizations. The “winner” for each test case in the results are highlighted with  
854 bold font.

855 **6.2. Shared memory experiments.** We compare the shared memory perfor-  
856 mance of the dimension tree-based CP-ALS algorithm with the state of the art. We  
857 experimented with four methods called **ht-tree2**, **ht-tree3**, **ht-tree**, and **splatt**.  
858 The **ht-tree** method, implemented in HYPERTENSOR, uses a full BDT to carry out

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

879 **6.2.1. Sequential execution.** In Table 2, we give the sequential per-iteration  
 880 run time of all methods. We report the run time in seconds for **splatt**, and the  
 881 relative speedup with respect to **splatt** for the other three methods. We first note  
 882 that **ht-tree2** runs slightly slower than **splatt** on three dimensional Amazon and  
 883 NELL tensors (0.99x and 0.87x), and notably slower on Netflix tensor (0.60x). This  
 884 is so because SPLATT has a specially tuned implementation for 3-dimensional tensors,  
 885 whereas we use a single code for all dimensions. On all higher dimensional tensors,  
 886 **ht-tree2** performs significantly better than **splatt**, up to 2.08x on Random8D, which  
 887 shows the efficiency of our implementation for  $N$ -dimensional tensors even before  
 888 using a BDT. The gap between **splatt** and **ht-tree2** narrows using Random16D,  
 889 as **ht-tree2** depletes the memory in one NUMA node, which is discussed more in  
 890 subsection 6.2.4, and starts accessing the distant memory in the other NUMA node.  
 891 **ht-tree2** gets out of memory using Random32D as it stores  $O(N^2)$  index arrays.

892 We now measure the effect of dimension trees by comparing **ht-tree** with **ht-**  
 893 **tree2** in Table 2. These two methods use the same TTMV implementation, whereas  
 894 **ht-tree** uses a full BDT. On Delicious, Flickr, Netflix, and NELL, **ht-tree** obtains  
 895 1.78x, 1.61x, 1.63x, and 1.47x speedup over **ht-tree2** thanks to the BDT. Likewise,  
 896 on random tensors, we observed 1.43x, 1.91x, 3.01x speedup on tensors Random4D,  
 897 Random8D, and Random16D, respectively, using **ht-tree**. This validates our perfor-  
 898 mance expectation (Theorem 2) that as the dimensionality of the tensor increases, a  
 899 BDT results in significantly fewer TTMVs hence better performance.

900 Comparing **ht-tree** with **splatt** similarly yields a speedup of 1.98x, 1.98x, 1.28x,  
 901 2.05x, 3.97x, 3.94x, and 5.96x on tensors Delicious, Flickr, NELL, Random4D, Ran-  
 902 dom8D, Random16D, and Random32D, respectively, which similarly meets our ex-  
 903 pectation of performance gain from Theorem 2. On Amazon, **ht-tree** was only 2%  
 904 faster, whereas on Netflix, **splatt** was only 2% faster, which was the only instance in  
 905 which **splatt** had a slight edge over **ht-tree**.

906 Finally, we note that the performance gap between **ht-tree** and **ht-tree3** widens  
 907 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.

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

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.

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

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

911 **6.2.2. Shared memory parallel execution.** In Table 3, we give the run time  
 912 results of all methods with shared memory parallelism using 14 threads. We first note  
 913 that in HYPERTENSOR, using dimension trees consistently yields better execution  
 914 times. Using **ht-tree**, we obtain 2.15x, 1.66x, 1.58x, 1.32x, and 1.10x speedup over  
 915 **ht-tree2** on Delicious, Flickr, Netflix, NELL, and Amazon tensors, respectively. For  
 916 random tensors, we get 1.62x, 2.69x, and 4.23x speedup on Random4D, Random8D,  
 917 and Random16D. Comparing **ht-tree** with **splatt**, we observe a speedup of 2.00x,  
 918 1.81x, 1.46x, 1.47x, 2.18x, 3.47x, and 5.65x on tensors Delicious, Flickr, NELL, Ran-  
 919 dom4D, Random8D, Random16D, and Random32D, respectively. This demonstrates  
 920 that the use of a BDT in CP-ALS computations can be effectively parallelized in a  
 921 shared memory setting, on top of significantly reducing the amount of TTMV work.  
 922 On three dimensional Amazon and Netflix tensors, **splatt** has a slight edge over **ht-**  
 923 **tree** by 5% and 14% faster executions, respectively. Another point to note is that  
 924 **splatt** has somewhat better parallel speedup in general (over its own sequential run  
 925 time) than **ht-tree2** and **ht-tree**. This is mostly due to the fact that TTMV is a  
 926 memory-bound computation; hence, once the memory bandwidth is fully utilized, one  
 927 cannot expect further speedup through multi-threading. When performing TTMVs,  
 928 our implementation makes slightly more memory accesses due to extra pointer ar-  
 929 rays 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.

