A Parallel Nonzero CP Decomposition Algorithm for Higher Order Sparse Data Analysis

Oguz Kaya

Department of Computer Science
École Normale Superiéure de Lyon and INRIA, 46 Allée d'Italie 69007, Lyon, France
e-mail: oguz.kaya@ens-lyon.fr

***Abstract*—In the age of big data, tensor decomposition methods are increasingly being used due to their ability to naturally express and analyze high dimensional big data. Obtaining these decompositions gets very expensive both in terms of computational and memory requirements, particularly as the tensor's dimensionality increases. To efficiently handle such high dimensional tensors, we propose a parallel algorithm for computing a CP decomposition in which factor matrices are assumed to not contain any zero elements. This additional constraint enables a very efficient computation of the costliest step of the algorithms for computing CANDECOMP/PARAFAC (CP) decomposition for high dimensional tensors, and the performance gains increase with the dimensionality of tensor. For an N-dimensional tensor having k nonzero entries, our method provides $O(\log k)$ and $O(\log N \log k)$ faster preprocessing times, and performs $O(N)$ and $O(\log N)$ less work in computing a CP decomposition over two efficient state-of-the-art libraries SPLATT and HYPERTENSOR, respectively. With these algorithmic contributions and a highly tuned parallel implementation, we achieve up to 16.7x speedup in sequential, and up to 10.5x speedup in parallel executions over these libraries on a 28-core workstation. In doing so, we incur with up to 24x less preprocessing time, and use up to $O(\log N)$ less memory for storing intermediate computations. We show using a real-world tensor that the accuracy of our method is comparable to the standard CP decomposition.**

Keywords–parallel sparse tensor factorization; CP decomposition; higher order data analysis.

I. INTRODUCTION

In parallel with the overwhelming increase in the size of big data problems, the variety of data features also grows, which in turn raises the data dimensionality. Such high dimensional big data can be naturally modeled by tensors, or multi-dimensional arrays, and effectively analyzed using tensor decomposition methods. For this reason, tensors have been increasingly used in many application domains in the recent past, including the analysis of Web graphs [1], knowledge bases [2], recommender systems [3], signal processing [4], computer vision [5], health care [6], and many others [7]. In these applications, tensor decomposition algorithms are employed to perform a profound analysis of the high dimensional data to extract hidden information, or predict some missing data elements of interest. To this end, there have been considerable efforts in providing numerical algorithms for tensor decompositions [7], and in developing efficient computational methods and high performance software to render these algorithms amenable to use in real-world applications [8]–[13].

One of the most popular tensor decomposition methods is called the CANDECOMP/PARAFAC decomposition (CP, or CPD). The most common algorithm for computing the CP decomposition is an iterative method using alternating least squares (ALS), and is therefore called CP-ALS [14], [15]. At the core of the CP-ALS algorithm, each iteration involves a special operation called the matricized tensor-times Khatri-Rao product (MTTKRP). For a sparse N-dimensional tensor, this operation carries out the element-wise multiplication of $N-1$ matrix row vectors and accumulates their scaled sum, which is repeated for each nonzero element of the tensor. As the tensor gets higher dimensional, the cost of this operation dramatically increases, and efficiently carrying out this operation becomes crucial to be able to process such tensors. For this reason, there has been significant recent efforts in the literature towards efficiently computing this operation in particular, and CP-ALS in general, in different computational settings such as MATLAB [8], [16], MapReduce [17], shared memory [11], and distributed memory parallel environments [9], [13], [18], [19]. In this paper, we are interested in an efficient parallel computation of CP-ALS for high dimensional big sparse tensors in a shared memory environment, which is particularly motivated by emerging big data applications [6].

We summarize our contributions in this work as following:

- We introduce a method for efficiently computing MTTKRP for high dimensional sparse tensors. This scheme asymptotically reduces the computational cost of MTTKRP as well as the cost of a common precomputation step performed in the state-of-the-art.
- We provide an efficient parallelization of this computational scheme that runs up to 10.5 times faster than the state-of-the-art on a 28-core workstation.

The rest of the paper is organized as follows. In Section II, we provide our tensor notation and some tensor operations, then describe the CP-ALS algorithm and the MTTKRP operation. Next, in Section III, we provide an overview of existing methods for computing CP-ALS together with a summary of their computational and memory costs. Then, in Section IV we describe our method for computing MTTKRP and CP-ALS, which imposes a nonzero constraint on factor matrices to reduce the computational costs. We compare the complexity of our approach with the state of the art, and discuss a carefully tuned parallelization of this approach for a shared memory NUMA architecture. Finally, Section V provides a comparison of sequential and parallel executions of our method with two state-of-the-art implementations.

