PArallel RAndomly COMPressed Cubes (PARACOMP):

A Scalable Distributed Architecture for Big Tensor Decomposition

Nicholas D. Sidiropoulos, *Fellow, IEEE*, Evangelos E. Papalexakis, and Christos Faloutsos

This article combines a tutorial on state-of-art tensor decomposition as it relates to big data analytics, with original research on parallel and distributed computation of low-rank decomposition for big tensors, and a concise primer on Hadoop-MapReduce. A novel architecture for parallel and distributed computation of low-rank tensor decomposition that is especially well-suited for big tensors is proposed. The new architecture is based on parallel processing of a set of randomly compressed, reduced-size 'replicas' of the big tensor. Each replica is independently decomposed, and the results are joined via a master linear equation per tensor mode. The approach enables massive parallelism with guaranteed identifiability properties: if the big tensor is indeed of low rank and the system parameters are appropriately chosen, then the rank-one factors of the big tensor will indeed be recovered from the analysis of the reduced-size replicas. Furthermore, the architecture affords memory / storage and complexity gains of order $\frac{IJ}{F}$ for a big tensor of size $I \times J \times K$ of rank F with $F \leq I \leq J \leq K$. No sparsity is required in the tensor or the underlying latent factors, although such sparsity can be exploited to improve memory, storage and computational savings.

I. Introduction

Tensors are data structures indexed by three or more indices, say $(i, j, k, \cdots)$ - a generalization of matrices, which are data structures indexed by two indices, say (r, c) for (row,column). The term *tensor* has a different meaning in Physics, however it has been widely adopted across various disciplines in recent years to refer to what was previously known as a *multi-way array*. Matrices are two-way arrays, and there are three- and higher-way (or *higher-order*) tensors.

Tensor algebra has many similarities but also many striking differences with matrix algebra - e.g., determining tensor rank is NP-hard, and low-rank tensor factorization is unique under mild conditions. Tensor factorizations have already found many applications in signal processing (speech, audio, communications, radar, signal intelligence, machine learning) and well beyond. For example, tensor factorization can be used to blindly separate unknown mixtures of speech signals in reverberant environments [2]; 'untangle' audio sources in the spectrogram domain [3]; unravel mixtures of code-division communication signals without knowledge of their spreading codes [4]; localize emitters in radar and communication applications [5]; detect cliques in social networks [6]; and analyze fluorescence spectroscopy data [7], to name a few - see also [8] for additional machine learning applications.

Tensors are becoming increasingly important, especially for analyzing big data, and tensors easily turn really big, e.g., $1000 \times 1000 \times 1000 = 1$ billion entries. Memory issues related to tensor computations with large but sparse tensors have been considered in [9], [10], and incorporated in the sparse tensor toolbox http://www.sandia.gov/~tgkolda/TensorToolbox. The main idea in those references is to avoid intermediate product 'explosion' when computing sequential tensor - matrix ('mode') products, but the assumption is that the entire tensor fits in memory (in 'coordinate-wise' representation), and the mode products expand (as opposed to reduce) the size of the 'core' array that they are multiplied with. Adaptive tensor decomposition algorithms for cases where the data is serially acquired (or 'elongated') along one mode have been developed in [11], but these assume that the other two modes are relatively modest in size. More recently, a divide-and-conquer approach for decomposing big tensors has been proposed in [12]. The idea of [12] is to break the data in smaller 'boxes' which can be factored independently, and the results subsequently concatenated using an iterative process. This assumes that each smaller box admits a unique factorization (which cannot be guaranteed from 'global' uniqueness conditions alone), requires reconciling the different column permutations and scalings of the different blocks, and entails significant communication and signaling overhead.

All of the aforementioned techniques require that the full data be stored in (possibly distributed) memory. Realizing that this is a show-stopper for truly big tensors, [6] proposed a random sampling approach, wherein judiciously sampled 'significant' parts of the tensor are independently analyzed, and a common piece of data is used to anchor the different permutations and scalings. The downside of [6] is that it only works for sparse tensors, and it offers no identifiability guarantees - although it usually works well for sparse tensors. A different approach was taken in [13], which proposed *randomly* compressing a big tensor down to a far smaller one. Assuming that the big tensor admits a low-rank decomposition with sparse latent factors, such a random compression guarantees identifiability of the low-rank decomposition of the big tensor from the low-rank decomposition of the small tensor. This result can be viewed as a generalization of compressed sensing ideas from the linear to the multi-linear case. Still, this approach works only when the latent low-rank factors of the big tensor are known to be sparse - and this is often not the case.

This article considers appropriate compression strategies for big (sparse or dense) tensors that admit a low-rank decomposition / approximation, whose latent factors need not be sparse. Latent sparsity is usually associated with membership problems such as clustering and co-clustering [14]. A novel architecture for parallel and distributed computation of low-rank tensor decomposition that is especially well-suited for big tensors is proposed. The new architecture is based on parallel processing of a set of randomly compressed, reduced-size 'replicas' or the big tensor. Each replica is independently decomposed, and the results are joined via a master linear equation per tensor mode. The approach enables massive parallelism with guaranteed identifiability properties: if the big tensor is indeed of low rank and the system parameters are appropriately chosen, then the rank-one factors of the big tensor will indeed be recovered from the analysis of the reduced-size replicas. Furthermore, the architecture affords memory / storage and complexity gains of order $\frac{IJ}{F}$ for a big

N.D. Sidiropoulos (contact author) is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN USA. E-mail nikos@umn.edu, phone: (612) 625-1242, Fax: (612) 625-4583. Supported by NSF IIS-1247632.

E.E. Papalexakis and C. Faloutsos are with the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA. E-mail: (epapalex,christos)@cs.cmu.edu. Supported by NSF IIS-1247489.

Submitted to *IEEE Signal Processing Magazine*, Special Issue on Signal Processing for Big Data, Aug. 30, 2013 (white paper), Jan. 14, 2014 (full version); revised May 1, 2014; accepted June 3, 2014. Preliminary conference version of part of these results appears in *Proc. ICASSP 2014* [1].

tensor of size $I \times J \times K$ of rank F with $F \leq I \leq J \leq K$. No sparsity is required in the tensor or the underlying latent factors, although such sparsity can be exploited to improve memory, storage and computational savings.

This article combines i) a short tutorial on state-of-art tensor decomposition as it relates to big data analytics; ii) novel research results on tensor compression and parallel and distributed tensor decomposition; and iii) a concise primer on Hadoop-MapReduce, starting from a toy signal processing problem, and going up to sketching a Hadoop implementation of a proposed algorithm for 'tensor decomposition in the cloud'. The combination is timely and well-motivated given the emerging interest in (and relative scarcity of literature on) signal processing for big data analytics, and in porting / translating and developing new signal processing algorithms for cloud computing platforms.

Notation: A scalar is denoted by an italic letter, e.g. a. A column vector is denoted by a bold lowercase letter, e.g. $\mathbf{a}$ whose i-th entry is $\mathbf{a}(i)$. A matrix is denoted by a bold uppercase letter, e.g., $\mathbf{A}$ with (i,j)-th entry $\mathbf{A}(i,j)$; $\mathbf{A}(:,j)$ ($\mathbf{A}(i,:)$) denotes the j-th column (resp. i-th row) of $\mathbf{A}$. A tensor (three-way array) is denoted by an underlined bold uppercase letter, e.g., $\underline{\mathbf{X}}$, with (i,j,k)-th entry $\underline{\mathbf{X}}(i,j,k)$. $\underline{\mathbf{X}}(:,:,k)$ denotes the k-th frontal $I \times J$ matrix 'slab' of $\underline{\mathbf{X}}$, and similarly for the slabs along the other two modes. Vector, matrix and three-way array size parameters (mode lengths) are denoted by uppercase letters, e.g. I. $\circ$ stands for the vector outer product; i.e., for two vectors $\mathbf{a}$ ($I \times 1$) and $\mathbf{b}$ ($J \times 1$), $\mathbf{a} \circ \mathbf{b}$ is an $I \times J$ matrix with (i,j)-th element $\mathbf{a}(i)\mathbf{b}(j)$; i.e., $\mathbf{a} \circ \mathbf{b} = \mathbf{a}\mathbf{b}^T$. For three vectors, $\mathbf{a}$ ($I \times 1$), $\mathbf{b}$ ($J \times 1$), $\mathbf{c}$ ($K \times 1$), $\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$ is an $I \times J \times K$ three-way array with (i,j,k)-th element $\mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k)$. The vec$(\cdot)$ operator stacks the columns of its matrix argument in one tall column; $\otimes$ stands for the Kronecker product; $\odot$ stands for the Khatri-Rao (column-wise Kronecker) product: given $\mathbf{A}$ ($I \times F$) and $\mathbf{B}$ ($J \times F$), $\mathbf{A} \odot \mathbf{B}$ is the $JI \times F$ matrix

$$\mathbf{A} \odot \mathbf{B} = \left[\mathbf{A}(:,1) \otimes \mathbf{B}(:,1) \cdots \mathbf{A}(:,F) \otimes \mathbf{B}(:,F)\right]$$

For a square matrix $\mathbf{S}$, Tr$(\mathbf{S})$ denotes its trace, i.e., the sum of elements on its main diagonal. $||\mathbf{x}||_2^2$ is the Euclidean norm squared, and $||\mathbf{A}||_F^2$, $||\underline{\mathbf{X}}||_F^2$ the Frobenious norm squared - the sum of squares of all elements of the given vector, matrix, or tensor.

II. Tensor Decomposition Preliminaries

There is no comprehensive tutorial on tensor decompositions and applications from a signal processing point of view as of this writing, albeit there are several signal processing papers dealing with topics in tensor decomposition that have significant tutorial value. The concise introduction in [15] is still useful, albeit outdated. An upcoming *Signal Processing Magazine* tutorial [8] covers the basic concepts and models well, and touches upon numerous applications. We also refer the reader to [16] and [17], [18] for gentle introductions to tensor decompositions and applications from the viewpoint of computational linear algebra, chemistry, and the social sciences, respectively. Due to space limitations, here we only review essential concepts and results that directly relate to the core of our article.