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

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

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

935 **6.2.3. Preprocessing cost.** In Table 4, we provide symbolic TTV costs of our  
 936 methods as well as the precomputation cost of **splatt** for setting up its data struc-  
 937 tures. All runtimes are for a sequential execution; neither SPLATT nor HYPER TENSOR  
 938 parallelizes this step in their current version. We first note that **ht-tree** incurs sig-  
 939 nificantly less cost than **ht-tree2** in all instances. This is expected as **ht-tree2** has  
 940  $O(N^2)$  index arrays to be sorted, whereas **ht-tree** has only  $O(N \log N)$  of them. For  
 941 the same reason, **ht-tree** gets notably faster than **ht-tree3** as the tensor dimensionality  
 942 increases to 32. The cost is comparable between **splatt** and **ht-tree** for Delicious,  
 943 Flickr, Netflix, NELL, and Amazon tensors, but **splatt** takes significantly less time  
 944 for higher dimensional random tensors, up to 3.97x on Random8D, as it sorts only  
 945  $O(N)$  arrays.

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

951 **6.2.4. Memory usage.** We provide the memory consumption of all methods in  
 952 Table 5. The first column corresponds to the amount of memory used to store factor  
 953 matrices, which is common to all methods. We give the memory usage for storing  
 954 index arrays in GBs for **splatt**, and as the ratio to **splatt** for all other methods.  
 955 We also give the memory consumption for storing the value matrices of intermedi-  
 956 ate tensors for **ht-tree3** and **ht-tree** in GBs. We first note that **ht-tree2** uses the  
 957 highest amount of memory to store index arrays as expected, up to 8.32 times more  
 958 than **splatt** on Random16D. In all tensors, the amount of index memory used by **ht-**  
 959 **tree** is only slightly higher than **splatt**, the worst case being Flickr tensor for which  
 960 **ht-tree** consumes 1.35 times more memory in comparison. There is a multitude of  
 961 reasons for this observation. First, even though **splatt** uses only  $O(N)$  index arrays,  
 962 we realized upon inspecting the implementation that it uses two different representa-  
 963 tions 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	<b>splatt</b>	<b>ht-tree2</b>	<b>ht-tree3</b>		<b>ht-tree</b>	
		index	index	index	value	index	value
Delicious	3	<b>7</b>	2.44	*	*	1.21	5.7
Flickr	4.5	<b>4.8</b>	2.29	*	*	1.35	4.3
Netflix	0.1	<b>3.4</b>	1.85	*	*	1.29	1.4
NELL	0.6	<b>2.5</b>	1.88	*	*	1.28	0.5
Amazon	1.4	<b>49.6</b>	1.84	*	*	1.32	65.4
Random4D	6	<b>9.1</b>	2.21	*	*	1.12	10.2
Random8D	11.9	<b>21</b>	4.24	1.08	15.3	1.08	30.5
Random16D	23.8	<b>44.9</b>	8.32	1.47	15.3	1.23	45.7
Random32D	47.7	<b>94.3</b>	-	2.40	15.3	1.31	61

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

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

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

976 **6.3. Distributed memory experiments.** We compare the performance and  
 977 the scalability of the fine- and the medium-grain parallel CP-ALS algorithms. In  
 978 these experiments, we do not use SPLATT software to benchmark medium-grain par-  
 979 allelization for two reasons. First, we would like to compare the effect of load balance  
 980 and communication cost in different algorithms using different partitionings, while  
 981 isolating the effects of the efficiency of local CP-ALS computations. Since SPLATT’s  
 982 medium-grain implementation does not use BDTs for local TTMVs, and is slower,  
 983 comparing it against HYPERTENSOR’s fine-grain implementation which has faster  
 984 local TTMVs would not be fair, nor would correctly reveal the effect of different  
 985 partitioning strategies. Second, we were not able to get SPLATT to work on our  
 986 distributed system despite our full efforts. Therefore, we instead performed medium-  
 987 grain partitioning of tensors following the description of SPLATT’s heuristic [40], and  
 988 ran HYPERTENSOR on these partitions which incurs the same cost in terms of the  
 989 communication volume and the number of messages as SPLATT, while using more effi-  
 990 cient TTMV kernels. For local CP-ALS computations, we use the BDT-based method  
 991 **ht-tree** for shared memory parallelism, as it gives the best performance. This way,  
 992 the experiments become more precise in terms of measuring the influence of medium-  
 993 and fine-grain algorithms and associated partitionings on parallel scalability.