II. BACKGROUND

A. Notation

We mostly follow the tensor notation used in [7], [19]. We denote the set $\{1, \ldots, k\}$ of integers by $\mathbb{N}_k$ for $k \in \mathbb{Z}^+$. We denote vectors using bold lowercase Roman letters, as in $\mathbf{x}$. Similarly for matrices, we use bold uppercase Roman letters, e.g., $\mathbf{X}$. For tensors, we use bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{X}}$. We define the *order* of a tensor as the number of its *dimensions* or *modes*, and denote it by N. We use italic lowercase letters with corresponding indices to represent vector, matrix, and tensor

 ISBN: 978-1-61208-567-8

elements, e.g., x_i for a vector $\mathbf{x}$, x_{ij} for a matrix $\mathbf{X}$, and x_{ijk} for a 3-dimensional tensor $\boldsymbol{\mathcal{X}}$. For column vectors of a matrix, we use the corresponding lowercase letters with a subscript corresponding to the column index, e.g., $\mathbf{x}_i$ to denote the ith column of $\mathbf{X}$. A *slice* of a tensor in the nth mode is a set of tensor elements obtained by fixing the index only along the nth mode. For matrix rows and columns as well as tensor slices, we use the MATLAB notation, e.g., $\mathbf{X}(i,:)$ and $\mathbf{X}(:,j)$ are the ith row and the jth column of $\mathbf{X}$, whereas $\boldsymbol{\mathcal{X}}(:,:,k)$ represents the kth slice of $\boldsymbol{\mathcal{X}}$ in the third dimension.

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

The *Hadamard product* of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^I$ is a vector $\mathbf{w} = \mathbf{u} * \mathbf{v}, \mathbf{w} \in \mathbb{R}^I$, where $w_i = u_i \cdot v_i$ for $i \in \mathbb{N}_I$. Similarly, *Hadamard division* of the two vectors is a vector $\mathbf{w} = \mathbf{u} \oslash \mathbf{v}$ with elements $w_i = u_i / v_i$, $v_i \neq 0$. Hadamard product and division of matrices of same size are defined similarly. The *Kronecker product* of vectors $\mathbf{u} \in \mathbb{R}^I$ and $\mathbf{v} \in \mathbb{R}^J$ results in the vector $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ where $\mathbf{w} \in \mathbb{R}^{IJ}$ is defined as

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

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

$$\mathbf{W} = \mathbf{U} \odot \mathbf{V} = \left[\mathbf{u}_1 \otimes \mathbf{v}_1, \ldots, \mathbf{u}_K \otimes \mathbf{v}_K\right], \qquad (1)$$

where $\mathbf{W} \in \mathbb{R}^{IJ \times K}$. We use the shorthand notation $\circ_{i \neq n} \mathbf{U}^{(i)}$ to denote an associative operation $\mathbf{U}^{(1)} \circ \cdots \circ \mathbf{U}^{(n-1)} \circ \mathbf{U}^{(n+1)} \circ \cdots \circ \mathbf{U}^{(N)}$ over a set $\{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\}$ of matrices (and similarly for vectors).

B. CP decomposition

The rank-R CP-decomposition of a tensor $\boldsymbol{\mathcal{X}}$ expresses $\boldsymbol{\mathcal{X}}$ as the sum of R rank-1 tensors. For instance, for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$, with CP decomposition we obtain $\boldsymbol{\mathcal{X}} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ where $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$, and $\mathbf{c}_r \in \mathbb{R}^K$. This decomposition gives an element-wise approximation (or equality) $x_{i,j,k} \approx \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$. The minimum R value rendering this approximation an equality for all tensor elements is called as the *rank* (or CP-rank) of the tensor $\boldsymbol{\mathcal{X}}$. Here, the matrices $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_R]$, $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_R]$, and $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_R]$ are called the *factor matrices*, or *factors*. For an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, we use $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$ to refer to the factor matrices respectively having $I_1, \ldots, I_N$ rows and R columns.

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

1: **for** $n = 1, \ldots, N$ **do** ▶ Initialization
2: $\quad \mathbf{W}^{(n)} \leftarrow {\mathbf{U}^{(n)}}^T \mathbf{U}^{(n)}$
3: **repeat**
4: $\quad$ **for** $n = 1, \ldots, N$ **do**
5: $\quad\quad \mathbf{M}^{(n)} \leftarrow \mathbf{X}_{(n)} (\odot_{(i \neq n)} \mathbf{U}^{(i)})$ ▶ MTTKRP
6: $\quad\quad \mathbf{H}^{(n)} \leftarrow *_{(i \neq n)} \mathbf{W}^{(n)}$
7: $\quad\quad \mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)} {\mathbf{H}^{(n)}}^{\dagger}$
8: $\quad\quad \lambda \leftarrow$ COLUMN-NORMALIZE$(\mathbf{U}^{(n)})$
9: $\quad\quad \mathbf{W}^{(n)} \leftarrow {\mathbf{U}^{(n)}}^T \mathbf{U}^{(n)}$
10: **until** convergence or the maximum number of iterations
11: **return** $[\![\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

Figure 1. CP-ALS: ALS algorithm for computing CPD.

Input: $\boldsymbol{\mathcal{X}}$: N-dimensional sparse tensor
$\mathbf{U}^{(1)} \ldots \mathbf{U}^{(N)}$: Factor matrices
n: The dimension of matricization for MTTKRP
Output: $\mathbf{M}^{(n)}$: MTTKRP result matrix of size $I_n \times R$

1: $\mathbf{M}^{(n)} \leftarrow 0$
2: **for** $x_{i_1,\ldots,i_n,\ldots,i_N} \in \boldsymbol{\mathcal{X}}$ **do**
3: $\quad \mathbf{M}^{(n)}(i_n, :) \mathrel{+}= x_{i_1,\ldots,i_n,\ldots,i_N} \left[*_{(j \neq n)} \left(\mathbf{U}^{(j)}(i_j, :)\right)\right]$

Figure 2. Performing MTTKRP for a sparse tensor $\boldsymbol{\mathcal{X}}$ in a mode n.