Rank decomposition: The rank of an $I \times J$ matrix $\mathbf{X}$ is the smallest number of rank-one matrices (vector outer products of the form $\mathbf{a} \circ \mathbf{b}$) needed to synthesize $\mathbf{X}$ as

$$\mathbf{X} = \sum_{f=1}^{F} \mathbf{a}_f \circ \mathbf{b}_f = \mathbf{A}\mathbf{B}^T,$$

where $\mathbf{A} := [\mathbf{a}_1, \cdots, \mathbf{a}_F]$, and $\mathbf{B} := [\mathbf{b}_1, \cdots, \mathbf{b}_F]$. This relation can be expressed element-wise as

$$\mathbf{X}(i,j) = \sum_{f=1}^{F} \mathbf{a}_f(i)\mathbf{b}_f(j).$$

The rank of an $I \times J \times K$ three-way array $\underline{\mathbf{X}}$ is the smallest number of outer products needed to synthesize $\underline{\mathbf{X}}$ as

$$\underline{\mathbf{X}} = \sum_{f=1}^{F} \mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f.$$

This relation can be expressed element-wise as

$$\underline{\mathbf{X}}(i,j,k) = \sum_{f=1}^{F} \mathbf{a}_f(i)\mathbf{b}_f(j)\mathbf{c}_f(k).$$

In the sequel we will assume that F is minimal, i.e., $F = \text{rank}(\underline{\mathbf{X}})$, unless otherwise noted. The tensor $\underline{\mathbf{X}}$ comprises K 'frontal' slabs of size $I \times J$; denote them $\{\mathbf{X}_k\}_{k=1}^K$, with $\mathbf{X}_k := \underline{\mathbf{X}}(:,:,k)$. Re-arranging the elements of $\underline{\mathbf{X}}$ in a tall matrix $\mathbf{X} := [\text{vec}(\mathbf{X}_1), \cdots, \text{vec}(\mathbf{X}_K)]$, it can be shown that

$$\mathbf{X} = (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T \Longleftrightarrow \mathbf{x} := \text{vec}(\mathbf{X}) = (\mathbf{C} \odot \mathbf{B} \odot \mathbf{A})\,\mathbf{1},$$

where, $\mathbf{A}$, $\mathbf{B}$ are as defined for the matrix case, $\mathbf{C} := [\mathbf{c}_1, \cdots, \mathbf{c}_F]$, $\mathbf{1}$ is a vector of all 1's, and we have used the vectorization property of the Khatri-Rao product $\text{vec}(\mathbf{A}\mathbf{D}(\mathbf{d})\mathbf{B}^T) = (\mathbf{B} \odot \mathbf{A})\,\mathbf{d}$, where $\mathbf{D}(\mathbf{d})$ is a diagonal matrix with the vector $\mathbf{d}$ as its diagonal.

CANDECOMP-PARAFAC: The above rank decomposition model for tensors is known as *parallel factor analysis* (PARAFAC) [19], [20] or *canonical decomposition* (CANDECOMP) [21], or CP (and CPD) for CANDECOMP-PARAFAC (Decomposition), or *canonical polyadic decomposition* (CPD, again). CP is usually fitted using an alternating least squares procedure based on the model equation $\mathbf{X} = (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T$. In practice we will have $\mathbf{X} \approx (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T$, due to measurement noise and other imperfections, or simply because we wish to approximate a higher-rank model with a lower-rank one. Fixing $\mathbf{A}$ and $\mathbf{B}$, we solve

$$\min_{\mathbf{C}} ||\mathbf{X} - (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T||_F^2,$$

which is a linear least squares problem. We can bring any of the matrix factors to the right by reshuffling the data, yielding corresponding conditional updates for $\mathbf{A}$ and $\mathbf{B}$. We can revisit each matrix in a circular fashion until convergence of the cost function, and this is the most commonly adopted aproach to fitting the CP model, in good part because of its conceptual and programming simplicity, plus the ease with which one can incorporate additional constraints on the columns of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ [7].

Tucker3: CP is in a way the most basic tensor model, because of its direct relationship to tensor rank and the concept of rank decomposition; but other algebraic tensor models exist, and the most notable one is known as *Tucker3*. Like CP, Tucker3 is a sum of outer products model, involving outer products of columns of three matrices, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$. Unlike CP however, which restricts interactions to corresponding columns (so that the first column of $\mathbf{A}$ only appears in one outer product involving the first column of $\mathbf{B}$ and the first column of $\mathbf{C}$), Tucker3 includes all outer products of every column of $\mathbf{A}$ with every column of $\mathbf{B}$ and every column of $\mathbf{C}$. Each such outer product is further weighted by the corresponding entry of a so-called *core tensor*, whose dimensions are equal to the number of columns of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$.

Consider again the $I \times J \times K$ three-way array $\underline{\mathbf{X}}$ comprising K matrix slabs $\{\mathbf{X}_k\}_{k=1}^K$, arranged into the tall matrix $\mathbf{X} := [\text{vec}(\mathbf{X}_1), \cdots, \text{vec}(\mathbf{X}_K)]$. The Tucker3 model can be written in matrix form as

$$\mathbf{X} \approx (\mathbf{B} \otimes \mathbf{A})\mathbf{G}\mathbf{C}^T,$$

where $\mathbf{G}$ is the core tensor in matrix form, and $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ can be assumed orthogonal without loss of generality, because linear transformations of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ can be absorbed in $\mathbf{G}$. The non-zero elements of the core tensor determine the interactions between columns of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$. The associated model-fitting problem is

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{G}} ||\mathbf{X} - (\mathbf{B} \otimes \mathbf{A})\mathbf{G}\mathbf{C}^T||_F^2,$$

which is usually solved using an alternating least squares procedure. The Tucker3 model can be fully vectorized as $\text{vec}(\mathbf{X}) \approx (\mathbf{C} \otimes \mathbf{B} \otimes \mathbf{A})\,\text{vec}(\mathbf{G})$.

Identifiability: The distinguishing feature of the CP model is its essential uniqueness: under certain conditions, $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ can be identified from $\mathbf{X}$ up to a common permutation and scaling / counter-scaling of columns [19]–[26]. In contrast, Tucker3 is highly non-unique; the inclusion of all possible outer products of columns of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ results in over-parametrization that renders it unidentifiable in most cases of practical interest. Still, Tucker3 is useful as an exploratory tool and for data compression / interpolation; we will return to this shortly.

Consider an $I \times J \times K$ tensor $\underline{\mathbf{X}}$ of rank F. In vectorized form, it can be written as the $IJK \times 1$ vector $\mathbf{x} = (\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})\,\mathbf{1}$, for some $\mathbf{A}$ $(I \times F)$, $\mathbf{B}$ $(J \times F)$, and $\mathbf{C}$ $(K \times F)$ - a CP model of size $I \times J \times K$ and order F parameterized by $(\mathbf{A}, \mathbf{B}, \mathbf{C})$. (Notice the slight abuse of notation: we switched from $\mathbf{x} = (\mathbf{C} \odot \mathbf{B} \odot \mathbf{A})\,\mathbf{1}$ to $\mathbf{x} = (\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})\,\mathbf{1}$. The two are related via a row permutation, or by switching the roles of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$.) The *Kruskal-rank* of $\mathbf{A}$, denoted $k_{\mathbf{A}}$, is the maximum k such that *any* k columns of $\mathbf{A}$ are linearly independent ($k_{\mathbf{A}} \leq r_{\mathbf{A}} := \text{rank}(\mathbf{A})$).

Theorem 1: [22] Given $\underline{\mathbf{X}}$ ($\Leftrightarrow \mathbf{x}$), $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ are unique up to a common column permutation and scaling (e.g., scaling the first column of $\mathbf{A}$ and counter-scaling the first column of $\mathbf{B}$ and/or $\mathbf{C}$, so long as their product remains the same), provided that $k_{\mathbf{A}} + k_{\mathbf{B}} + k_{\mathbf{C}} \geq 2F + 2$. An equivalent and perhaps more intuitive way to express this is that the outer products $\mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f$ (i.e., the rank-one factors of $\underline{\mathbf{X}}$) are unique.

Note that we can always reshuffle the order of these rank-one factors (e.g., swap $\mathbf{a}_1 \circ \mathbf{b}_1 \circ \mathbf{c}_1$ and $\mathbf{a}_2 \circ \mathbf{b}_2 \circ \mathbf{c}_2$) without changing their sum $\underline{\mathbf{X}} = \sum_{f=1}^{F} \mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f$, but this is a trivial and inherently unresolvable ambiguity that we will ignore in the sequel. Theorem 1 is Kruskal's celebrated uniqueness result [22], see also follow-up work in [23]–[25]. Kruskal's result applies to given $(\mathbf{A}, \mathbf{B}, \mathbf{C})$, i.e., it can establish uniqueness of a given decomposition. Recently, more relaxed uniqueness conditions have been obtained, which only depend on the size and rank of the tensor - albeit they cover *almost* all tensors of the given size and rank, i.e., except for a set of measure zero. Two such conditions are summarized next.

Theorem 2: [27] (see also [24]) Consider an $I \times J \times K$ tensor $\underline{\mathbf{X}}$ of rank F. If

$$r_{\mathbf{C}} = F \text{ (which implies } K \geq F)$$

and

$$I(I-1)J(J-1) \geq 2F(F-1),$$

then the rank-one factors of $\underline{\mathbf{X}}$ are almost surely unique.

Theorem 3: [26] Consider an $I \times J \times K$ tensor $\underline{\mathbf{X}}$ of rank F. Order the dimensions so that $I \leq J \leq K$. Let i be maximal such that $2^i \leq I$, and likewise j maximal such that $2^j \leq J$. If $F \leq 2^{i+j-2}$, then the rank-one factors of $\underline{\mathbf{X}}$ are almost surely unique. For I, J powers of 2, the condition simplifies to $F \leq \frac{IJ}{4}$. More generally, the condition implies that if $F \leq \frac{(I+1)(J+1)}{16}$, then $\underline{\mathbf{X}}$ has a unique decomposition almost surely.

Before we proceed to discuss big data and cloud computing aspects of tensor decomposition, we state two lemmas from [13] which we will need in the sequel.

Lemma 1: [13] Consider $\tilde{\mathbf{A}} := \mathbf{U}^T\mathbf{A}$, where $\mathbf{A}$ is $I \times F$, and let the $I \times L$ matrix $\mathbf{U}$ be randomly drawn from an absolutely continuous distribution (e.g., multivariate Gaussian with a non-singular covariance matrix). Then $k_{\tilde{\mathbf{A}}} = \min(L, k_{\mathbf{A}})$ almost surely (with probability 1).

Lemma 2: [13] Consider $\tilde{\mathbf{A}} = \mathbf{U}^T\mathbf{A}$, where $\mathbf{A}$ $(I \times F)$ is deterministic, tall/square $(I \geq F)$ and full column rank $r_{\mathbf{A}} = F$, and the elements of $\mathbf{U}$ $(I \times L)$ are i.i.d. Gaussian zero mean, unit variance random variables. Then the distribution of $\tilde{\mathbf{A}}$ is absolutely continuous (nonsingular multivariate Gaussian).