994 We investigate the performance in two tables. In Table 6, we give the strong  
 995 scalability results of the medium- and the fine-grain algorithms up to 256 MPI ranks  
 996 using 4096 cores. Since we achieved the maximum scalability in most tensors with 256  
 997 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.

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

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

$\#nodes \times \#cores$	Amazon		
	med-gd	fine-rd	fine-lb
$64 \times 1$	-	0.55	<b>36.303</b>
$64 \times 16$	-	4.12	<b>19.37</b>
$128 \times 16$	31.82	5.86	<b>36.05</b>
$256 \times 16$	34.38	8.58	<b>63.69</b>

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

1000 In Table 6, we compare the execution of Algorithm 5 with three partitioning  
 1001 methods. The **fine-hp** and **fine-lb** correspond, respectively, to the standard hyper-  
 1002 graph partitioning and the label-propagation-like heuristic of subsection 4.2.4. For  
 1003 **fine-hp**, we used PaToH [12] with the default settings. On Amazon tensor, we could  
 1004 not obtain results for **fine-hp** as the tensor was too big for PaToH to handle. For  
 1005 **fine-lb**, we ran the three alternatives, each for three passes, and chose the partitioning  
 1006 with the smallest cut. The **med-gd** method corresponds to the medium-grain parti-  
 1007 tioning heuristic [40]. The **fine-rd** method refers to the random partitioning of the  
 1008 fine-grain hypergraph, which is given as a reference to illustrate the impact of a good  
 1009 partitioning. Due to memory constraints, we were not able to execute Algorithm 5  
 1010 on a single node, as the original tensors are large. Therefore, for each tensor, we give  
 1011 the results starting from the minimum number of nodes needed, and for the same  
 1012 instance we also give the single threaded results. We use the run time of the single  
 1013 threaded execution of the fastest method as our baseline (highlighted with shaded  
 1014 cells) in computing the parallel speedup. In passing from one core per node to 16  
 1015 cores per node, we see some speedup over 16 in Table 6; this is because we use two  
 1016 threads per core (see subsection 6.1).

1017 In order to be able to analyze the speedup results of Table 6, we give the number  
 1018 of tensor nonzeros per part, computational load (the number of Hadamard products),  
 1019 communication volume, and the number of messages incurred by these three parti-  
 1020 tionings, using 256 MPI ranks in Table 7. For the four performance metrics, we  
 1021 give the maximum and the average value observed across all processes. We see in all  
 1022 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.
<i>Delicious</i>								
fine-hp	547K	547K	1947K	1807K	<b>199K</b>	<b>137K</b>	2039	2018
fine-lb	564K	547K	<b>1849K</b>	<b>1737K</b>	309K	265K	2040	2040
fine-rd	<b>550K</b>	547K	2747K	2737K	1083K	1080K	2040	2040
medium-gd	598K	547K	2353K	2214K	624K	571K	<b>1096</b>	<b>1096</b>
<i>Flickr</i>								
fine-hp	<b>441K</b>	441K	<b>1334K</b>	<b>1221K</b>	<b>54K</b>	<b>38K</b>	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	<b>1152</b>	<b>1152</b>
<i>Netflix</i>								
fine-hp	<b>392K</b>	392K	1439K	1211K	49K	<b>19K</b>	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	<b>1184K</b>	<b>1177K</b>	<b>26K</b>	24K	<b>642</b>	<b>642</b>
<i>NELL</i>								
fine-hp	<b>307K</b>	307K	1219K	<b>787K</b>	<b>46K</b>	<b>23K</b>	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	<b>871K</b>	855K	58K	51K	<b>583</b>	<b>570</b>
<i>Amazon</i>								
fine-lb	5104K	4955K	<b>16871K</b>	<b>16241K</b>	<b>211K</b>	203K	1503	1503
fine-rd	<b>4962K</b>	4955K	20964K	20940K	4656K	4651K	1530	1530
medium-gd	19984K	4955K	50023K	16255K	230K	<b>170K</b>	<b>550</b>	<b>514</b>

1023 gracefully translates into balancing the actual computational load.