The standard algorithm for computing the CP decomposition is CP-ALS, which establishes a good trade-off between convergence rate (number of iterations) and iteration cost [7]. It is an iterative algorithm, shown in Figure 1, that progressively updates the factors $\mathbf{U}^{(n)}$ in an alternating fashion starting from an initial guess, which can be randomly set or obtained from the truncated SVD of the matricizations of $\boldsymbol{\mathcal{X}}$ [7]. CP-ALS iterates until it can no longer improve the solution, or it reaches the allowed maximum number of iterations. Each iteration of CP-ALS consists of N *subiterations*, and the nth subiteration updates $\mathbf{U}^{(n)}$ using $\boldsymbol{\mathcal{X}}$ as well as other factor matrices.

Computing the matrix $\mathbf{M}^{(n)} \in \mathbb{R}^{I_n \times R}$ at Line 5 of Figure 1 is the sole part involving the tensor $\boldsymbol{\mathcal{X}}$, and is the most expensive computational step of the CP-ALS algorithm, both for sparse and dense tensors. The operation $\mathbf{M}^{(n)} \leftarrow \boldsymbol{\mathcal{X}}_{(n)} (\odot_{(i \neq n)} \mathbf{U}^{(i)})$ is called the *matricized tensor-times Khatri-Rao product* (MTTKRP). The Khatri-Rao product of the involved $\mathbf{U}^{(n)}$s defines a matrix of size $(\prod_{i \neq n} I_i) \times R$ according to Equation (1), which can get very costly in terms of computational and memory requirements when I_i or N is large—which is indeed the case for many real-world sparse tensors. To alleviate this, various methods are proposed in the literature that enable performing MTTKRP without forming the Khatri-Rao product, which is made possible by exploiting the sparsity of the tensor. In Figure 2, we provide a such algorithm for computing MTTKRP in mode n using a sparse tensor $\boldsymbol{\mathcal{X}}$. This computation amounts to performing Hadamard product of $N - 1$ vectors of size R and accumulating these products for each nonzero element of the tensor. The overall cost of the algorithm is $O(\text{nnz}(\boldsymbol{\mathcal{X}}) N R)$ as each nonzero induces an Hadamard product of $N - 1$ row vectors followed by an addition of a row vector of size R.

III. Related Work

Computing the CP decomposition of a big sparse tensor can get very costly; for this reason, computing it efficiently

by exploiting the sparsity of the tensor has attracted significant recent interest in the scientific community in various computational settings. In [20], Bader and Kolda demonstrate efficiently carrying out many sparse tensor operations, including MTTKRP, in MATLAB. Their approach for MTTKRP translates into Figure 2. In [21], Chi and Kolda present an alternative Alternating Poisson Regression (CP-APR) algorithm for computing the CP decomposition of large scale sparse datasets. GIGATENSOR [17] provides a distributed memory parallelization of CP-ALS using the Map-Reduce framework. DFACTO [18] is a distributed memory parallel implementation (C++, MPI) that formulates MTTKRP as a series of sparse matrix-vector multiplication. SPLATT [11]–[13] is an efficient parallelization of MTTKRP and CP-ALS both in shared [11], [12] and distributed memory environments [13] using OpenMP and MPI, and is implemented in C. DFACTO and SPLATT aim to reduce the cost of Figure 2 with the help of the following observation. For a 3-dimensional tensor $\boldsymbol{\mathcal{X}}$ having nonzeros x_{i,j,k_1} and x_{i,j,k_2}, one can first compute $x_{i,j,k_1}\mathbf{U}^{(3)}(k_1,:)+x_{i,j,k_2}\mathbf{U}^{(3)}(k_2,:)$, then multiply this result with $\mathbf{U}^{(2)}(j,:)$ to obtain the final contribution to $\mathbf{M}^{(1)}(i,:)$. In other words, multiplying all tensor nonzeros with a matrix in one dimension before proceeding to the other can potentially reduce the number of Hadamard multiplications performed, and this reduction is possible owing to overlapping nonzero indices. However, the worst-case complexity of this approach stays the same, i.e., $O(\text{nnz}(\boldsymbol{\mathcal{X}})NR)$. HYPERTENSOR [10] is an efficient sparse tensor factorization library implemented in C++ using OpenMP and MPI for parallelism. It employs a data structure called dimension tree to reduce the MTTKRP cost by storing and reusing some intermediate results for MTTKRP in the course of CP-ALS iterations. MTTKRP is expressed as a series of R tensor-times-vector multiply operations, whose amortized cost translates to $O(\text{nnz}(\boldsymbol{\mathcal{X}})\log NR)$ for each tensor dimension, which provides significant performance gains as the tensor's dimensionality increases.