III. Tensor Compression

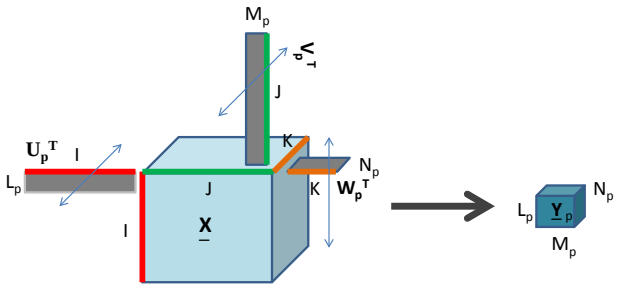

Fig. 1. Schematic illustration of tensor compression: going from an $I \times J \times K$ tensor $\underline{\mathbf{X}}$ to a much smaller $L_p \times M_p \times N_p$ tensor $\underline{\mathbf{Y}}_p$ via multiplying (every slab of) $\underline{\mathbf{X}}$ from the I-mode with $\mathbf{U}_p^T$, from the J-mode with $\mathbf{V}_p^T$, and from the K-mode with $\mathbf{W}_p^T$, where $\mathbf{U}_p$ is $I \times L_p$, $\mathbf{V}_p$ is $J \times M_p$, and $\mathbf{W}_p$ is $K \times N_p$.

When dealing with big tensors $\underline{\mathbf{X}}$ that do not fit in main memory, a reasonable idea is to try to compress $\underline{\mathbf{X}}$ to a much smaller tensor that somehow captures most of the systematic variation in $\underline{\mathbf{X}}$. The commonly used compression method is to fit a low-dimensional orthogonal Tucker3 model (with low mode-ranks) [17], [18], then regress the data onto the fitted mode-bases. This idea has been exploited in existing CP model-fitting software, such as COMFAC [28], as a useful quick-and-dirty way to initialize alternating least squares computations in the uncompressed domain, thus accelerating convergence. A key issue with Tucker3 compression of big tensors is that it requires computing singular value decompositions of the various matrix unfoldings of the full data, in an alternating fashion. This is a serious bottleneck for big data. Another issue is that Tucker3 compression is lossy, and it cannot guarantee that identifiability properties will be preserved. Finally, fitting a CP model to the compressed data can only yield an approximate model for the original uncompressed data, and eventually decompression and iterations with the full data are required to obtain fine estimates.

Consider compressing $\mathbf{x}$ into $\mathbf{y} = \mathbf{S}\mathbf{x}$, where $\mathbf{S}$ is $d \times IJK$, $d \ll IJK$. Sidiropoulos & Kyrillidis [13] proposed using a specially structured compression matrix $\mathbf{S} = \mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T$, which corresponds to multiplying (every slab of) $\underline{\mathbf{X}}$ from the I-mode with $\mathbf{U}^T$, from the J-mode with $\mathbf{V}^T$, and from the K-mode with $\mathbf{W}^T$, where $\mathbf{U}$ is $I \times L$, $\mathbf{V}$ is $J \times M$, and $\mathbf{W}$ is $K \times N$, with $L \leq I$, $M \leq J$, $N \leq K$ and $LMN \ll IJK$; see Fig. 1. Such an $\mathbf{S}$ corresponds to compressing each mode individually, which is often natural, and the associated multiplications can be efficiently implemented - see call-outs #1 and #2. Due to a fortuitous property of the Kronecker product

[29],

$$\left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)(\mathbf{A} \odot \mathbf{B} \odot \mathbf{C}) = \left((\mathbf{U}^T\mathbf{A}) \odot (\mathbf{V}^T\mathbf{B}) \odot (\mathbf{W}^T\mathbf{C})\right),$$

from which it follows that

$$\mathbf{y} = \left((\mathbf{U}^T\mathbf{A}) \odot (\mathbf{V}^T\mathbf{B}) \odot (\mathbf{W}^T\mathbf{C})\right)\mathbf{1} = \left(\tilde{\mathbf{A}} \odot \tilde{\mathbf{B}} \odot \tilde{\mathbf{C}}\right)\mathbf{1}.$$

i.e., the compressed data follow a CP model of size $L \times M \times N$ and order F parameterized by $(\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}})$, with $\tilde{\mathbf{A}} := \mathbf{U}^T\mathbf{A}$, $\tilde{\mathbf{B}} := \mathbf{V}^T\mathbf{B}$, $\tilde{\mathbf{C}} := \mathbf{W}^T\mathbf{C}$.

This is nice to know, but we are really - naturally - interested in obtaining answers to the following two questions:

1) Under what conditions on $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are $(\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}})$ identifiable from $\mathbf{y}$?
2) Under what conditions, if any, are $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ identifiable from $(\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}})$?

We start by answering the first question in the next section.

Call-out #1: Complexity of multi-way compression? Multiplying a dense $L \times I$ matrix $\mathbf{U}^T$ with a dense vector $\mathbf{a}$ to compute $\mathbf{U}^T\mathbf{a}$ has complexity LI. Taking the product of $\mathbf{U}^T$ and the first $I \times J$ frontal slab $\underline{\mathbf{X}}(:,:,1)$ of the $I \times J \times K$ tensor $\underline{\mathbf{X}}$ has complexity LIJ. Pre-multiplying from the left all frontal slabs of $\underline{\mathbf{X}}$ by $\mathbf{U}^T$ (computing a *mode product*) therefore requires $LIJK$ operations, when all operands are dense. Multi-way compression as in Fig. 1 comprises three mode products, suggesting a complexity of $LIJK + MLJK + NLMK$, if the first mode is compressed first, followed by the second, and then the third mode. Notice that the order in which the mode products are computed affects the complexity of the overall operation; but order-wise, this is $O(\min(L, M, N)IJK)$. Also notice that if I, J, K are of the same order, and so are L, M, N, then the overall complexity is $O(LI^3)$. If $\mathbf{a}$ is sparse with NZ($\underline{\mathbf{a}}$) nonzero elements, we can compute $\mathbf{U}^T\mathbf{a}$ as a weighted sum of the columns of $\mathbf{U}^T$ corresponding to the nonzero elements of $\mathbf{a}$. This reduces matrix-vector multiplication complexity to $L\text{NZ}(\underline{\mathbf{a}})$. It easily follows that if $\underline{\mathbf{X}}$ has NZ $(\underline{\mathbf{X}})$ nonzero elements, the complexity of pre-multiplying from the left all frontal slabs of $\underline{\mathbf{X}}$ by $\mathbf{U}^T$ can be reduced to $L\text{NZ}(\underline{\mathbf{X}})$. The problem is that, after computing the first mode product, the resulting tensor will be dense! - hence subsequent mode products cannot exploit sparsity to reduce complexity. Note that, in addition to computational complexity, memory or secondary storage to save the intermediate results of the computation becomes an issue, even if the original tensor $\underline{\mathbf{X}}$ is sparse.

IV. Stepping-stone results

The following result is a direct consequence of Lemma 1 and Kruskal's uniqueness condition in Theorem 1.

Theorem 4: Let $\mathbf{x} = (\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})\mathbf{1} \in \mathbb{R}^{IJK}$, where $\mathbf{A}$ is $I \times F$, $\mathbf{B}$ is $J \times F$, $\mathbf{C}$ is $K \times F$, and consider compressing it to $\mathbf{y} = \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{x} = \left((\mathbf{U}^T\mathbf{A}) \odot (\mathbf{V}^T\mathbf{B}) \odot (\mathbf{W}^T\mathbf{C})\right)\mathbf{1} = \left(\tilde{\mathbf{A}} \odot \tilde{\mathbf{B}} \odot \tilde{\mathbf{C}}\right)\mathbf{1} \in \mathbb{R}^{LMN}$, where the mode-compression matrices $\mathbf{U}$ $(I \times L, L \leq I)$, $\mathbf{V}$ $(J \times M, M \leq J)$, and $\mathbf{W}$ $(K \times N, N \leq K)$ are independently drawn from an absolutely continuous distribution. If

$$\min(L, k_{\mathbf{A}}) + \min(M, k_{\mathbf{B}}) + \min(N, k_{\mathbf{C}}) \geq 2F + 2,$$

then $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}$ are almost surely identifiable from the compressed data $\mathbf{y}$ up to a common column permutation and scaling.

More relaxed conditions for identifiability of $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}$ can be derived from Lemma 2, and Theorems 2 and 3.

Theorem 5: For $\mathbf{x}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{U}, \mathbf{V}, \mathbf{W}$, and $\mathbf{y}$ as in Theorem 4, if $F \leq \min(I, J, K)$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are all full column rank (F), $N \geq F$, and

$$L(L-1)M(M-1) \geq 2F(F-1),$$

then $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}$ are almost surely identifiable from the compressed data $\mathbf{y}$ up to a common column permutation and scaling.

Call-out #2: Complexity of multi-way compression - redux. In scalar form, the (ℓ, m, n)-th element of the tensor $\underline{\mathbf{Y}}$ after multi-way compression can be written as

$$\underline{\mathbf{Y}}(\ell, m, n) = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} \mathbf{U}(i, \ell)\mathbf{V}(j, m)\mathbf{W}(k, n)\underline{\mathbf{X}}(i, j, k)$$

Claim 1: Suppose that $\underline{\mathbf{X}}$ is sparse, with NZ $(\underline{\mathbf{X}})$ nonzero elements, and suppose that it is stored as a serial list with entries formatted as $[i, j, k, v]$, where v is the nonzero value at tensor position (i, j, k). Suppose that the list is indexed by an integer index s, i.e., $[i(s), j(s), k(s), v(s)]$ is the record corresponding to the s-th entry of the list. Then the following simple algorithm will compute the multi-way compressed tensor $\underline{\mathbf{Y}}$ in only $LMN\text{NZ}(\underline{\mathbf{X}})$ operations, requiring only LMN cells of memory to store the result, and $IL + JM + KN$ cells of memory to store the matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$.

Algorithm 1: Efficient multi-way compression pseudo-code:

```
Y=zeros(L,M,N);
for s=1:NZX,
 for ell=1:L,
  for m=1:M,
   for n=1:N,
    Y(ell,m,n) = Y(ell,m,n)+ U(i(s),ell)*V(j(s),m)*W(k(s),n)*v(s);
   end
  end
 end
end
```

Notice that, even if $\underline{\mathbf{X}}$ is dense (i.e., NZ $(\underline{\mathbf{X}}) = IJK$), the above algorithm only needs to read each element of $\underline{\mathbf{X}}$ once, so complexity will be $LMNIJK$ but memory will still be very modest: only LMN cells of memory to store the result, and $IL + JM + KN$ cells of memory to store the matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$. Contrast this to the 'naive' way of serially computing the mode products, whose complexity order is $O(\min(L, M, N)IJK)$ but whose memory requirements are huge for dense $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$, due to intermediate result explosion - even for sparse $\underline{\mathbf{X}}$. We see a clear complexity - memory trade-off between the two approaches for dense data, but Algorithm 1 is a clear winner for sparse data, because sparsity is lost after the first mode product. Notice that the above algorithm can be fully parallelized in several ways - by splitting the list of nonzero elements across cores or processors (paying in terms of auxiliary memory replications to store partial results for $\underline{\mathbf{Y}}$ and the matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ locally at each processor), or by splitting the (ℓ, m, n) loops - at the cost of replicating the data list. As a final word, the memory access pattern (whether we read and write consecutive memory elements in blocks, or make wide 'strides') is the performance-limiting factor for truly big data, and the above algorithm makes strides in reading elements of $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$, and writing elements of $\underline{\mathbf{Y}}$. There are ways to reduce these strides, at the cost of requiring more memory and more floating point operations.