1024 Using 256 MPI ranks on Delicious, **fine-hp** and **fine-lb** are 2.5x and 1.9x faster  
1025 over **med-gd**. We observe in Table 7 that this is due to better minimization of the  
1026 total and the maximum communication volume. On Flickr, **fine-hp** is 3.9x faster  
1027 than **med-gd** at 256 MPI ranks with 14.7x and 10.6x less total and maximum com-  
1028 munication volume, while **fine-lb** shows a speedup of 3.2x over **med-gd** with 6.4x  
1029 and 5.4x less total and maximum communication cost. In both tensors, **med-gd** re-  
1030 sults in about the half the communication volume of **fine-rd**. In overall, on Delicious  
1031 **fine-hp** and **fine-lb** obtain 183x and 142x speedup using 4096 cores, whereas **med-**  
1032 **gd** and **fine-rd** give 75x and 40x speedup for the same tensor. On Flickr, **fine-hp**  
1033 and **fine-lb** similarly yield 178x and 148x speedup using 4096 cores, while **med-gd**  
1034 and **fine-rd** could achieve 46x and 23x speedup. For the Delicious and Flickr tensors,  
1035 while passing from 8 nodes (with  $8 \times 16$  cores) to 256 nodes (with  $256 \times 16$  cores),  
1036 **med-gd** results in 5.29x and 4.32x speedup. The **fine-hp** and **fine-lb** result in 11.33x  
1037 and 10.11x speedup for Delicious, and 11.62x and 10.50x speedup for Flickr in the  
1038 same scenario. **fine-rd** is significantly slower than the other methods, incurring the  
1039 highest communication as shown in Table 7.

1040 On Netflix and NELL, **med-gd** yields 321x and 197x speedup using 4096 cores.  
1041 **fine-hp** shows a comparable performance with 261x and 164x speedup, whereas **fine-**  
1042 **lb** is slightly slower than **fine-hp** with 236x and 158x speedup. In passing from  
1043 4 nodes (with  $4 \times 16$  cores) to 256 nodes (with  $256 \times 16$  cores), **med-gd** results in  
1044 12.73x and 12.03x speedup for Netflix and NELL, respectively. The **fine-hp** and **fine-**  
1045 **lb** partitioning result in 10.18x and 11.18x speedup for Netflix, and 9.72x and 9.32x  
1046 speedup for NELL in the same scenario. **fine-rd** is similarly the slowest of all methods  
1047 giving 67x and 62x speedup for these two tensors. Using Netflix and NELL, **med-**

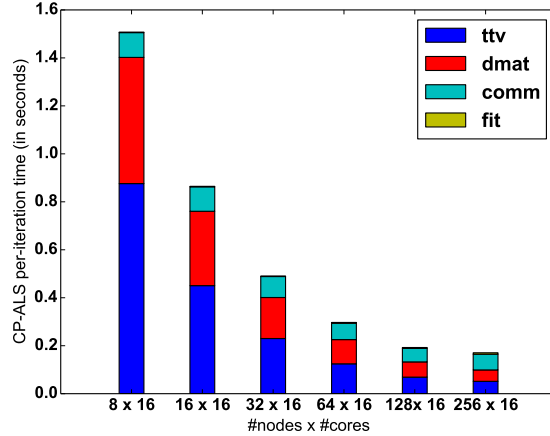


Fig. 3: Running time dissection of a parallel CP-ALS iteration using **fine-hp** and **ht-tree** scheme. The legends **ttv**, **dmat**, **comm**, and **fit** correspond to the time spent for TTMVs, dense matrix operations following the TTMVs, communication, and fit computation. The time for fit computation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

