Memory-Efficient Parallel Computation of Tensor and Matrix Products for Big Tensor Decomposition

Niranjay Ravindran*, Nicholas D. Sidiropoulos*, Shaden Smith†, and George Karypis†
* Dept. of Electrical and Computer Engineering † Dept. of Computer Science and Engineering
University of Minnesota, Minneapolis

***Abstract*— Low-rank tensor decomposition has many applications in signal processing and machine learning, and is becoming increasingly important for analyzing big data. A significant challenge is the computation of intermediate products which can be much larger than the final result of the computation, or even the original tensor. We propose a scheme that allows memory-efficient in-place updates of intermediate matrices. Motivated by recent advances in big tensor decomposition from multiple compressed replicas, we also consider the related problem of memory-efficient tensor compression. The resulting algorithms can be parallelized, and can exploit but do not require sparsity.**

I. Introduction

Tensors (or multi-way arrays) are data structures indexed by three or more indices. They are a generalization of matrices which have only two indices: a row index and a column index. Many examples of real world data are stored in the form of very large tensors, for example, the Never Ending Language Learning (NELL) database [1] has dimensions 26 million $\times$ 26 million $\times$ 48 million. Many big tensors are also very sparse, for instance, the NELL tensor has only 144 million non-zero entries.

Tensor factorizations have already found many applications in chemistry, signal processing, and machine learning, and they are becoming more and more important for analyzing big data. *Parallel Factor Analysis* (PARAFAC) [2] or *Canonical Decomposition* (CANDECOMP) [3], referred to hereafter as CP, synthesizes an $I \times J \times K$ three-way tensor $\underline{\mathbf{X}}$ as the sum of F outer products:

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

where $\circ$ denotes the vector outer product, $\mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f(i,j,k) = \mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k)$ for $1 \leq i \leq I, 1 \leq j \leq J$ and $1 \leq k \leq K$ and $\mathbf{a}_f \in \mathbb{R}^{I\times 1}, \mathbf{b}_f \in \mathbb{R}^{J\times 1}, \mathbf{c}_f \in \mathbb{R}^{K\times 1}$. The smallest F that allows synthesizing $\underline{\mathbf{X}}$ this way is the *rank* of $\underline{\mathbf{X}}$. A matrix of rank F can be synthesized as the sum of F rank one matrices. Similarly, a rank F tensor can be synthesized as the sum of F rank one tensors, $\mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f \in \mathbb{R}^{I\times J\times K}$ for $f = 1, \ldots, F$. Compared to other tensor decompositions, the CP model is special because it can be interpreted as rank decomposition, and because of its uniqueness properties. Under certain conditions, the rank-one tensors $\mathbf{a}_f \circ \mathbf{b}_f \circ \mathbf{c}_f$ are unique [4]–[6]. That is, given $\underline{\mathbf{X}}$ of rank F, there is only one way to decompose it into F rank one tensors. This is important in many applications where we are interested in unraveling the latent factors that generate the observed data.

Least-squares fitting a rank F CP model to a tensor is an NP-hard problem; but the method of *alternating least squares* (ALS) usually yields good approximate solutions in practice, approaching the Cramér-Rao bound in 'well-determined' cases. While very useful in practice, this approach presents several important but also interesting challenges in the case of big tensors. For one, the size of intermediate products can be much larger than the final result of the computation, referred to as the *intermediate data explosion* problem. Another bottleneck is that the entire tensor needs to be accessed in each ALS iteration, requiring large amounts of memory and incurring large data transport costs. Moreover, the tensor data is accessed in different orders inside each iteration, which makes efficient block caching of the tensor data for fast memory access difficult (unless the tensor is replicated multiple times in memory). Addressing these concerns is critical in making big tensor decomposition practical. Further, due to the large number of computations involved and possibly distributed storage of the big tensor, it is desirable to formulate scalable and parallel tensor decomposition algorithms.

In this work, we propose two improved algorithms. No sparsity is required for either algorithm, although sparsity can be exploited for memory and computational savings. The first is a scheme that effectively allows in-place updates of the factors in each ALS iteration with no intermediate memory explosion, as well as reduced memory and complexity relative to prior methods. Further, the algorithm has a consistent block access pattern of the tensor data, which can be exploited for efficient caching/pre-fetching of data. Finally, the algorithm can be parallelized such that each parallel thread only requires access to one portion of the big tensor, favoring a distributed storage setting.