IV. PARALLEL CP DECOMPOSITION USING NONZERO FACTORS

Here, we first introduce our approach for efficiently performing MTTKRP with the assumption that factor matrices do not involve zeros or very small entries. When a such entry $\mathbf{U}^{(n)}(i,j)$ with $|\mathbf{U}^{(n)}(i,j)|<\epsilon$ is encountered in the course of CP-ALS, we slightly "perturb" the factor matrix by replacing it with $sign(\mathbf{U}^{(n)}(i,j))\epsilon$ where $sign(x)$ equals to -1 if x is negative, and 1 otherwise. In practice such a perturbation is expected to have a negligible impact on the quality of solution for a sufficiently small ϵ. Next, we introduce a shared memory parallelization of this scheme, and argue how to establish load balance among processes. Finally, we discuss optimization strategies for better parallel performance on a NUMA architecture.

A. Computing CP decomposition with nonzero factors

The cost of the algorithm in Figure 1 is dominated by the MTTKRP step at Line 5 that involves the multiplication of the elements of the sparse tensor $\boldsymbol{\mathcal{X}}$ with the rows of $N-1$ factor matrices at each subiteration. This amounts to performing $N-1$ vector Hadamard products and a vector addition for each nonzero element of the tensor, as pointed out at Line 3 of Figure 2. Here, we present a new technique for efficiently performing this costly step with the nonzero factor matrix assumption. In this case, for each nonzero $x_{i_1,\ldots,i_N}\in\boldsymbol{\mathcal{X}}$, instead of performing the Hadamard product of $N-1$ row vectors, one can precompute a vector $\mathbf{z}_{i_1,\ldots,i_N}\in\mathbb{R}^R$ as $\mathbf{z}_{i_1,\ldots,i_N}=*_{(n\in\mathbb{N}_N)}\mathbf{U}^{(n)}(i_n,:)$, then perform the MTTKRP update due to this nonzero as $\mathbf{M}^{(n)}(i_n,:)\mathrel{+}=\mathbf{z}_{i_1,\ldots,i_N}\oslash\mathbf{U}^{(n)}(i_n,:)$. A similar idea is also employed in the CP-APR algorithm for handling sparse tensors [21]. Here, the cost per nonzero reduces to a single Hadamard division, which can always be performed since $\mathbf{U}^{(n)}(i_n,j)\neq 0$. Once new $\mathbf{U}^{(n)}$ is computed using $\mathbf{M}^{(n)}$ at Line 7, $\mathbf{z}_{i_1,\ldots,i_N}$ needs to be updated accordingly with the new $\mathbf{U}^{(n)}(i_n,:)$. This can be done by dividing it with the old value of $\mathbf{U}^{(n)}(i_n,:)$, then multiplying by its new value, which amounts to a single Hadamard multiplication and division. This way, instead of $N-1$ vector Hadamard products, we perform a Hadamard multiplication and two Hadamard divisions for each tensor element, which effectively reduces the cost of MTTKRP to $O(\text{nnz}(\boldsymbol{\mathcal{X}})R)$. In contrast, SPLATT and DFACTO require up to $N-1$ vector Hadamard products per tensor nonzero, yielding the worst-case complexity $O(\text{nnz}(\boldsymbol{\mathcal{X}})NR)$, and HYPERTENSOR takes $O(\text{nnz}(\boldsymbol{\mathcal{X}})\log NR)$ time. Therefore, our approach provides significant computational gains over all these methods particularly as $\boldsymbol{\mathcal{X}}$ gets higher dimensional. In doing so, we use only $\mathbf{U}^{(n)}$ for executing CP-ALS in dimension n, whereas SPLATT and DFACTO access all $N-1$ factor matrices except $\mathbf{U}^{(n)}$; hence, our method also yields a better memory footprint.

In our method, we need to store the matrix $\mathbf{Z}$ which takes $O(\text{nnz}(\boldsymbol{\mathcal{X}})R)$ space. In contrast, HYPERTENSOR uses $O(\log N)$ buffers each taking up to $O(\text{nnz}(\boldsymbol{\mathcal{X}})R)$ space. SPLATT uses only $O(PR)$ memory for intermediate results for an execution using P threads, yet it incurs the highest computational cost.

B. Parallelization

Performing MTTKRP in a mode n amounts to performing a divide-add operation for each vector $z_{i_1,\ldots,i_N}$ to eventually form the matrix row $\mathbf{M}^{(n)}(i_n,:)$. Similarly, after the new $\mathbf{U}^{(n)}(i_n,:)$ is computed, one needs to update the vector $z_{i_1,\ldots,i_N}$ with a Hadamard multiplication and a division. Therefore, all nonzero elements of $\boldsymbol{\mathcal{X}}$ whose nth index equals to i_n contributes a summand to $\mathbf{M}^{(n)}(i_n,:)$, and the corresponding vectors in $\mathbf{Z}$ needs to be updated subsequently using the old and new values of $\mathbf{U}^{(n)}(i_n,:)$. To perform this, for each dimension $n\in\{1,\ldots,N\}$ and for each matrix row $i_n\in I_n$ we compute a *reduction list* of tensor nonzeros whose nth index is i_n, which we denote as $rl^{(n)}(i_n)$. This way, each row $\mathbf{M}^{(n)}(i_n,:)$ can be computed in parallel by performing $|rl^{(n)}(i_n)|$ vector operations. Similarly, once $\mathbf{U}^{(n)}$ is computed, one can process each row i_n in parallel to update the corresponding vectors $z_{i_1,\ldots,i_N}$ for each contributing nonzero $x_{i_1,\ldots,i_N}$.