Remark 1: $F \leq \min(I, J, K) \Rightarrow$ full column rank $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ almost surely, i.e., tall matrices are full column rank except for a set of measure zero. In other words, if $F \leq \min(I, J, K)$ and $\mathbf{A}$,

$\mathbf{B}$, $\mathbf{C}$ are themselves considered to be independently drawn from an absolutely continuous distribution with respect to the Lebesgue measure in $\mathbb{R}^{IF}$, $\mathbb{R}^{JF}$, and $\mathbb{R}^{KF}$, respectively, then they will all be full column rank with probability 1.

Theorem 6: For $\mathbf{x}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{U}, \mathbf{V}, \mathbf{W}$, and $\mathbf{y}$ as in Theorem 4, if $F \leq \min(I, J, K)$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are all full column rank (F), $L \leq M \leq N$, and

$$(L+1)(M+1) \geq 16F,$$

then $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}$ are almost surely identifiable from the compressed data $\mathbf{y}$ up to a common column permutation and scaling.

V. Main results

Theorems 4, 5, 6 can establish uniqueness of $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \tilde{\mathbf{C}}$, but we are ultimately interested in $\mathbf{A}, \mathbf{B}, \mathbf{C}$. We know that $\tilde{\mathbf{A}} = \mathbf{U}^T\mathbf{A}$, and we know $\mathbf{U}^T$, but, unfortunately, it is a fat matrix that cannot be inverted. In order to uniquely recover $\mathbf{A}$, one needs additional structural constraints. Sidiropoulos & Kyrillidis [13] proposed exploiting column-wise sparsity in $\mathbf{A}$ (and likewise $\mathbf{B}, \mathbf{C}$), which is often plausible in practice[1]. Sparsity is a powerful constraint, but it is not always valid (or a sparsifying basis may be unknown). For this reason, we propose here a different solution, based on creating and factoring a number of randomly reduced 'replicas' of the full data.

Consider spawning P randomly compressed reduced-size 'replicas' $\left\{\underline{\mathbf{Y}}_p\right\}_{p=1}^{P}$ of the tensor $\underline{\mathbf{X}}$, where $\underline{\mathbf{Y}}_p$ is created using mode compression matrices $(\mathbf{U}_p, \mathbf{V}_p, \mathbf{W}_p)$, see Fig. 2. Assume that identifiability

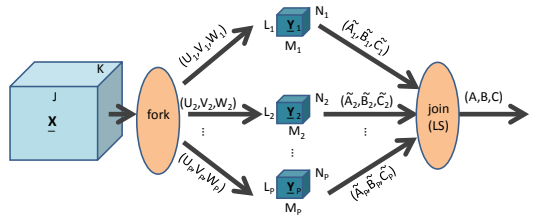

Fig. 2. Schematic illustration of the PARACOMP fork-join architecture. The fork step creates a set of P randomly compressed reduced-size 'replicas' $\left\{\underline{\mathbf{Y}}_p\right\}_{p=1}^{P}$. Each $\underline{\mathbf{Y}}_p$ is obtained by applying $(\mathbf{U}_p, \mathbf{V}_p, \mathbf{W}_p)$ to $\underline{\mathbf{X}}$, as detailed in Fig. 1. Each $\underline{\mathbf{Y}}_p$ is then independently factored (all P threads can be executed in parallel). The join step collects the estimated mode loading sub-matrices $\left(\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p\right)$ from the P threads, and, after anchoring all to a common permutation and scaling, solves a master linear least squares problem per mode to estimate the full mode loading matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C})$.

conditions per Theorem 5 or Theorem 6 hold, so that $\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p$ are almost surely identifiable (up to permutation and scaling of columns) from $\underline{\mathbf{Y}}_p$. Then, upon factoring $\underline{\mathbf{Y}}_p$ into F rank-one components, we obtain

$$\tilde{\mathbf{A}}_p = \mathbf{U}_p^T\mathbf{A}\mathbf{\Pi}_p\mathbf{\Lambda}_p, \tag{1}$$

where $\mathbf{\Pi}_p$ is a permutation matrix, and $\mathbf{\Lambda}_p$ is a diagonal scaling matrix with nonzero elements on its diagonal. Assume that the first two columns of each $\mathbf{U}_p$ (rows of $\mathbf{U}_p^T$) are common, and let $\bar{\mathbf{U}}$ denote this common part, and $\bar{\mathbf{A}}_p$ denote the first two rows of $\tilde{\mathbf{A}}_p$. We therefore have

$$\bar{\mathbf{A}}_p = \bar{\mathbf{U}}^T\mathbf{A}\mathbf{\Pi}_p\mathbf{\Lambda}_p.$$

Dividing each column of $\bar{\mathbf{A}}_p$ by the element of maximum modulus in that column, and denoting the resulting $2 \times F$ matrix $\hat{\mathbf{A}}_p$, we obtain

$$\hat{\mathbf{A}}_p = \bar{\mathbf{U}}^T\mathbf{A}\mathbf{\Lambda}\mathbf{\Pi}_p.$$

[1] $\mathbf{A}$ need only be sparse with respect to (when expressed in) a suitable basis, provided the sparsifying basis is known *a priori*.

Notice that $\mathbf{\Lambda}$ does not affect the ratio of elements in each 2×1 column. If these ratios are distinct (which is guaranteed almost surely if $\bar{\mathbf{U}}$ and $\mathbf{A}$ are independently drawn from absolutely continuous distributions), then the different permutations can be matched by sorting the ratios of the two coordinates of each 2×1 column of $\hat{\mathbf{A}}_p$.

In practice using a few more 'anchor' rows will improve the permutation-matching performance, and is recommended in difficult cases with high noise variance. When S anchor rows are used, the optimal permutation matching problem can be cast as

$$\min_{\mathbf{\Pi}} ||\hat{\mathbf{A}}_1 - \hat{\mathbf{A}}_p\mathbf{\Pi}||_F^2,$$

where optimization is over the set of permutation matrices. This may appear to be a hard combinatorial problem at first sight; but it is not. Using

$$||\hat{\mathbf{A}}_1 - \hat{\mathbf{A}}_p\mathbf{\Pi}||_F^2 = \text{Tr}\left((\hat{\mathbf{A}}_1 - \hat{\mathbf{A}}_p\mathbf{\Pi})^T(\hat{\mathbf{A}}_1 - \hat{\mathbf{A}}_p\mathbf{\Pi})\right) =$$

$$||\hat{\mathbf{A}}_1||_F^2 + ||\hat{\mathbf{A}}_p\mathbf{\Pi}||_F^2 - 2\text{Tr}(\hat{\mathbf{A}}_1^T\hat{\mathbf{A}}_p\mathbf{\Pi}) =$$

$$||\hat{\mathbf{A}}_1||_F^2 + ||\hat{\mathbf{A}}_p||_F^2 - 2\text{Tr}(\hat{\mathbf{A}}_1^T\hat{\mathbf{A}}_p\mathbf{\Pi}).$$

It follows that we may instead

$$\max_{\mathbf{\Pi}} \text{Tr}(\hat{\mathbf{A}}_1^T\hat{\mathbf{A}}_p\mathbf{\Pi}),$$

over the set of permutation matrices. This is what is known as the *Linear Assignment Problem* (LAP), and it can be efficiently solved using the *Hungarian Algorithm*.

After this column permutation-matching process, we go back to (1) and permute its columns to obtain $\breve{\mathbf{A}}_p$ satisfying

$$\breve{\mathbf{A}}_p = \mathbf{U}_p^T\mathbf{A}\mathbf{\Pi}\mathbf{\Lambda}_p.$$

It remains to get rid of $\mathbf{\Lambda}_p$. For this, we normalize each column by dividing it with its norm. This finally yields

$$\breve{\mathbf{A}}_p = \mathbf{U}_p^T\mathbf{A}\mathbf{\Pi}\mathbf{\Lambda}.$$

For recovery of $\mathbf{A}$ up to permutation and scaling of its columns, we then require that the matrix of the linear system

$$\begin{bmatrix} \breve{\mathbf{A}}_1 \\ \vdots \\ \breve{\mathbf{A}}_P \end{bmatrix} = \begin{bmatrix} \mathbf{U}_1^T \\ \vdots \\ \mathbf{U}_P^T \end{bmatrix} \mathbf{A}\mathbf{\Pi}\mathbf{\Lambda} \tag{2}$$

be full column rank. This implies that

$$2 + \sum_{p=1}^{P}(L_p - 2) \geq I$$

i.e.,

$$\sum_{p=1}^{P}(L_p - 2) \geq I - 2.$$

Note that every sub-matrix contains the two anchor rows which are common, and duplicate rows clearly do not increase the rank. Also note that once the dimensionality requirement is met, the matrix will be full rank with probability 1, because its non-redundant entries are drawn from a jointly continuous distribution (by design).

Assuming $L_p = L$, $\forall p \in \{1, \cdots, P\}$ for simplicity (and symmetry of computational load), we obtain $P(L-2) \geq I - 2$, or, in terms of the number of threads

$$P \geq \frac{I-2}{L-2}.$$

Likewise, from the corresponding full column rank requirements for the other two modes, we obtain

$$P \geq \frac{J}{M}, \text{ and } P \geq \frac{K}{N}.$$

Notice that we do not subtract 2 from numerator and denominator for the other two modes, because the permutation of columns of $\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p$ is *common* - so it is enough to figure it out from one mode, and apply it to other modes as well. In short,

$$P \geq \max\left(\frac{I-2}{L-2}, \frac{J}{M}, \frac{K}{N}\right)$$

Call-out #3: The color of compressed noise. Consider a noisy tensor $\underline{\mathbf{Y}} = \underline{\mathbf{X}} + \underline{\mathbf{Z}}$, where $\underline{\mathbf{Z}}$ denotes zero-mean additive white noise. In vectorized form, $\mathbf{y} = \mathbf{x} + \mathbf{z}$, with $\mathbf{y} := \text{vec}\,(\underline{\mathbf{Y}})$, $\mathbf{x} := \text{vec}\,(\underline{\mathbf{X}})$, $\mathbf{z} := \text{vec}\,(\underline{\mathbf{Z}})$. After multi-way compression, one obtains the reduced-size tensor $\underline{\mathbf{Y}}_c$, whose vectorized representation $\mathbf{y}_c := \text{vec}\,(\underline{\mathbf{Y}}_c) = \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{y} = \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{x} + \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{z}$. Let $\mathbf{z}_c := \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{z}$. Clearly, $E[\mathbf{z}_c] = 0$, and

$$E\left[\mathbf{z}_c\mathbf{z}_c^T\right] = E\left[\left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\mathbf{z}\mathbf{z}^T\left(\mathbf{U} \otimes \mathbf{V} \otimes \mathbf{W}\right)\right]$$
$$= \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)E\left[\mathbf{z}\mathbf{z}^T\right]\left(\mathbf{U} \otimes \mathbf{V} \otimes \mathbf{W}\right) =$$
$$= \sigma^2\left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)\left(\mathbf{U} \otimes \mathbf{V} \otimes \mathbf{W}\right) =$$
$$= \sigma^2\left(\left(\mathbf{U}^T\mathbf{U}\right) \otimes \left(\mathbf{V}^T\mathbf{V}\right) \otimes \left(\mathbf{W}^T\mathbf{W}\right)\right),$$

where we have used two properties of the Kronecker product: transposition

$$\left(\mathbf{A} \otimes \mathbf{B}\right)^T = \mathbf{A}^T \otimes \mathbf{B}^T,$$

and the 'mixed product rule' [29]

$$\left(\mathbf{A} \otimes \mathbf{B}\right)\left(\mathbf{C} \otimes \mathbf{D}\right) = \left(\mathbf{AC} \otimes \mathbf{BD}\right).$$

We see that, if $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are orthonormal, then the noise in the compressed domain is white. Note that, for large I and $\mathbf{U}$ drawn from a zero-mean unit-variance uncorrelated distribution, $\mathbf{U}^T\mathbf{U} \approx \mathbf{I}$ by the law of large numbers. Furthermore, even if $\mathbf{z}$ is not Gaussian, $\mathbf{z}_c$ will be approximately Gaussian for large IJK, by the Central Limit Theorem. From these, it follows that least-squares fitting is approximately optimal in the compressed domain, even if it is not so in the uncompressed domain. Compression thus makes least-squares fitting 'universal'!