1048 **gd** gets 23% and 20% faster than **fine-hp**, and 36% and 25% faster than **fine-lb**. In  
 1049 [Table 7](#), we see that this is due to **med-gd** incurring smaller maximum communication  
 1050 volume and having fewer messages. We investigated this outcome and observed that  
 1051 when a tensor is long in one mode and short in all others, the communication due  
 1052 to the long mode dominates the overall cost, and the communication for the small  
 1053 modes remains negligible in comparison. On Netflix with  $P = 256$  MPI ranks, using  
 1054 **med-gd** with  $P = p \times q \times r$  topology, the worst case communication volume for the  
 1055 first mode is upper-bounded by  $480K(qr - 1)$ , as  $I_1 = 480K$  indices are distributed  
 1056 to  $p$  process “slices” each with  $qr$  processes. Similarly, for the second and the third  
 1057 modes, the worst case communication volumes are  $17K(pr - 1)$  and  $2K(pq - 1)$ . In  
 1058 such cases, choosing a large  $p$  and smaller  $q$  and  $r$  significantly reduces the worst-  
 1059 case communication cost in the first mode, while the cost in other modes stays low.  
 1060 The medium-grain heuristic achieves this. Specifically, on Netflix, the medium-grain  
 1061 heuristic chooses a grid size of  $64 \times 4 \times 1$ . This advantage is lost when there are at  
 1062 least two long dimensions (see Delicious and Flickr), as using more processes in one  
 1063 long mode can increase the communication significantly in the other long modes.

1064 On Amazon tensor, **med-gd** starts to lose scalability at 256 MPI ranks. We ob-  
 1065 serve in [Table 7](#) that this is due to load imbalance. Amazon tensor has some relatively  
 1066 “dense” slices that make load balancing difficult for the medium-grain heuristic. This  
 1067 problem never arises in the fine-grain partitioning due to finer granularity of tasks; as  
 1068 a result, **fine-lb** runs 1.85x faster than **med-gd** using 256 MPI ranks. In this tensor,  
 1069 from 128 nodes to 256 nodes, **med-gd** displays a speedup of 1.08. With **fine-lb**, the  
 1070 parallel algorithm enjoys 1.86x speedup in passing from 64 nodes to 128 nodes, and  
 1071 3.2x speedup in passing from 64 to 256 nodes. In overall, **med-gd** gives 34x speedup  
 1072 over the baseline whereas **fine-lb** gets significantly faster with 63x speedup.

1073 In [Figure 3](#), we present the dissection of the parallel run time for a CP-ALS  
 1074 iteration on Flickr tensor using 256 MPI ranks. We choose Flickr as representative,  
 1075 as it includes the highest proportion of dense matrix operations in comparison to  
 1076 all other tensors. Despite this fact and using a BDT for faster TTMVs, the TTMV

1077 step still remains to be the dominant computational cost. In this figure, we first  
 1078 observe that the workload due to TTMV and dense matrix computations decrease  
 1079 with the increasing number of processes. Second, we expect in general that having  
 1080 more processes increases the total communication volume; yet we observe in the plot  
 1081 that the communication cost declines until 128 MPI ranks. This is because a good  
 1082 partitioning can reduce the communication volume per process (while increasing the  
 1083 total communication volume). At 256 MPI ranks, however, communication cost starts  
 1084 to increase and become the bottleneck. The fit computation takes a negligible amount  
 1085 of time hence is not discernible in the plot.

1086 On Flickr tensor, the three variants of the **fine-lb** took 58.38, 89.66, and 65.04  
 1087 seconds to partition the hypergraph, **med-gd** took 190 seconds, and **fine-hp** took  
 1088 207 minutes. In all data instances, **fine-lb** gives good results while being a fast par-  
 1089 titioning heuristic. **fine-hp** consistently provides better partitions than **fine-lb** in all  
 1090 instances, yet the partitioning cost might render it impractical to use in real-world  
 1091 scenarios. **med-gd** heuristic is only effective when the tensor nonzeros are homoge-  
 1092 neously distributed, and the tensor has only one large dimension. One might consider  
 1093 reducing the communication volume on a medium-grain topology using hypergraph  
 1094 partitioning, yet the high number of constraints prevents this approach from being  
 1095 amenable. Therefore, we believe that **fine-lb** serves well in most practical situations.

1096 **7. Conclusion.** We investigated an efficient computation of successive tensor-  
 1097 times-vector multiplication in the context of the well-known CP-ALS algorithm for  
 1098 sparse tensor factorization. We introduced a computational scheme using dimension  
 1099 trees that asymptotically reduces the computational cost of the TTMV operations  
 1100 for higher order tensors while using a reasonable amount of memory. Our technique  
 1101 provides performance benefits for lower order tensors, and gets progressively better  
 1102 as the dimensionality of the tensor increases in comparison to the state of the art.  
 1103 We proposed an effective shared memory parallelization of this method with a pre-  
 1104 computation step in order to efficiently carry out numerical computations within  
 1105 the CP-ALS iterations. We introduced a fine-grain parallelization approach in the  
 1106 distributed memory setting, compared it against a recently proposed medium-grain  
 1107 variant, discussed good partitionings for both approaches, and validated these findings  
 1108 with experiments on real-world tensors. The proposed computational scheme can  
 1109 be applied to both dense and sparse tensors as well as other tensor decomposition  
 1110 algorithms involving successive tensor-times-vector and -matrix multiplications. We  
 1111 are planning to investigate this potential in our future work.