The above approach directly decomposes the big tensor and requires accessing the entire tensor data multiple times, and so may not be suited for very large tensors that do not fit in memory. To address this, recent results indicate that randomly compressing the big tensor into multiple small tensors, independently decomposing each small tensor in parallel, and finally merging the resulting decompositions, can identify the correct decomposition of the big tensor *and* provide significant memory and computational savings. The tensor data is

Work supported by NSF IIS-1247632. Authors can be reached at ravi0022@umn.edu, nikos@umn.edu (contact author), shaden@cs.umn.edu, karypis@cs.umn.edu.

accessed only once during the compression stage, and further operations are only performed on the smaller tensors. If the big tensor is indeed of low rank, and the system parameters are appropriately chosen, then its low rank decomposition is fully recoverable from those of the smaller tensors. Moreover, each of the smaller tensors can be decomposed in parallel without requiring access to the big tensor data. This approach will be referred to as PARACOMP (parallel randomly compressed tensor decomposition) in the sequel.

While PARACOMP provides a very attractive alternative to directly decomposing a big tensor, the actual compression of the big tensor continues to pose a challenge. One approach is to compute the value of each element of the compressed tensor one at a time, which requires very little intermediate memory and can exploit sparsity in the data, but has very high computational complexity for dense data. Another approach is to compress one mode of the big tensor at a time, which greatly reduces the complexity of the operations, but requires large amounts of intermediate memory. In fact, for sparse tensors, the intermediate result after compressing one mode is, in general, dense, and can occupy more memory than the original tensor, even if the final compressed tensor is small.

To address these problems, we propose a new algorithm for compressing big tensors via multiple mode products. This algorithm achieves the same flop count as compressing the tensor one mode at a time for dense tensors, and better flop count than computing the value of the compressed tensor one element at a time for sparse tensors, but with only limited intermediate memory, typically smaller than the final compressed tensor.

II. Memory Efficient CP Decomposition

Let $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ be $I \times F$, $J \times F$ and $K \times F$ matrices, whose columns are comprised of $\{\mathbf{a}_f\}_{f=1}^{F}, \{\mathbf{b}_f\}_{f=1}^{F}$ and $\{\mathbf{c}_f\}_{f=1}^{F}$ respectively. Let $\mathbf{X}(:,:,k)$ denote the k^{th} $I \times J$ matrix "slice" of $\underline{\mathbf{X}}$.

Let $\mathbf{X}_{(3)}^T$ denote the $IJ \times K$ matrix whose k^{th} column is $\text{vec}(\mathbf{X}(:,:,k))$. Similarly define the $JK \times I$ matrix $\mathbf{X}_{(1)}^T$ and the $IK \times J$ matrix $\mathbf{X}_{(2)}^T$. Then, there are three equivalent ways of expressing the decomposition in (1) as

$$\mathbf{X}_{(1)}^T = (\mathbf{C} \odot \mathbf{B})\mathbf{A}^T \quad (2)$$

$$\mathbf{X}_{(2)}^T = (\mathbf{C} \odot \mathbf{A})\mathbf{B}^T \quad (3)$$

$$\mathbf{X}_{(3)}^T = (\mathbf{B} \odot \mathbf{A})\mathbf{C}^T \quad (4)$$

where $\odot$ represents the Khatri-Rao product (note the transposes in the definitions and expressions above). This can also be written as:

$$\text{vec}(\underline{\mathbf{X}}) = (\mathbf{C} \odot \mathbf{B} \odot \mathbf{A})\mathbf{1} \quad (5)$$

using the vectorization property of the Khatri-Rao product. In practice, due to the presence of noise, or in order to fit a higher rank tensor to a lower rank model, we seek an approximate solution to the above decomposition, in the least squares sense

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} ||\mathbf{X}_{(1)} - (\mathbf{C} \odot \mathbf{B})\mathbf{A}^T||_F^2. \quad (6)$$

using the represenation (2). The above is an NP-hard problem in general, but fixing $\mathbf{A}$, $\mathbf{B}$ and solving for $\mathbf{C}$ is only a linear least-squares problem which has a closed-form solution for $\mathbf{C}$:

$$\mathbf{C} = \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^\dagger \quad (7)$$

where $*$ stands for the Hadamard (element-wise) product and $\dagger$ denotes the pseudo-inverse of a matrix. Rearranging (6) to use the forms (2) and (3) can similarly provide solutions for updating $\mathbf{A}$ and $\mathbf{B}$ respectively, given that the other two factors are fixed:

$$\mathbf{A} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \quad (8)$$

$$\mathbf{B} = \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^\dagger \quad (9)$$

Iterating the above steps yields an *alternating least-squares* (ALS) algorithm to fit a rank-F CP model to the tensor $\underline{\mathbf{X}}$.