The parallel algorithm for computing the CP decomposition is shown in Figure 3. In the main subiteration loop, we first compute the MTTKRP result $\mathbf{M}^{(n)}$ in parallel using P processes at Lines 4–9. This step assumes a partition $\mathcal{I}_1^{(n)},\ldots,\mathcal{I}_P^{(n)}$ of row indices $1,\ldots,I_n$. Each process p computes the set $\mathcal{I}_p^{(n)}$ of rows of the matrix $\mathbf{M}^{(n)}$ independently thanks to the precomputed reduction lists. Immediately after the process uses the entry $\mathbf{z}_{i_1,\ldots,i_N}$ in MTTKRP, it divides it by the old value of $\mathbf{U}^{(n)}(i,:)$. Once $\mathbf{M}^{(n)}$ is formed, at Line 10 we compute $\mathbf{H}^{(n)}$ by performing the Hadamard

Input: $\mathcal{X}$: An N-mode tensor, $\mathcal{X} \in \mathbb{R}^{I_1,\ldots,I_N}$
R: The rank of CP decomposition
$\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}$: Initial factor matrices with nonzero entries
Output: $[\![\lambda; \mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$: The rank-R CP decomposition of $\mathcal{X}$
1: Initialize $\mathbf{Z}$ and $\mathbf{W}^{(n)}$ for all $n \in \mathbb{N}_N$.
2: **repeat**
3: **for** $n = 1,\ldots,N$ **do**
4: **parallel for** $p = 1 \ldots P$ **do** ▶ Compute $\mathbf{M}^{(n)}(\mathcal{I}_p^{(n)},:)$
5: **for** $i_n \in \mathcal{I}_p^{(n)}$ **do**
6: $\mathbf{M}^{(n)}(i_n,:) \leftarrow 0$
7: **for** $\mathbf{z}_{i_1,\ldots,i_N} \in rl^{(n)}(i_n)$ **do**
8: $\mathbf{M}^{(n)}(i_n,:) \mathrel{+}= \mathbf{z}_{i_1,\ldots,i_N} / \mathbf{U}^{(n)}(i_n,:)$
9: $\mathbf{z}_{i_1,\ldots,i_N} = \mathbf{z}_{i_1,\ldots,i_N} \oslash \mathbf{U}^{(n)}(i_n,:)$
10: $\mathbf{H}^{(n)} \leftarrow *_{(i \neq n)} \mathbf{W}^{(i)}$ ▶ Matrix Hadamard product
11: $\mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)} \mathbf{H}^{(n)\dagger}$ ▶ Row-parallel GEMM
12: $\lambda \leftarrow$ NONZERO-COLUMN-NORMALIZE$(\mathbf{U}^{(n)})$
13: $\mathbf{W}^{(n)} \leftarrow \mathbf{U}^{(i)^T} \mathbf{U}^{(i)}$ ▶ Row-parallel SYRK
14: **parallel for** $p = 1 \ldots P$ **do** ▶ Update $\mathbf{Z}$
15: **for** $i_n \in \mathcal{I}_p^{(n)}$ **do**
16: **for** $\mathbf{z}_{i_1,\ldots,i_N} \in rl^{(n)}(i_n)$ **do**
17: $\mathbf{z}_{i_1,\ldots,i_N} = \mathbf{z}_{i_1,\ldots,i_N} * \mathbf{U}^{(n)}(i_n,:)$
18: **until** convergence or reaching maximum number of iterations
19: **return** $[\![\lambda; \mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$

Figure 3. Parallel CP-ALS with nonzero factors.

product of $R \times R$ matrices $\mathbf{W}^{(k)}$ for $k \neq n$, whose cost is negligible as R is a small constant in practice. Next, at Line 11 we update the factor $\mathbf{U}^{(n)}$ in a parallel dense matrix multiplication step, in which each process p performs the multiplication $\mathbf{M}^{(n)}(\mathcal{I}_p^{(n)},:)\mathbf{H}^{(n)\dagger}$. Once $\mathbf{U}^{(n)}$ is computed, we swap its small entries with ϵ or $-\epsilon$ and normalize its columns in a combined step at Line 12 in which each process p works on the sub-matrix $\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)$. Using the updated $\mathbf{U}^{(n)}$, we first compute the new $\mathbf{W}^{(n)}$ in another parallel dense matrix multiplication step at line Line 13, where the process p similarly performs $\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)^T\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)$, then multiply the entries of $\mathbf{Z}$ with the corresponding rows of $\mathbf{U}^{(n)}$ using the same parallelization scheme as in Line 4. The initialization of matrices $\mathbf{Z}$ as well as $\mathbf{W}^{(n)}$ at Line 1 are done in parallel similar to the manner of updating these matrices in the iteration loop. At the end of each iteration, one has to check the convergence as well. This computation takes insignificant amount of time [19], hence we skip the details.