1112 **Acknowledgments.** Some preliminary experiments were carried out using the  
 1113 workstations and the PSMN cluster at ENS Lyon. This work was performed using  
 1114 HPC resources from GENCI-[TGCC/CINES/IDRIS] (Grant 2016 - i2016067501).

1115

## REFERENCES

- 1116 [1] E. ACAR, D. M. DUNLAVY, AND T. G. KOLDA, *A scalable optimization approach for fitting*  
 1117 *canonical tensor decompositions*, Journal of Chemometrics, 25 (2011), pp. 67–86.  
 1118 [2] C. A. ANDERSSON AND R. BRO, *The N-way toolbox for MATLAB*, Chemometrics and Intelligent  
 1119 Laboratory Systems, 52 (2000), pp. 1–4.  
 1120 [3] W. AUSTIN, G. BALLARD, AND T. G. KOLDA, *Parallel tensor compression for large-scale scien-*  
 1121 *tific data*, in IEEE International Parallel and Distributed Processing Symposium (IPDPS),  
 1122 Chicago, IL, USA, May 23–27, 2016, pp. 912–922.  
 1123 [4] B. W. BADER AND T. G. KOLDA, *Efficient MATLAB computations with sparse and factored*  
 1124 *tensors*, SIAM Journal on Scientific Computing, 30 (2007), pp. 205–231.