Remark 2: Note that if, say, $\mathbf{A}$ can be identified and it is full column rank, then $\mathbf{B}$ and $\mathbf{C}$ can be identified by solving a linear least squares problem - but this requires access to the full big tensor data. In the same vein, if $\mathbf{A}$ and $\mathbf{B}$ are identified, then $\mathbf{C}$ can be identified from the full big tensor data even if $\mathbf{A}$ and $\mathbf{B}$ are not full column rank individually - it is enough that $\mathbf{A} \odot \mathbf{B}$ is full column rank, which is necessary for identifiability of $\mathbf{C}$ even from the big tensor, hence not restrictive. PARACOMP-based identification, on the other hand, only requires access to the factors derived from the small replicas. This is clearly advantageous, as the raw big tensor data can be discarded after compression, and there is no need for retrieving huge amounts of data from cloud storage.

One can pick the mode used to figure out the permutation ambiguity, leading to the symmetrized condition $P \geq \min\{P_1, P_2, P_3\}$ with

$$P_1 = \max\left(\frac{I-2}{L-2}, \frac{J}{M}, \frac{K}{N}\right)$$

$$P_2 = \max\left(\frac{I}{L}, \frac{J-2}{M-2}, \frac{K}{N}\right)$$

$$P_3 = \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right)$$

If the compression ratios in the different modes are similar, it makes sense to use the longest mode for this purpose; if this is the last mode, then

$$P \geq \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right)$$

We have thus established the following result.

Theorem 7: In reference to Fig. 2, assume $\mathbf{x} := \text{vec}\,(\underline{\mathbf{X}}) = (\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})\,\mathbf{1} \in \mathbb{R}^{IJK}$, where $\mathbf{A}$ is $I \times F$, $\mathbf{B}$ is $J \times F$, $\mathbf{C}$ is $K \times F$ (i.e., the rank of $\underline{\mathbf{X}}$ is at most F). Assume that $F \leq I \leq J \leq K$, and $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are all full column rank (F). Further assume that $L_p = L$, $M_p = M$, $N_p = N$, $\forall p \in \{1, \cdots, P\}$, $L \leq M \leq N$, $(L+1)(M+1) \geq 16F$, the elements of $\{\mathbf{U}_p\}_{p=1}^P$ are drawn from a jointly continuous distribution, and likewise for $\{\mathbf{V}_p\}_{p=1}^P$, while each $\mathbf{W}_p$ contains two common anchor columns, and the elements of $\{\mathbf{W}_p\}_{p=1}^P$ (except for the repeated anchors, obviously) are drawn from a jointly continuous distribution. Then the data for each thread $\mathbf{y}_p := \text{vec}\left(\underline{\mathbf{Y}}_p\right)$ can be uniquely factored, i.e., $\left(\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p\right)$ is unique up to column permutation and scaling. If, in addition to the above, we also have $P \geq \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right)$ parallel threads, then $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ are almost surely identifiable from the thread outputs $\left\{\left(\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p\right)\right\}_{p=1}^P$ up to a common column permutation and scaling.

The above result is indicative of a family of results that can be derived, using different CP identifiability results. Its significance may not be immediately obvious, so it is worth elaborating further at this point. On one hand, Theorem 7 shows that fully parallel computation of the big tensor decomposition is possible – the first such result, to the best of our knowledge, that *guarantees* identifiability of the big tensor decomposition from the intermediate small tensor decompositions, without placing stringent additional constraints. On the other hand, the conditions appear convoluted, and the memory / storage and computational savings, if any, are not necessarily easy to see. The following claim nails down the take-home message.

Claim 2: Under the conditions of Theorem 7, if $\frac{K-2}{N-2} = \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right)$, then the memory / storage and computational complexity savings afforded by the architecture shown in Fig. 2 relative to brute-force computation are of order $\frac{IJ}{F}$.

Proof 1: Each thread must store LMN elements, and we have $P = \frac{K-2}{N-2}$ threads in all, leading to a total data size of order LMK versus IJK, so the ratio is $\frac{IJ}{LM}$. The condition $(L+1)(M+1) \geq 16F$ only requires LM to be of order F, hence the total compression ratio can be as high as $O\left(\frac{IJ}{F}\right)$. Turning to overall computational complexity, note that optimal low-rank tensor factorization is NP-hard, even in the rank-one case. Practical tensor factorization algorithms, however, typically have complexity $O(IJKF)$ (per iteration, and overall if a bound on the maximum number of iterations is enforced). It follows that the practical complexity order for factoring out the P parallel threads is $O(PLMNF)$ versus $O(IJKF)$ for the brute-force computation. Taking into account the lower bound on P, the ratio is again of order $\frac{IJ}{LM}$, and since the condition $(L+1)(M+1) \geq 16F$ only requires LM to be of order F, the total computational complexity gain can be as high as $O\left(\frac{IJ}{F}\right)$.

Remark 3: The complexity of solving the master linear equation (2) in the final merging step for $\mathbf{A}$ may be a source of concern - especially because it hasn't been accounted for in the overall complexity calculation. Solving a linear system of order of I equations

in I unknowns generally requires $O(I^3)$ computations; but closer scrutiny of the system matrix in (2) reveals interesting features. If all elements of the compression matrices $\{\mathbf{U}_p\}$ (except for the common anchors) are independent and identically distributed with zero mean and unit variance, then, after removing the redundant rows, the system matrix in (2) will have approximately orthogonal columns for large I. This implies that its left pseudo-inverse will simply be its transpose, approximately. This reduces the complexity of solving (2) to I^2F. If higher accuracy is required, the pseudo-inverse may be computed off-line and stored. It is also important to stress that (2) is only solved once for each mode at the end of the overall process, whereas tensor decomposition typically takes many iterations. In short, the constants are such that we need to worry more about the compression (fork) and decomposition stages, rather than the final join stage.

Theorem 7 assumes $F \leq \min(I, J, K)$ in order to ensure (via Lemma 2) absolute continuity of the compressed factor matrices, which is needed to invoke almost sure uniqueness per [26]. Cases where $F > \min(I, J, K)$ can be treated using Kruskal's condition for unique decomposition of each compressed replica

$$\min(L, k_{\mathbf{A}}) + \min(M, k_{\mathbf{B}}) + \min(N, k_{\mathbf{C}}) \geq 2F + 2.$$

It can be shown that $k_{\mathbf{A}} = \min(I, F)$ for almost every $\mathbf{A}$ (except for a set of measure zero in $\mathbb{R}^{IF}$); and likewise $k_{\mathbf{B}} = \min(J, F)$, and $k_{\mathbf{C}} = \min(K, F)$, for almost every $\mathbf{B}$ and $\mathbf{C}$. This simplifies[2] the above condition to

$$\min(L, I, F) + \min(M, J, F) + \min(N, K, F) \geq 2F + 2.$$

Assume $I \geq F$, $J \geq F$, but $K < F$, and pick $L = M = F$, and $N = 3$. Then the condition further reduces to

$$2F + \min(3, K) \geq 2F + 2,$$

which is satisfied for any $K \geq 2$ (i.e., for any tensor). We also need

$$P \geq \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right),$$

which in this case ($N = 3$) reduces to

$$P \geq \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right).$$

When $\frac{I}{L} = \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right)$, then there are $\frac{I}{L}$ parallel threads of size $LMN = 3F^2$ each, for total cloud storage $3IF$, i.e., order IF; hence the overall compression ratio (taking all replicas into account) is of order $\frac{IJK}{IF} = \frac{JK}{F}$. The ratio of overall complexity orders is also $\frac{IJKF}{IF^2} = \frac{JK}{F}$. This is the same type of result as the one we derived for the case $F \leq \min(I, J, K)$. On the other hand, when $K - 2 = \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right)$, there are $K-2$ parallel threads of size $LMN = 3F^2$ each, for total cloud storage $3F^2(K-2)$, i.e., order KF^2; hence the overall compression ratio is $\frac{IJK}{KF^2} = \frac{IJ}{F^2}$, and the ratio of overall complexity orders is also $\frac{IJKF}{KF^3} = \frac{IJ}{F^2}$. We see that there is a penalty factor F relative to the case $F \leq \min(I, J, K)$; this is likely an artifact of the method of proof, which we hope to improve in future work. We summarize the result in the following theorem.

Theorem 8: In reference to Fig. 2, assume $\mathbf{x} := \text{vec}\left(\underline{\mathbf{X}}\right) = (\mathbf{A} \odot \mathbf{B} \odot \mathbf{C})\,\mathbf{1} \in \mathbb{R}^{IJK}$, where $\mathbf{A}$ is $I \times F$, $\mathbf{B}$ is $J \times F$, $\mathbf{C}$ is $K \times F$ (i.e., the rank of $\underline{\mathbf{X}}$ is at most F). Assume that $I \geq F$, $J \geq F$ (K can be $< F$), and pick $L_p = L$, $M_p = M$, $N_p = N$, $\forall p \in \{1, \cdots, P\}$, with $L = M = F$, and $N = 3$. The compression matrices are chosen as in Theorem 7. If $P \geq \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right)$, then $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ is identifiable from $\left\{\left(\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p\right)\right\}_{p=1}^{P}$, for almost every $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ and almost every set of compression matrices. When $\frac{I}{L} = \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right)$, the total storage and complexity gains are of order $\frac{JK}{F}$; whereas if $K-2 = \max\left(\frac{I}{L}, \frac{J}{M}, K-2\right)$, the total storage and complexity gains are of order $\frac{IJ}{F^2}$.