Reduction lists for a dimension n can be computed by making two passes over the tensor nonzero indices in nth dimension to form a very efficient compressed data structure consisting of $rl^{(n)}$, which correspond to reduction list pointers, and $rli^{(n)}$, which correspond to the elements in the reduction list, in $O(\text{nnz}(\mathcal{X}))$ time. This yields $O(N\text{nnz}(\mathcal{X}))$ cost for all dimensions, and we can process each dimension in parallel. In contrast, existing methods in the literature require sorting the tensor indices which takes $O(N\text{nnz}(\mathcal{X})\log(\text{nnz}(\mathcal{X})))$ time for SPLATT [12], and $O(N \log N \text{nnz}(\mathcal{X})\log(\text{nnz}(\mathcal{X})))$ for HYPERTENSOR [19]. We show this data structure for a small tensor $\mathcal{X} \in \mathbb{R}^{5\times5\times5\times5}$ in Figure 4, and skip the computational details.

C. Load balancing

For a P-way parallel execution of Figure 3, one needs to partition the row indices $1,\ldots,I_n$ into P sets $\mathcal{I}_1^{(n)},\ldots,\mathcal{I}_P^{(n)}$ for each dimension n. There are two types of computational costs imposed on each process p by a such partition. First,

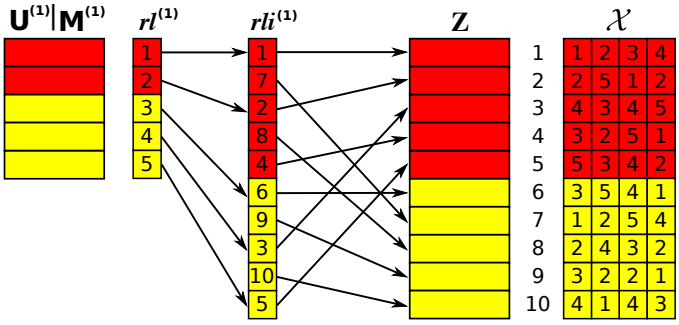

Figure 4. Performing MTTKRP for a 4-dimensional tensor $\mathcal{X} \in \mathbb{R}^{5\times5\times5\times5}$ in the first mode. Red (dark) and yellow (light) colors represent memory regions that are first touched by two different threads.

the process p performs $O(\sum_{i\in\mathcal{I}_p^{(n)}} |rl^{(n)}(i)|)$ vector Hadamard operations at Lines 8, 9 and 17. Second, it performs the multiplication of matrices of size $|\mathcal{I}_p^{(n)}| \times R$ and $R \times R$ at Line 11, and of two matrices of size $|\mathcal{I}_p^{(n)}| \times R$ at Line 13. To balance the first cost pertaining to sparse tensor computations, one has to make sure that the associated cost $\sum_{i\in\mathcal{I}_p^{(n)}} |rl^{(n)}(i)|$ is partitioned equitably to processes. Regarding the second cost for dense matrix operations, each process should have equal number of rows, i.e., $|\mathcal{I}_p^{(n)}|$ should be balanced. Though these rows can be partitioned arbitrarily, in practice we desire to assign a contiguous set of rows to each thread to preserve the data locality. This not only helps improve the memory footprint of the MTTKRP step, but also increases the efficiency of BLAS routines used for dense matrix computations.

We define partitioning problem in this case as follows. For each row i_n, we have an associated pair $(|rl^{(n)}(i_n,:)|, 1)$ of costs that corresponds to sparse tensor and dense matrix computations, respectively. We aim to partition this "chain" of rows into P contiguous parts so that both cost metrics are balanced across processes. The single-cost version of this problem corresponds to the chains-on-chains partitioning (CCP) problem in the literature for which many fast optimal algorithms and effective heuristics exist [22]. We employ CCP algorithms by using only the first cost metric $|rl^{(n)}(i,:)|$ for partitioning, as we observe that in practice, balancing this metric also establishes good row-balance.

D. Optimizations for NUMA scalability

Performing MTTKRP for sparse tensors is an extremely memory bound operation as the tensor is very sparse in general, and the data accesses due to tensor nonzeros lack locality. Therefore, optimizing the memory footprint of the implementation plays a crucial role in obtaining high performance. Particularly on a NUMA architecture, one has to carefully allocate memory pages in NUMA nodes to be able to utilize the available memory bandwidth at maximum, and distribute the memory pages equitably across NUMA nodes. In most systems, this can be ensured by properly using memory first-touch policies after allocation, which in turn yields adequate memory page-to-socket bindings. In our implementation, after the allocation each thread performs a first-touch on the matrix rows as well as the rows of $rl^{(1)}$ and $rli^{(1)}$ that it owns. For the matrix $\mathbf{Z}$, each thread initializes a block of $\text{nnz}(\mathcal{X})/P$ vectors. This way, we not only maximize the NUMA bandwidth utilization, but also aim to reduce the inter-NUMA node memory requests as much as possible. This allocation scheme together with a balanced partitioning is

 ISBN: 978-1-61208-567-8

TABLE I. PER-ITERATION CP-ALS RUNTIME RESULTS (IN SECONDS) FOR SEQUENTIAL, SINGLE-SOCKET (14 THREADS) AND DUAL-SOCKET PARALLEL EXECUTIONS OF ALL METHODS.

Method | Data	ten-4D	ten-8D	ten-16D	ten-32D
$P = 1$				
splatt	175.1	791.2	3766.9	19994.8
hypertensor	**98.3**	306.3	929.3	2873.8
cp-eps	130.3	**284.9**	**586.9**	**1199.4**
$P = 14$				
splatt	15.1	68.4	280.3	1292.2
hypertensor	**11.8**	33.3	88.5	354.5
cp-eps	13.1	**27.5**	**56.12**	**111.31**
$P = 28$				
splatt	11.5	47.2	190.7	683.8
hypertensor	13.0	36.6	69.1	215.0
cp-eps	**8.3**	**17.5**	**36.2**	**65.3**

shown in Figure 4 for two threads.