- 1125 [5] B. W. BADER, T. G. KOLDA, ET AL., *Matlab tensor toolbox version 2.6*. Available online  
1126 <http://www.sandia.gov/~tgkolda/TensorToolbox/>, February 2015.
- 1127 [6] M. BASKARAN, B. MEISTER, N. VASILACHE, AND R. LETHIN, *Efficient and scalable computa-*  
1128 *tions with sparse tensors*, in IEEE Conference on High Performance Extreme Computing  
1129 (HPEC), Sept 2012, pp. 1–6.
- 1130 [7] J. BENNETT AND S. LANNING, *The Netflix Prize*, in Proceedings of KDD cup and workshop,  
1131 vol. 2007, 2007, p. 35.
- 1132 [8] S. BIRD, E. LOPER, AND E. KLEIN, *Natural Language Processing with Python*, O’Reilly Media  
1133 Inc., 2009.
- 1134 [9] J. BUURLAGE, *Self-improving sparse matrix partitioning and bulk-synchronous pseudo-*  
1135 *streaming*, master’s thesis, Utrecht University, 2016.
- 1136 [10] A. CARLSON, J. BETTERIDGE, B. KISIEL, B. SETTLES, E. R. H. JR., AND T. M. MITCHELL,  
1137 *Toward an architecture for never-ending language learning*, in AAAI, vol. 5, 2010, p. 3.
- 1138 [11] D. J. CARROLL AND J. CHANG, *Analysis of individual differences in multidimensional scaling*  
1139 *via an N-way generalization of “Eckart-Young” decomposition*, Psychometrika, 35 (1970),  
1140 pp. 283–319.
- 1141 [12] Ü. V. ÇATALYÜREK AND C. AYKANAT, *PaToH: A Multilevel Hypergraph Partitioning Tool,*  
1142 *Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533  
1143 Turkey. <http://bmi.osu.edu/~umit/software.htm>, 1999.
- 1144 [13] Ü. V. ÇATALYÜREK AND C. AYKANAT, *A hypergraph-partitioning approach for coarse-grain*  
1145 *decomposition*, in Supercomputing, ACM/IEEE 2001 Conference, Denver, Colorado, 2001,  
1146 p. 42.
- 1147 [14] Ü. V. ÇATALYÜREK, C. AYKANAT, AND B. UÇAR, *On two-dimensional sparse matrix partition-*  
1148 *ing: Models, methods, and a recipe*, SIAM Journal on Scientific Computing, 32 (2010),  
1149 pp. 656–683.
- 1150 [15] Ü. V. ÇATALYÜREK, *Hypergraph Models for Sparse Matrix Partitioning and Reordering*, PhD  
1151 thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.
- 1152 [16] J. H. CHOI AND S. V. N. VISHWANATHAN, *DFacTo: Distributed factorization of tensors*, in  
1153 27th Advances in Neural Information Processing Systems, Montreal, Quebec, Canada,  
1154 2014, pp. 1296–1304.
- 1155 [17] O. GÖRLITZ, S. SIZOV, AND S. STAAB, *PINTS: Peer-to-peer infrastructure for tagging systems*,  
1156 in Proceedings of the 7th International Conference on Peer-to-Peer Systems, Berkeley, CA,  
1157 USA, 2008, USENIX Association, p. 19.
- 1158 [18] L. GRASEDYCK, *Hierarchical singular value decomposition of tensors*, SIAM Journal on Matrix  
1159 Analysis and Applications, 31 (2010), pp. 2029–2054.
- 1160 [19] R. A. HARSHMAN, *Foundations of the PARAFAC procedure: Models and conditions for an*  
1161 *“explanatory” multi-modal factor analysis*, UCLA Working Papers in Phonetics, 16 (1970),  
1162 pp. 1–84.
- 1163 [20] V. HENNE, *Label propagation for hypergraph partitioning*, master’s thesis, Karlsruhe Institute  
1164 of Technology, Germany, 2015.
- 1165 [21] J. HÅSTAD, *Tensor rank is NP-complete*, Journal of Algorithms, 11 (1990), pp. 644–654.
- 1166 [22] U. KANG, E. PAPALEXAKIS, A. HARPALE, AND C. FALOUTSOS, *GigaTensor: Scaling tensor*  
1167 *analysis up by 100 times - Algorithms and discoveries*, in Proceedings of the 18th ACM  
1168 SIGKDD International Conference on Knowledge Discovery and Data Mining, New York,  
1169 NY, USA, 2012, ACM, pp. 316–324.
- 1170 [23] L. KARLSSON, D. KRESSNER, AND A. USCHMAJEW, *Parallel algorithms for tensor completion*  
1171 *in the CP format*, Parallel Computing, 57 (2016), pp. 222–234.
- 1172 [24] G. KARYPIS AND V. KUMAR, *Multilevel algorithms for multi-constraint hypergraph partitioning*,  
1173 Tech. Report 99-034, University of Minnesota, Department of Computer Science/Army  
1174 HPC Research Center, Minneapolis, MN 55455, November 1998.
- 1175 [25] O. KAYA AND B. UÇAR, *High-performance parallel algorithms for the Tucker decomposition of*  
1176 *higher order sparse tensors*, Tech. Report RR-8801, Inria, Oct 2015.
- 1177 [26] O. KAYA AND B. UÇAR, *Scalable sparse tensor decompositions in distributed memory systems*,  
1178 in Proceedings of the International Conference for High Performance Computing, Network-  
1179 ing, Storage and Analysis, New York, NY, USA, 2015, ACM, pp. 77:1–77:11.
- 1180 [27] O. KAYA AND B. UÇAR, *High performance parallel algorithms for the Tucker decomposition of*  
1181 *sparse tensors*, in 45th International Conference on Parallel Processing (ICPP ’16), Aug  
1182 2016, pp. 103–112.
- 1183 [28] T. G. KOLDA AND B. BADER, *The TOPHITS model for higher-order web link analysis*, in  
1184 Proceedings of Link Analysis, Counterterrorism and Security, 2006.
- 1185 [29] T. G. KOLDA AND B. BADER, *Tensor decompositions and applications*, SIAM Review, 51 (2009),  
1186 pp. 455–500.