[2]Meaning: if the simplified condition holds, then CP decomposition of each reduced replica is unique for almost every $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ and almost every set of compression matrices $(\mathbf{U}, \mathbf{V}, \mathbf{W})$.

A. Latent sparsity

If latent sparsity is present, we can exploit it to reduce P. Assume that every column of $\mathbf{A}$ ($\mathbf{B}$, $\mathbf{C}$) has at most n_a (resp. n_b, n_c) nonzero elements. A column of $\mathbf{A}$ can be uniquely recovered from only $2n_a$ incoherent linear equations [30]. Therefore, we may replace the condition

$$P \geq \max\left(\frac{I}{L}, \frac{J}{M}, \frac{K-2}{N-2}\right),$$

with

$$P \geq \max\left(\frac{2n_a}{L}, \frac{2n_b}{M}, \frac{2n_c-2}{N-2}\right). \tag{3}$$

Assuming

$$\frac{2n_c-2}{N-2} = \max\left(\frac{2n_a}{L}, \frac{2n_b}{M}, \frac{2n_c-2}{N-2}\right),$$

it is easy to see that the total cloud storage and complexity gains are of order $\frac{IJ}{F}\frac{K}{n_c}$ - improved by a factor of $\frac{K}{n_c}$. It is interesting to compare this result with the one in Sidiropoulos & Kyrillidis [13], which corresponds to using $P = 1$ in our present context. Notice that (3) implies $L \geq \frac{2n_a}{P}$, $M \geq \frac{2n_b}{P}$, $N - 2 \geq \frac{2n_c-2}{P} \Rightarrow N \geq \frac{2n_c}{P} + 2(1-\frac{1}{P}) \Rightarrow N \geq \frac{2n_c}{P}$. Substituting $P = 1$ we obtain $L \geq 2n_a$, $M \geq 2n_b$, $N \geq 2n_c$, which is exactly the condition required in [13]. We see that PARACOMP subsumes [13], offering greater flexibility in terms of choosing P to reduce the size of replicas for easier in-memory processing, at the cost of an additional merging step at the end. Also note that PARACOMP is applicable in the case of dense latent factors, whereas [13] is not.

Remark 4: In practice we will use a higher P, i.e.,

$$P \geq \max\left(\frac{\mu n_a}{L}, \frac{\mu n_b}{M}, \frac{\mu n_c-2}{N-2}\right),$$

with $\mu \in \{3, 4, 5\}$ instead of 2, and an ℓ_1 sparse under-determined linear equations solver for the final merging step for $\mathbf{A}$. This will increase complexity from $O(I^2F)$ to $O(I^{3.5}F)$, and the constants are such that the difference is significant. This is the price paid for the reduced memory and intermediate complexity benefits afforded by latent sparsity.

VI. MapReduce implementation

With the proliferation of large collections of data, as well as big clusters of (usually commodity) computers that were largely under-utilized, arose the need for a unified framework of scalable distributed computation in the cloud. In [31], Dean *et al.* from Google introduced such a framework, called *MapReduce*. MapReduce provides a very versatile level of programming abstraction: it conceals all its inner workings from the programmer, and simply requires the implementation of two functions: `Map` and `Reduce`.

Call-out #4: Is component ordering preserved after compression? Consider randomly compressing a rank-one tensor $\underline{\mathbf{X}} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$, written in vectorized form as $\mathbf{x} = \mathbf{a} \otimes \mathbf{b} \otimes \mathbf{c}$ (recall that the Kronecker product $\otimes$ and the Khatri-Rao product $\odot$ coincide when all arguments involved are vectors). The compressed tensor is $\tilde{\underline{\mathbf{X}}}$, in vectorized form

$$\tilde{\mathbf{x}} = \left(\mathbf{U}^T \otimes \mathbf{V}^T \otimes \mathbf{W}^T\right)(\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{c}) =$$

$$(\mathbf{U}^T\mathbf{a}) \otimes (\mathbf{V}^T\mathbf{b}) \otimes (\mathbf{W}^T\mathbf{c}),$$

using the mixed product rule [29]. It follows $||\tilde{\mathbf{x}}||_2^2 = \tilde{\mathbf{x}}^T\tilde{\mathbf{x}} =$

$$\left((\mathbf{a}^T\mathbf{U}) \otimes (\mathbf{b}^T\mathbf{V}) \otimes (\mathbf{c}^T\mathbf{W})\right)\left((\mathbf{U}^T\mathbf{a}) \otimes (\mathbf{V}^T\mathbf{b}) \otimes (\mathbf{W}^T\mathbf{c})\right) =$$

$$(\mathbf{a}^T\mathbf{U}\mathbf{U}^T\mathbf{a}) \otimes (\mathbf{b}^T\mathbf{V}\mathbf{V}^T\mathbf{b}) \otimes (\mathbf{c}^T\mathbf{W}\mathbf{W}^T\mathbf{c}) =$$

$$||\mathbf{U}^T\mathbf{a}||_2^2||\mathbf{V}^T\mathbf{b}||_2^2||\mathbf{W}^T\mathbf{c}||_2^2,$$

where we have used the transposition and mixed product rules, and that the Kronecker product of scalars is a plain product. Notice that for our choice of $\mathbf{U}$ (i.i.d. zero-mean Gaussian of variance 1, i.e., `randn(I,L)` in Matlab), $\mathbf{U}^T\mathbf{U} \approx \mathbf{I}_{L\times L}$, but $\mathbf{U}\mathbf{U}^T$ is rank-deficient ($L < I$), thus far from $\mathbf{I}_{I\times I}$. However, considering one generic element of $\mathbf{U}^T\mathbf{a}$, say $\mathbf{u}^T\mathbf{a}$, and its magnitude-square, note that $|\mathbf{u}^T\mathbf{a}|^2 = \mathbf{a}^T\mathbf{u}\mathbf{u}^T\mathbf{a}$, so

$$E[|\mathbf{u}^T\mathbf{a}|^2] = \mathbf{a}^T E[\mathbf{u}\mathbf{u}^T]\mathbf{a} = \mathbf{a}^T\mathbf{a} = ||\mathbf{a}||_2^2.$$

Next, it can be shown that

$$\text{Var}[|\mathbf{u}^T\mathbf{a}|^2] = 2||\mathbf{a}||_2^4.$$

So now, looking at $||\mathbf{U}^T\mathbf{a}||_2^2$,

$$E[||\mathbf{U}^T\mathbf{a}||_2^2] = L||\mathbf{a}||_2^2,$$

and, since the different rows of $\mathbf{U}^T$ are independent, hence variance adds up

$$\text{Var}[||\mathbf{U}^T\mathbf{a}||_2^2] = L2||\mathbf{a}||_2^4.$$

So $||\mathbf{U}^T\mathbf{a}||_2^2$ has mean^2/variance ('SNR') of $\frac{L}{2}$.
Turning to $||\tilde{\mathbf{x}}||_2^2 = ||\mathbf{U}^T\mathbf{a}||_2^2||\mathbf{V}^T\mathbf{b}||_2^2||\mathbf{W}^T\mathbf{c}||_2^2$, it can be shown that it has mean

$$E[||\tilde{\mathbf{x}}||_2^2] = LMN||\mathbf{a}||_2^2||\mathbf{b}||_2^2||\mathbf{c}||_2^2,$$

and mean^2/variance ('SNR')

$$\frac{\left(E[||\tilde{\mathbf{x}}||_2^2]\right)^2}{\text{Var}[||\tilde{\mathbf{x}}||_2^2]} = \frac{L^2M^2N^2}{(L^2+2L)(M^2+2M)(N^2+2N) - L^2M^2N^2}.$$

Assuming without loss of generality that $L \leq M \leq N$, this 'SNR' is of order $\frac{L}{2}$. What this means is that, for moderate L, M, N and beyond, the Frobenious norm of a compressed rank-one tensor component (= Euclidean norm of the corresponding vectorized representation) is approximately proportional to the Frobenious norm of the uncompressed rank-one tensor component of the original tensor. In other words: compression approximately preserves component ordering. This is important because it implies that low-rank least-squares approximation of the compressed tensor approximately corresponds to low-rank least-squares approximation of the big tensor. The result also suggests that it may be possible to match the component permutations across replicas simply by sorting component energies. These are ignored in the permutation-matching procedure discussed in the main text, due to the normalization needed to account for the scaling ambiguity. Including energy in the matching process will enhance robustness to noise. It seems intriguing to try rank ('principal component') deflation in this context, but we will pursue this elsewhere due to space limitations.

The `Map` function runs in parallel on many machines; each instance reads data serially from the Distributed File System[3] (DFS), performs some sort of parsing or computation on that data, and emits a series of (`key,value`) pairs. Consequently, the `Reduce` function runs in parallel on a set of machines, and each instance of `Reduce` receives as input (`key,value`) pairs with the same `key`; it performs some sort of (user defined) aggregation or computation on these `values`, and then emits a series of (`key',value'`) pairs, which are eventually written to DFS. This way, any task that can be expressed as a combination of a `Map` and a `Reduce` function may be run in a distributed fashion on a cluster of computers, on data that is also stored in the cloud, and much bigger than what a typical personal computer can store or process in memory. The MapReduce framework also deals with machine failures (an issue which arises very often in large clusters of computers) in a way that is transparent to the programmer. Among other 'safety measures', MapReduce uses three-way replication of each computation, so that even if one machine fails, there are still two backup machines that are carrying out the same task. This way, the user does not have to deal with the frustrations of machine failures. The original MapReduce implementation is internal to Google; however, there exists a very robust and well-tested open source implementation by Apache, called *Hadoop* [32]. The two primary programming languages that can be used with Hadoop are Java and Python.

Signal processing algorithms are generally not realizable as a single MapReduce task, but it is often possible to break up a given algorithm in parts, each of which may be written as a MapReduce computation. In this way, the overall signal processing algorithm can be implemented as a chain of MapReduce tasks.

The most typical introductory example of a MapReduce task is the `WordCount` application [33], where the goal is to estimate the frequency of occurrence of each word in a corpus. Given that MapReduce was originally developed by Google, a search engine which relies heavily on indexing large collections of text in order to provide fast and accurate search results, the `WordCount` example fits perfectly in the original context. In call-out #5 we instead use a very simple and common signal processing task to illustrate the way MapReduce works: computing the histogram of a big speech/audio, image, or video signal. The particular kind of signal is not important here, but bear in mind that our motivation is to be able to handle big data, distributed over the cloud. In order to simplify exposition, we assume that the signal of interest is integer-valued.

A. Sketch of PARACOMP in MapReduce

We now provide a sketch of an implementation of PARACOMP in MapReduce. As in Figure V, we break the algorithm down to three distinct steps: 1) Compression, 2) Decomposition, and 3) Recovery of factor matrices. Each of the three steps consists of a few MapReduce chain tasks.