V. EXPERIMENTS

We compared our algorithm with two state-of-the-art implementations, SPLATT and HYPERTENSOR, and ran them on a workstation having 768GBs of memory and two Intel(R) Xeon(R) E5-2695 CPUs, each having 14 cores running at 2.30GHz as well as L1, L2, and L3 caches of sizes 32K, 256K, and 35M, respectively. We performed two types of experiments to measure the parallel scalability and the accuracy of our method. The first experiment compares the runtime of three methods using synthetically generated high dimensional sparse tensors. The second experiment uses a real-world tensor to compare the quality of approximation of the standard CP-ALS computation with our method. In all experiments, we use $\epsilon = 10^{-6}$ as the threshold parameter.

A. Scalability

We compare the runtime of three methods using 4, 8, 16, and 32-dimensional randomly (uniform) generated tensors of size 10M at each dimension, and having 100M nonzero elements. We employ synthetic data instead of real-world tensors for two reasons. First, with random data we are able to control the dimensionality of the tensor while fixing other tensor parameters, e.g., dimension sizes, the number of nonzeros, and the distribution of nonzero indices; thereby, observe the performance of the algorithms with the increasing tensor dimensionality in a controlled manner. Second, there is a lack of available big high dimensional sparse tensors in the literature in parallel to the lack of efficient computational tools to handle such tensors. We run all three implementations using 1, 14 (single socket), and 28 cores/threads (two sockets) for 20 CP-ALS iterations using $R = 16$, and report the average time spent per iteration in Table I with labels **splatt**, **hypertensor**, and **cp-eps** corresponding to SPLATT, HYPERTENSOR, and our algorithm provided in Figure 3, respectively.

In Table I, we observe that using ten-4D, **hypertensor** runs the fastest using 1 and 14 cores, yet **cp-eps** surpasses **hypertensor** using 28-cores owing to better NUMA optimizations described in Section IV-D. In all other instances, **cp-eps** stays the fastest among all three methods, and the performance gains increase steadily as the tensor's dimensionality grows. Using 4-dimensional to 32-dimensional tensors, we observe that the speedup of **cp-eps** over **splatt** consistently increases from 1.39x to 10.47x using 28 threads, and from 1.34x to 16.67x using a single thread, which conforms with the algorithm complexities provided in Section IV-A. Similarly, the speedup of **cp-eps** over **hypertensor** varies from 1.56x to 3.29x using 28 threads, and from 0.75x to 2.40x using a single thread. Figure 5 demonstrates the speedup results of **cp-eps** and **hypertensor** over **splatt** for ten-32D.

TABLE II. INITIAL SETUP TIME (IN SECONDS) FOR PARALLEL SPARSE CP-ALS.

Method | Data	ten-4D	ten-8D	ten-16D	ten-32D
splatt	60	84	232	617
hypertensor	70	167	418	983
cp-eps	**16**	**18**	**23**	**41**

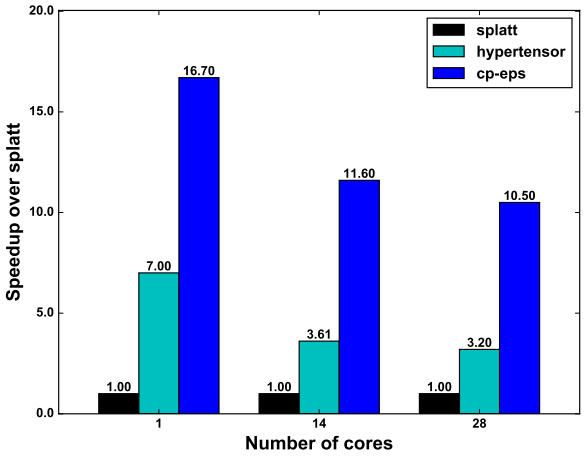

Figure 5. Speedup over **splatt** on ten32D.

In Table II, we compare the time spent on setting up data structures for MTTKRP using 28 threads on all datasets. We observe that **cp-eps** performs the preprocessing step up to 15x faster than **splatt**, and up to 24x faster than **hypertensor**. This significant improvement is possible owing to the smaller asymptotic complexity described in Section IV-B, as well as the parallelization using **cp-eps**.