- 1187 [30] L. D. LATHAUWER AND B. D. MOOR, *From matrix to tensor: Multilinear algebra and signal*  
 1188 *processing*, in Institute of Mathematics and Its Applications Conference Series, vol. 67,  
 1189 1998, pp. 1–16.
- 1190 [31] T. LENGAUER, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley–Teubner, Chich-  
 1191 ester, U.K., 1990.
- 1192 [32] J. LESKOVEC AND A. KREVL, *SNAP Datasets: Stanford large network dataset collection*. <http://snap.stanford.edu/data>, June 2014.
- 1193 [33] J. LI, J. CHOI, I. PERROS, J. SUN, AND R. VUDUC, *Model-driven sparse CP decomposition for*  
 1194 *higher-order tensors*, in IPDPS 2017, 31th IEEE International Symposium on Parallel and  
 1195 Distributed Processing, Orlando, FL, USA, May 2017, pp. 1048–1057.
- 1196 [34] I. PERROS, R. CHEN, R. VUDUC, AND J. SUN, *Sparse hierarchical Tucker factorization and its*  
 1197 *application to healthcare*, in Data Mining (ICDM), 2015 IEEE International Conference  
 1198 on, Nov 2015, pp. 943–948.
- 1200 [35] A. H. PHAN, P. TICHAVSKÝ, AND A. CICHOCKI, *Fast alternating LS algorithms for high order*  
 1201 *CANDECOMP/PARAFAC tensor factorizations*, IEEE Transactions on Signal Processing,  
 1202 61 (2013), pp. 4834–4846.
- 1203 [36] S. RENDLE AND T. S. LARS, *Pairwise interaction tensor factorization for personalized tag rec-*  
 1204 *ommendation*, in Proceedings of the Third ACM International Conference on Web Search  
 1205 and Data Mining, WSDM '10, New York, NY, USA, 2010, ACM, pp. 81–90.
- 1206 [37] S. RENDLE, B. M. LEANDRO, A. NANOPOULOS, AND L. SCHMIDT-THIEME, *Learning optimal*  
 1207 *ranking with tensor factorization for tag recommendation*, in Proceedings of the 15th ACM  
 1208 SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09,  
 1209 New York, NY, USA, 2009, ACM, pp. 727–736.
- 1210 [38] G. M. SLOTA, K. MADDURI, AND S. RAJAMANICKAM, *PuLP: Scalable multi-objective multi-*  
 1211 *constraint partitioning for small-world networks*, in Proc. 2nd IEEE Int'l. Conf. on Big  
 1212 Data (BigData), IEEE, Oct. 2014, pp. 481–490.
- 1213 [39] S. SMITH AND G. KARYPIS, *Tensor-matrix products with a compressed sparse tensor*, in Pro-  
 1214 ceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms,  
 1215 ACM, 2015, p. 7.
- 1216 [40] S. SMITH AND G. KARYPIS, *A medium-grained algorithm for sparse tensor factorization*, in  
 1217 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016,  
 1218 Chicago, IL, USA, May 23-27, 2016, 2016, pp. 902–911.
- 1219 [41] S. SMITH, J. PARK, AND G. KARYPIS, *An exploration of optimization algorithms for high per-*  
 1220 *formance tensor completion*, Proceedings of the 2016 ACM/IEEE conference on Super-  
 1221 computing, (2016).
- 1222 [42] S. SMITH, N. RAVINDRAN, N. D. SIDIROPOULOS, AND G. KARYPIS, *SPLATT: Efficient and par-*  
 1223 *allel sparse tensor-matrix multiplication*, in 29th IEEE International Parallel & Distributed  
 1224 Processing Symposium, Hyderabad, India, May 2015, IEEE Computer Society, pp. 61–70.
- 1225 [43] P. SYMEONIDIS, A. NANOPOULOS, AND Y. MANOLOPOULOS, *Tag recommendations based on*  
 1226 *tensor dimensionality reduction*, in Proceedings of the 2008 ACM Conference on Recom-  
 1227 mender Systems, New York, NY, USA, 2008, ACM, pp. 43–50.
- 1228 [44] G. TOMASI AND R. BRO, *A comparison of algorithms for fitting the parafac model*, Computa-  
 1229 tional Statistics & Data Analysis, 50 (2006), pp. 1700 – 1734.
- 1230 [45] J. UGANDER AND L. BACKSTROM, *Balanced label propagation for partitioning massive graphs*, in  
 1231 Proceedings of the Sixth ACM International Conference on Web Search and Data Mining,  
 1232 WSDM '13, New York, NY, USA, 2013, ACM, pp. 507–516.
- 1233 [46] M. A. O. VASILESCU AND D. TERZOPOULOS, *Multilinear analysis of image ensembles: Tensor-*  
 1234 *Faces*, in Computer Vision—ECCV 2002, Springer, 2002, pp. 447–460.
- 1235 [47] N. ZHENG, Q. LI, S. LIAO, AND L. ZHANG, *Flickr group recommendation based on tensor de-*  
 1236 *composition*, in Proceedings of the 33rd International ACM SIGIR Conference on Research  
 1237 and Development in Information Retrieval, SIGIR '10, NY, USA, 2010, ACM, pp. 737–738.