Compression: For the compression step, we first need to create P triplets of random compression matrices $\mathbf{U}_p$, $\mathbf{V}_p$, $\mathbf{W}_p$. This may be carried out simply by a mapper that emits p (the 'replica' index) as key, and the dimensions of the matrices as the value. Thus, each reducer is responsible for creating and storing on DFS all three compression matrices. Depending on how large the compression matrices are, instead of assigning a single reducer the burden of creating an entire batch of $\mathbf{U}_p$, $\mathbf{V}_p$, $\mathbf{W}_p$, we may instead choose to assign each reducer to create a single row of

[3]DFS is defined by MapReduce.

each of the matrices. Taking a closer look at Algorithm 1 in Call-out #2, we can devise a MapReduce task for the compression step. Let us assume that the tensor is stored in a text file, in multiple lines (as many as the non-zero values in the tensor), in the form

$$i(s), j(s), k(s), v(s)$$

which is appropriate for sparse tensors. Each mapper reads a segment of that file, processing one line at a time. By inspecting the core equation of Algorithm 1 in Call-out #2

$$\underline{\mathbf{Y}}(\ell, m, n) = \underline{\mathbf{Y}}(\ell, m, n) + \mathbf{U}(i(s), \ell)\mathbf{V}(j(s), m)\mathbf{W}(k(s), n)v(s),$$

we see that for each mapper, it suffices to hold $\mathbf{U}(i(s), :)$, $\mathbf{V}(j(s), :)$, and $\mathbf{W}(k(s), :)$ in memory, so that it calculates the contribution of the current non-zero value of the tensor $v(s)$ to the partial sum that comprises $\underline{\mathbf{Y}}(\ell, m, n)$. Since L, M, N are considerably smaller than I, J, K, we use $O(L + M + N)$ of memory on each mapper. Thus, each mapper emits as key the concatenation of (ℓ, m, n) and as value $\mathbf{U}(i(s), \ell)\mathbf{V}(j(s), m)\mathbf{W}(k(s), n)v(s)$. Finally, each reducer receives all partial values of the sum that builds $\underline{\mathbf{Y}}(\ell, m, n)$ up, sums up all incoming values, and emits a pair with key equal to (ℓ, m, n) and value equal to $\underline{\mathbf{Y}}(\ell, m, n)$, which is eventually written to DFS.

Since we execute multiple repetitions of the compression step, we may concatenate the repetition number p to the key that is emitted by the mapper, as well as the key emitted by the reducer. Thus, at the end, there will be one file containing the non-zero values for each compressed tensor in the form:

$$p, \ell, m, n, \underline{\mathbf{Y}}_p(\ell, m, n)$$

Decomposition: For the decomposition step, we spawn P parallel processes on different machines, each one fitting the CP decomposition to the respective compressed tensor. In order to do that in the MapReduce framework, we may use the `Map` function to feed the appropriate data to each reducer. More specifically, each mapper will read portions of the file created by the compression step, and use as a key the repetition index p, and as value the rest of the row, i.e. $(\ell, m, n, \underline{\mathbf{Y}}_p(\ell, m, n))$. Consequently, P reducers will be spawned, each receiving all the data of a single compressed tensor. We assume that the compressed tensor fits in the main memory of a single machine, therefore each reducer simply stores the incoming values in a three dimensional array, and proceeds with in-memory computation of the CP decomposition. In case the reducers are unable to store the compressed tensor in main memory, there exist methods that fit the CP decomposition on MapReduce [34]. However, solving each one of the parallel decompositions on MapReduce would significantly hurt performance, therefore we should aim for compressed tensors that fit in memory.

Recovery of factor matrices: The final step involves the recovery of the factor matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ from the partial factors as obtained from the parallel decomposition step. Recovery for each factor matrix is achieved by stacking the partial results on top of each other, as well as the compression matrices in a similar fashion, and solving a least squares problem involving these two matrices. The stacking of both partial factors and compression matrices can be done through a simple MapReduce task: each mapper will be emitting (i, f) (i.e., the indices of each matrix coefficient) as key, and the value will be the coefficient of the matrix at (i, f) (denoted by v) and the index p, indicating the replica number. Then, each reducer will emit tuples of the form

$$i', f, v$$

where i' will be the original row index adjusted appropriately using p, in order to account for the stacking.

In order to solve the least squares step, we may use scalable algorithms that implement the Moore-Penrose Pseudoinverse on MapReduce [35]. After pseudoinversion, we need to carry out matrix multiplication, a problem which has also been thoroughly studied for MapReduce [36].

Call-out #5: Solving a toy problem in Hadoop - MapReduce.
Consider a large speech/audio, image, or video signal, stored as a text file, with each line containing a signal value. This file is stored in a distributed fashion, in DFS. In order to compute its histogram, it suffices to use a single MapReduce job.

`Map`: Each mapper gets a portion of the file and reads it line-by-line. For each line-entry, n, the mapper sets `key`$= n$ and `value`$= 1$, and emits a $(n,1)$ pair.

`Reduce`: As mentioned earlier, each instance of a reducer receives all such (`key,value`) pairs that have the same `key`. In this particular case, all instances of number n will be processed by the same reducer, since the `Map` function set `key`$= n$. As a consequence, each reducer has all the information needed in order to calculate the exact count of appearances of a given number n. Thus, each reducer simply calculates the total number of $(n, 1)$ pairs (denoted by f), and emits a single tuple (n, f), which contains the number and its corresponding frequency of occurrence.

Finally, when all reducers have terminated, the output of the above MapReduce task will contain lines in the form: (`number, frequency`).

Even though the above example is very simple, the logic that underlies the transformation of an algorithm into a series of MapReduce tasks is the same: decompose the algorithm into self-contained pieces, find a (`key, value`) representation for the intermediate data of each piece, and finally express this computation as a pair of `Map` and `Reduce` functions.

VII. Illustrative numerical results

Our theorems ensure that PARACOMP works with ideal low- and known-rank tensors, but what if there is measurement noise or other imperfections, or we underestimate the rank? Does the overall approach fall apart in this case? From call-outs #3 and #4, we have good reasons to believe that this is not the case, but one cannot be confident without numerical experiments that corroborate intuition. In this section, we provide indicative results to illustrate what can be expected from PARACOMP and the effect of various parameters on estimation performance.

In all cases considered, $I = J = K = 500$, the noiseless tensor has rank $F = 5$, and is synthesized by randomly and independently drawing $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, each from an i.i.d. zero-mean, unit-variance Gaussian distribution (`randn(500,5)` in Matlab), and then taking their tensor product; i.e., computing the sum of outer products of corresponding columns of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. Gaussian i.i.d. measurement noise is then added to this noiseless tensor to yield the observed tensor to be analyzed. The nominal setup uses $L = M = N = 50$ (so that each replica is 0.1% of the original tensor), and $P = 12$ replicas are created for the analysis (so the overall cloud storage used for all replicas is 1.2% of the space needed to store the original tensor). $S = 3$ common anchor rows (instead of $S = 2$, which is the

minimum possible) are used to fix the permutation and scaling ambiguity. These parameter choices satisfy PARACOMP identifiability conditions without much additional 'slack'. The standard deviation of the measurement noise is nominally set to $\sigma = 0.01$.

Fig. 3 shows the total squared error for estimating $\mathbf{A}$, i.e., $||\mathbf{A} - \hat{\mathbf{A}}||_2^2$, where $\hat{\mathbf{A}}$ denotes the estimate of $\mathbf{A}$ obtained using PARACOMP, as a function of $L = M = N$. The baseline is the total squared error attained by directly fitting the uncompressed $500 \times 500 \times 500$ tensor using a mature tensor decomposition algorithm (COMFAC - available at www.ece.umn.edu/~nikos) - the size of the uncompressed tensor used here makes such direct fitting possible, for comparison purposes. We see that PARACOMP yields respectable accuracy with only 1.2% of the full data, and is just an order of magnitude worse than the baseline algorithm when $L = M = N = 150$, corresponding to 32% of the full data. This is one way we can trade-off memory/storage/computation versus estimation accuracy in the PARACOMP framework: by controlling the size of each replica. Another way to trade-off memory/storage/computation for accuracy is through P. Fig. 4 shows accuracy as a function of the number of replicas (computation threads) P, for fixed $L = M = N = 50$. Finally, Fig. 5 plots accuracy as a function of measurement noise variance σ^2, for $L = M = N = 50$ and $P = 12$.

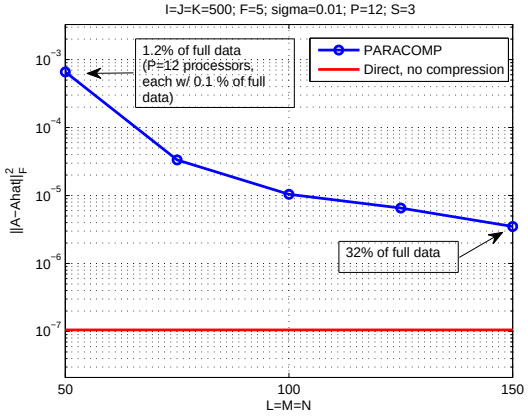

Fig. 3. MSE as a function of $L = M = N$.

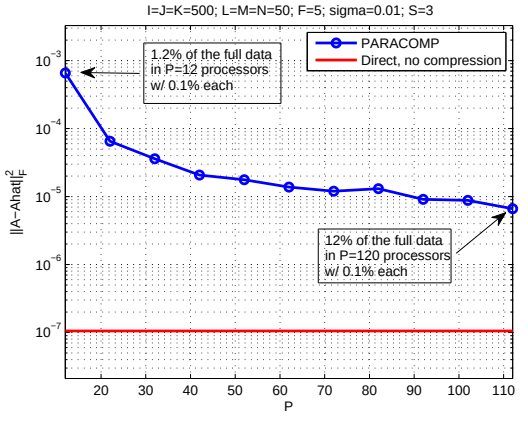

Fig. 4. MSE as a function of P, the number of replicas / parallel threads spawned.

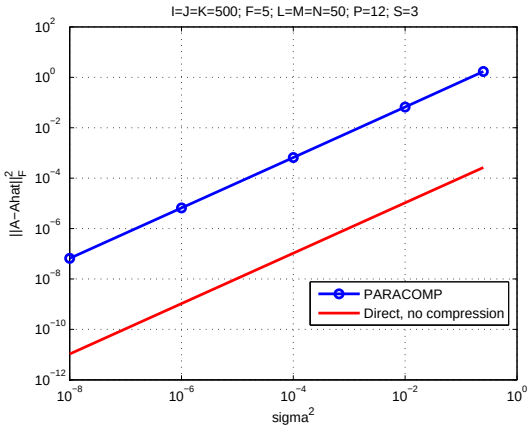

Fig. 5. MSE as a function of additive white Gaussian noise variance σ^2.