B. Accuracy

In computing Figure 3, we impose the constraint on factor matrices that they do not contain very small elements, which perturbs the decomposition slightly and can potentially affect the quality of approximation. To assess this, we compare the accuracy of this method with that of the original CP-ALS algorithm. We employ a 3-dimensional real-world tensor obtained from the Never Ending Language Learning (NELL) dataset of the "Read the Web" project [2], which consists of tuples of the form $(entity \times relation \times entity)$ such as ('Chopin', 'plays musical instrument', 'piano'). Nonzeros of this tensor correspond to such "facts" discovered by NELL from the web, while the values of nonzeros are set to the "belief" scores for these facts. The tensor is of size 1.6M×297×338K and has 2M nonzeros. We run **hypertensor** and **cp-eps** 100 times with the rank of approximation $R \in \{25, 50, 75, 100\}$, and compute the geometric mean of approximation quality in each case. In Figure 6 we detail all these results. We observe that both methods produce equally good approximations to the original

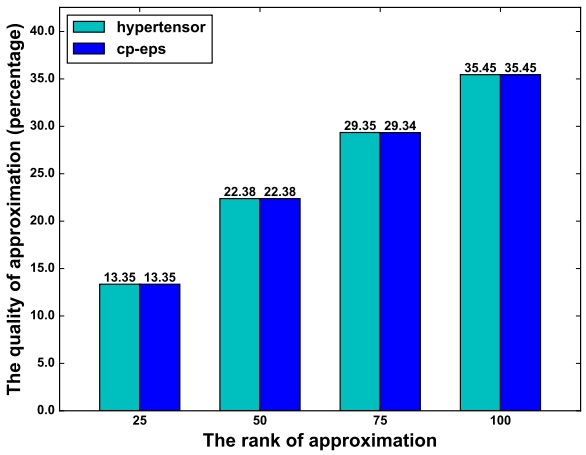

Figure 6. Accuracy comparison of **hypertensor** and **cp-eps** on NELL-2. We show the geometric mean of approximation values of 100 CP-ALS executions with random initial factor matrices using both methods.

tensor up to a small margin of error for $R = 75$ mostly due to randomization in factor matrix initialization. This shows that the nonzero constraint on factor matrices indeed has a negligible effect on the accuracy.

VI. Conclusion

In this work, we propose an efficient parallel algorithm for computing a CP decomposition in which the factor matrices are assumed to not contain any zero elements. This constraint enables a computational scheme that provides $O(\log k)$ and $O(\log N \log k)$ faster preprocessing times for a tensor with k nonzero entries, and performs $O(N)$ and $O(\log N)$ less MTTKRP work over two efficient state-of-the-art implementations **splatt** and **hypertensor** with a negligible effect on the accuracy. We achieve up to 16.7x speedup in sequential and 10.5x speedup in parallel executions over these methods, with up to 24x less data preprocessing time, using up to $O(\log N)$ less memory for storing intermediate computations. With these advancements, our approach renders the analysis of higher order big sparse datasets amenable both in terms of computational and memory requirements for real world applications.

Acknowledgment

Part of this work was supported by GENCI-[TGCC/CINES/IDRIS] (Grant 2016-17 - i2016067501), and was performed using compute resources at ENS de Lyon.

References

[1] T. G. Kolda and B. Bader, "The TOPHITS model for higher-order web link analysis," in Proceedings of Link Analysis, Counterterrorism and Security 2006, 2006, pp. 26–29.

[2] A. Carlson et al., "Toward an architecture for never-ending language learning," in AAAI, vol. 5, 2010, p. 3.

[3] S. Rendle and T. S. Lars, "Pairwise interaction tensor factorization for personalized tag recommendation," in Proceedings of the Third ACM International Conference on Web Search and Data Mining, ser. WSDM '10. New York, NY, USA: ACM, 2010, pp. 81–90.

[4] L. D. Lathauwer and B. D. Moor, "From matrix to tensor: Multilinear algebra and signal processing," in Institute of Mathematics and Its Applications Conference Series, vol. 67, 1998, pp. 1–16.

[5] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear analysis of image ensembles: TensorFaces," in Computer Vision—ECCV 2002. Springer, 2002, pp. 447–460.

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

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

[8] B. W. Bader et al., "Matlab tensor toolbox version 2.6," Available online, Retrieved: March 2017.

[9] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 77:1–77:11.

[10] ——, "High performance parallel algorithms for the Tucker decomposition of sparse tensors," in 45th International Conference on Parallel Processing (ICPP '16), Aug 2016, pp. 103–112.

[11] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in 29th IEEE International Parallel & Distributed Processing Symposium. Hyderabad, India: IEEE Computer Society, May 2015, pp. 61–70.

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

[13] ——, "A medium-grained algorithm for sparse tensor factorization," in 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, 2016, pp. 902–911.

[14] D. J. Carroll and J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition," Psychometrika, vol. 35, no. 3, 1970, pp. 283–319.

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

[16] C. A. Andersson and R. Bro, "The N-way toolbox for MATLAB," Chemometrics and Intelligent Laboratory Systems, vol. 52, no. 1, 2000, pp. 1–4.

[17] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "GigaTensor: Scaling tensor analysis up by 100 times - Algorithms and discoveries," in Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324.

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

[19] O. Kaya and B. Uçar, "Parallel CP decomposition of sparse tensors using dimension trees," Inria - Research Centre Grenoble – Rhône-Alpes, Research Report RR-8976, Nov. 2016.

[20] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," SIAM Journal on Scientific Computing, vol. 30, no. 1, December 2007, pp. 205–231.

[21] E. C. Chi and T. G. Kolda, "On tensors, sparsity, and nonnegative factorizations," SIAM Journal on Matrix Analysis and Applications, vol. 33, no. 4, 2012, pp. 1272–1299.

[22] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," Journal of Parallel and Distributed Computing, vol. 64, no. 8, 2004, pp. 974 – 996.

 ISBN: 978-1-61208-567-8