VIII. Summary and Take-Home Points

Summary: We have reviewed the basics of tensors and tensor decomposition, and presented a novel architecture for parallel and distributed computation of low-rank tensor decomposition that is especially well-suited for big tensors. It is based on parallel processing of a set of randomly compressed, reduced-size 'replicas' of the big tensor. We have also provided a friendly introduction to Hadoop-MapReduce, starting from a toy signal processing problem, and going up to sketching a Hadoop implementation of 'tensor decomposition in the cloud'.

Motivation and impact: There is rapidly growing interest in signal processing for big data analytics, and in porting / translating and developing new signal processing algorithms for cloud computing platforms. Tensors are multi-dimensional signals that have found numerous applications in signal processing, machine learning, data mining, and well beyond (psychology, chemistry, life sciences, ...), and they are becoming increasingly important for online marketing, social media, search engines, and many more applications. Tensors easily grow to be really big, as their total size is the product of mode sizes, hence exponential in the number of modes ('dimensions' in signal processing parlance). Big tensor data will thus be a big part of big data.

Take-home points:

1) PARACOMP enables massive parallelism with guaranteed identifiability properties: if the big tensor is indeed of low rank and the system parameters are appropriately chosen, then the rank-one factors of the big tensor will indeed be recovered from the analysis of the reduced-size replicas.
2) PARACOMP affords memory / storage and complexity gains of order up to $\frac{IJ}{F}$ for a big tensor of size $I \times J \times K$ of rank F. No sparsity is required, although such sparsity can be exploited to improve memory, storage and computational savings.
3) We have shown that using white noise-like compression matrices
   - approximately preserves component ordering;
   - ensures that the compressed noise is approximately white if the original measurement noise is white; and
   - makes the compressed noise look Gaussian - rendering classical least-squares CP algorithms well-suited for fitting the reduced-size replicas, even if the measurement noise in the big tensor is far from Gaussian.
4) Each replica is independently decomposed, and the results are joined via a master linear equation per tensor mode. The

number of replicas and the size of each replica can be adjusted to fit the number of computing nodes and the memory available to each node, and each node can run its own CP software, depending on its computational capabilities. This flexibility is why PARACOMP is better classified as a computational architecture, as opposed to a method or algorithm.

Authors

Nicholas D. Sidiropoulos (e-mail: nikos@umn.edu) is currently a Professor in the Department of ECE at the University of Minnesota. He has over 15 years of experience in tensor decomposition and its applications. His research interests include topics in signal processing, communications, convex optimization and approximation of NP-hard problems, and cross-layer resource allocation for wireless networks. His current research focuses primarily on signal and tensor analytics, with applications in cognitive radio, big data, and preference measurement. He received the NSF/CAREER award in 1998, and the IEEE Signal Processing Society (SPS) Best Paper Award in 2001, 2007, and 2011. He served as IEEE SPS Distinguished Lecturer (2008-2009), and as Chair of the IEEE Signal Processing for Communications and Networking Technical Committee (2007-2008). He served as Associate Editor for IEEE Transactions on Signal Processing (2000 - 2006), IEEE Signal Processing Letters (2000 - 2002), and on the editorial board of IEEE Signal Processing Magazine (2009-2011). He currently serves as Area Editor for IEEE Transactions on Signal Processing (2012 -), and as Associate Editor for Signal Processing. He received the 2010 IEEE Signal Processing Society Meritorious Service Award.

Evangelos Papalexakis (e-mail: epapalex@cs.cmu.edu) is a Ph.D. student in the Computer Science Department at Carnegie Mellon University, advised by Professor Faloutsos. Evangelos earned a Diploma and M.Sc. in Electronic and Computer Engineering at the Technical University of Crete, Chania, Greece. Evangelos has considerable experience in tensor decomposition and applications, as well as parallel and distributed computations in Hadoop. Being comfortable in both 'worlds', he has published in the IEEE Transactions on Signal Processing and ICASSP, as well as in prime computer science conferences and journals. Evangelos is also the liaison that connects the CMU and UMN groups in a joint NSF project on Big Tensor Data and its applications in automated web-based language learning and brain data mining.

Christos Faloutsos (e-mail: christos@cs.cmu.edu) is a Professor at Carnegie Mellon University. He has received the Presidential Young Investigator Award by the National Science Foundation (1989), the Research Contributions Award in ICDM 2006, the SIGKDD Innovations Award (2010), nineteen "best paper" awards (including two "test of time" awards), and four teaching awards. He is an ACM Fellow, he has served as a member of the executive committee of SIGKDD; he has published over 200 refereed articles, 11 book chapters and one monograph. He holds six patents and he has given over 30 tutorials and over 10 invited distinguished lectures. His research interests include data mining for graphs and streams, fractals, database performance, and indexing for multimedia and bio-informatics data. Christos has a long-term interest in tensor decompositions and their practical applications in data mining, having published many well-appreciated papers in the area.

REFERENCES

[1] N. Sidiropoulos, E. Papalexakis, and C. Faloutsos, "A Parallel Algorithm for Big Tensor Decomposition Using Randomly Compressed Cubes (PARACOMP)," in *Proc. IEEE ICASSP 2014,* May 4-9, Florence, Italy (submitted).

[2] D. Nion, K. Mokios, N. Sidiropoulos, and A. Potamianos, "Batch and adaptive PARAFAC-based blind separation of convolutive speech mixtures," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 18, no. 6, pp. 1193–1207, 2010.

[3] C. Fevotte and A. Ozerov, "Notes on nonnegative tensor factorization of the spectrogram for audio source separation: Statistical insights and towards self-clustering of the spatial cues," in *Exploring Music Contents*, ser. Lecture Notes in Computer Science, S. Ystad, M. Aramaki, R. Kronland-Martinet, and K. Jensen, Eds. Springer Berlin, 2011, vol. 6684, pp. 102–115.

[4] N. Sidiropoulos, G. Giannakis, and R. Bro, "Blind PARAFAC receivers for DS-CDMA systems," *IEEE Transactions on Signal Processing*, vol. 48, no. 3, pp. 810–823, 2000.

[5] N. Sidiropoulos, R. Bro, and G. Giannakis, "Parallel factor analysis in sensor array processing," *IEEE Transactions on Signal Processing*, vol. 48, no. 8, pp. 2377–2388, 2000.

[6] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Parcube: Sparse parallelizable tensor decompositions." in *ECML/PKDD (1)*, ser. Lecture Notes in Computer Science, P. A. Flach, T. D. Bie, and N. Cristianini, Eds., vol. 7523. Springer, 2012, pp. 521–536.

[7] R. Bro and N. Sidiropoulos, "Least squares regression under unimodality and non-negativity constraints," *Journal of Chemometrics*, vol. 12, pp. 223–247, 1998.

[8] A. Cichocki, D. Mandic, C. Caiafa, A.-H. Phan, G. Zhou, Q. Zhao, and L. De Lathauwer, "Multiway Component Analysis: Tensor Decompositions for Signal Processing Applications," *IEEE Signal Processing Magazine*, 2014 (to appear).

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

[10] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *ICDM 2008: Proceedings of the 8th IEEE International Conference on Data Mining*, December 2008, pp. 363–372.

[11] D. Nion and N. Sidiropoulos, "Adaptive algorithms to track the parafac decomposition of a third-order tensor," *IEEE Trans. on Signal Processing*, vol. 57, no. 6, pp. 2299–2310, 2009.

[12] A. Phan and A. Cichocki, "Parafac algorithms for large-scale problems," *Neurocomputing*, vol. 74, no. 11, pp. 1970–1984, 2011.

[13] N. Sidiropoulos and A. Kyrillidis, "Multi-way compressed sensing for sparse low-rank tensors," *IEEE Signal Processing Letters*, vol. 19, no. 11, pp. 757–760, 2012.

[14] E. Papalexakis, N. Sidiropoulos, and R. Bro, "From k -means to higher-way co-clustering: Multilinear decomposition with sparse latent factors," *IEEE Trans. on Signal Processing*, vol. 61, no. 2, pp. 493–506, 2013.

[15] N. Sidiropoulos, "Low-rank decomposition of multi-way arrays: A signal processing perspective," in *Proc. IEEE SAM Workshop, Sitges, Barcelona, Spain*, 2004.

[16] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM REVIEW*, vol. 51, no. 3, pp. 455–500, 2009.

[17] A. Smilde, R. Bro, P. Geladi, and J. Wiley, *Multi-way analysis with applications in the chemical sciences*. Wiley, 2004.

[18] P. Kroonenberg, *Applied multiway data analysis*. Wiley, 2008.

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

[20] ——, "Determination and proof of minimum uniqueness conditions for PARAFAC-1," *UCLA Working Papers in Phonetics*, vol. 22, pp. 111–117, 1972.

[21] 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, pp. 283–319, 1970.

[22] J. Kruskal, "Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear Algebra and its Applications*, vol. 18, no. 2, pp. 95–138, 1977.

[23] N. Sidiropoulos and R. Bro, "On the uniqueness of multilinear decomposition of N-way arrays," *Journal of chemometrics*, vol. 14, no. 3, pp. 229–239, 2000.

[24] T. Jiang and N. Sidiropoulos, "Kruskal's permutation lemma and the identification of CANDECOMP/PARAFAC and bilinear models with constant modulus constraints," *IEEE Transactions on Signal Processing*, vol. 52, no. 9, pp. 2625–2636, 2004.

[25] A. Stegeman and N. Sidiropoulos, "On Kruskal's uniqueness condition for the CANDECOMP/PARAFAC decomposition," *Linear Algebra and its Applications*, vol. 420, no. 2-3, pp. 540–552, 2007.

[26] L. Chiantini and G. Ottaviani, "On generic identifiability of 3-tensors of small rank," *SIAM. J. Matrix Anal. & Appl.*, vol. 33, no. 3, pp. 1018–1037, 2012.

[27] A. Stegeman, J. ten Berge, and L. De Lathauwer, "Sufficient conditions for uniqueness in CANDECOMP/PARAFAC and INDSCAL with random component matrices," *Psychometrika*, vol. 71, no. 2, pp. 219–229, 2006.

[28] R. Bro, N. Sidiropoulos, and G. Giannakis, "A fast least squares algorithm for separating trilinear mixtures," in *Proc. ICA99 Int. Workshop on Independent Component Analysis and Blind Signal Separation*, 1999, pp. 289–294. [Online]. Available: http://www.ece.umn.edu/~nikos/comfac.m

[29] J. Brewer, "Kronecker products and matrix calculus in system theory," *IEEE Trans. on Circuits and Systems*, vol. 25, no. 9, pp. 772–781, 1978.

[30] D. Donoho and M. Elad, "Optimally sparse representation in general (nonorthogonal) dictionaries via minimization," *Proc. Nat. Acad. Sci.*, vol. 100, no. 5, pp. 2197–2202, 2003.

[31] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[32] Apache, "Hadoop." [Online]. Available: http://hadoop.apache.org/

[33] A. Hadoop, "Word count example." [Online]. Available: http://wiki.apache.org/hadoop/WordCount

[34] 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*. ACM, 2012, pp. 316–324.

[35] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2011, pp. 13–25.

[36] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.