The above algorithm, while useful in practice, presents several challenges for the computations (7), (8) and (9). For one, the computation of the intermediate Khatri-Rao products, for example, the $IJ \times F$ matrix $(\mathbf{B} \odot \mathbf{A})$, can be much larger than the final result of the computation (7), and may require large amounts of intermediate memory. This is referred to as *intermediate data explosion*. Another drawback is that the entire tensor data $\underline{\mathbf{X}}$ needs to be accessed for each of these computations in each ALS iteration, requiring large amounts of memory and incurring large data transport costs. Moreover, the access pattern of the tensor data is different for (7), (8) and (9), making efficient block caching of the tensor data for fast memory access difficult (unless the tensor is repeated three times in memory, one for each of the matricizations $\mathbf{X}_{(1)}, \mathbf{X}_{(2)}$ and $\mathbf{X}_{(3)}$). Addressing these concerns is critical in making big tensor decomposition practical. Furthermore, due to the large number of computations involved and possibly distributed storage of the big tensor, it is desirable to formulate the algorithm so it is easily and efficiently parallelizable.

Without memory-efficient algorithms, the direct computation of, say, $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$, requires $O(JKF)$ memory to store $\mathbf{C} \odot \mathbf{B}$, in addition to $O(\text{NNZ})$ memory to store the tensor data, where NNZ is the number of non-zero elements in the tensor $\underline{\mathbf{X}}$. Further, JKF flops are required to compute $(\mathbf{C} \odot \mathbf{B})$, and $JKF + 2F$ NNZ flops to compute its product with $\mathbf{X}_{(1)}$. This step is the bottleneck in the computations (7)-(9). Note that the pseudo-inverse can be computed at relatively less complexity since $\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}$ is an $F \times F$ matrix.

In [7], [8], a variety of tensor analysis algorithms exploiting sparsity in the data to reduce memory storage requirements are presented. The *Tensor Toolbox* [9] computes $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ with $3F$ NNZ flops using NNZ intermediate memory (on top of that required to store the tensor) [7]. The algorithm avoids intermediate data explosion by 'accumulating' tensor-matrix operations, but it does not provision for efficient parallelization (the accumulation step must be performed serially). In [10], a parallel algorithm that computes $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ with $5F$ NNZ flops using $O(\max(J + \text{NNZ}, K + \text{NNZ}))$ intermediate memory is presented. The algorithm in [10] admits MapReduce implementation.

Algorithm 1 Computing $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$

Input: $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}, \mathbf{B} \in \mathbb{R}^{J \times F}, \mathbf{C} \in \mathbb{R}^{K \times F}$
Output: $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{I \times F}$

1: $\mathbf{M}_1 \leftarrow \mathbf{0}$
2: **for** $k = 1, \ldots, K$ **do**
3: $\quad \mathbf{M}_1 \leftarrow \mathbf{M}_1 + \underline{\mathbf{X}}(:,:,k)\mathbf{B}\ \text{diag}(\mathbf{C}(k,:))$
4: **end for**

Algorithm 2 Computing $\mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})$

Input: $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}, \mathbf{A} \in \mathbb{R}^{I \times F}, \mathbf{C} \in \mathbb{R}^{K \times F}$
Output: $\mathbf{M}_2 \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A}) \in \mathbb{R}^{J \times F}$

1: $\mathbf{M}_2 \leftarrow \mathbf{0}$
2: **for** $k = 1, \ldots, K$ **do**
3: $\quad \mathbf{M}_2 \leftarrow \mathbf{M}_2 + \underline{\mathbf{X}}(:,:,k)\mathbf{A}\text{diag}(\mathbf{C}(k,:))$
4: **end for**

In this work, we present an algorithm for the computation of $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ which requires only $O(\text{NNZ})$ intermediate memory, that is, the updates of $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ can be effectively performed in place. This is summarized in Algorithm 1. Further, the algorithm has a consistent block access pattern of the tensor data, which can be used for efficient caching/pre-fetching, and offers complexity savings relative to [10], as quantified next.

Let I_k be the number of non-empty rows in the k-th $I \times J$ "slice" of the tensor, $\underline{\mathbf{X}}(:,:,k)$. Similarly, let J_k be the number of non-empty columns in $\underline{\mathbf{X}}(:,:,k)$. Define:

$$\text{NNZ}_1 = \sum_{k=1}^{K} I_k \tag{10}$$

$$\text{NNZ}_2 = \sum_{k=1}^{K} J_k \tag{11}$$

where NNZ_1 and NNZ_2 are the total number of non-empty rows and columns in the tensor $\underline{\mathbf{X}}$. Algorithm 1 computes $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ in F $\text{NNZ}_1 + F$ $\text{NNZ}_2 + 2F$ NNZ flops, assuming that empty rows and columns of $\underline{\mathbf{X}}(:,:,k)$ can be identified offline and skipped during the matrix multiplication and update of $\mathbf{M_1}$ operations. To see this, note that we only need to scale by $\text{diag}(\mathbf{C}(k,:))$ those rows of $\mathbf{B}$ corresponding to nonempty columns of $\underline{\mathbf{X}}(:,:,k)$, and this can be done using FJ_k flops, for a total of $F\text{NNZ}_2$. Next, the multiplications $\underline{\mathbf{X}}(:,:,k)\mathbf{B}\ \text{diag}(\mathbf{C}(k,:))$ can be carried out for all k at $2F$ NNZ flops (counting additions and multiplications). Finally, only rows of $\mathbf{M}_1$ corresponding to nonzero rows of $\underline{\mathbf{X}}(:,:,k)$ need to be updated, and the cost of each row update is F, since $\underline{\mathbf{X}}(:,:,k)\mathbf{B}\ \text{diag}(\mathbf{C}(k,:))$ has F columns. Hence the total $\mathbf{M}_1$ row updates cost is $F\text{NNZ}_1$ flops, for an overall F $\text{NNZ}_1 + F$ $\text{NNZ}_2 + 2F$ NNZ flops. This offers complexity savings relative to [10] since $\text{NNZ} > \text{NNZ}_1, \text{NNZ}_2$.

Algorithm 3 Computing $\mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})$

Input: $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}, \mathbf{A} \in \mathbb{R}^{I \times F}, \mathbf{B} \in \mathbb{R}^{J \times F}$
Output: $\mathbf{M}_3 \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A}) \in \mathbb{R}^{K \times F}$

1: **for** $k = 1, \ldots, K$ **do**
2: $\quad \mathbf{M}_3(k,:) \leftarrow \mathbf{1}^T (\mathbf{A} * (\underline{\mathbf{X}}(:,:,k)\mathbf{B}))$
3: **end for**

Algorithm 2 summarizes the method for computation of $\mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})$, which has the same complexity and storage requirements as Algorithm 1, while still maintaining the same pattern of accessing the tensor data in the form of $I \times J$ "slices". Algorithm 3 computes $\mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})$, but differs from Algorithms 1 and 2 in order to maintain the same access pattern of the tensor data (with similar complexity and storage). Hence, there is no need to re-order the tensor between each computation stage or store multiple copies of the tensor.

We do not discuss parallel implementations of Algorithms 1, 2, and 3 in detail in this work, but point out that the loops in Step 2 can be parallelized across K threads where each thread only requires access to an $I \times J$ slice of the tensor. This favors a distributed storage structure for the tensor data. The thread synchronization requirements are also different between Algorithms 1, 2 and Algorithm 3, since in Algorithm 3 only F elements of a single column of the matrix $\mathbf{M}_3$ will need to be updated (independently) by each thread. The procedure in Algorithm 3 can be used to compute the results in Algorithms 1 and 2, and vice versa, although this may not preserve the same access pattern of the tensor in terms of $I \times J$ slices. Efficient parallel implementations of the above algorithms are currently being explored in ongoing work.

III. Memory Efficient Computation of Tensor Mode Products

A critical disadvantage of directly using ALS for CP decomposition of a big tensor is that there is still a requirement of repeatedly accessing the entire tensor data multiple times for each iteration. While the parallelization suggested for Algorithms 1, 2 and 3 can split the tensor data over multiple parallel threads, thereby mitigating the data access cost, this cost may still dominate for very large tensors which may not fit in random access or other high performance memory.

Several methods have been proposed to alleviate the need to store the entire tensor in high-performance memory. Biased random sampling is used in [11], and is shown to work well for sparse tensors, albeit without identifiability guarantees. In [12], the big tensor is randomly compressed into a smaller tensor. If the big tensor admits a low-rank decomposition with sparse latent factors, the random sampling guarantees identifiability of the low-rank decomposition of the big tensor from that of the smaller tensor. However, this guarantee may not hold if the latent factors are not sparse. In [13], a method of randomly compressing the big tensor into multiple small tensors (PARACOMP) is proposed, where each small tensor is independently decomposed, and the decompositions are related through a master linear equation. The tensor data is accessed only once during the compression stage, and further operations are only performed on the smaller tensors. If the big tensor is indeed of low rank, and the system parameters are

appropriately chosen, then its low rank decomposition is fully recoverable from those of the smaller tensors. PARACOMP offers guaranteed identifiability, natural parallelization, and overall complexity and memory savings of order $\frac{IJ}{F}$. Sparsity is not required, but can be exploited in PARACOMP. Given

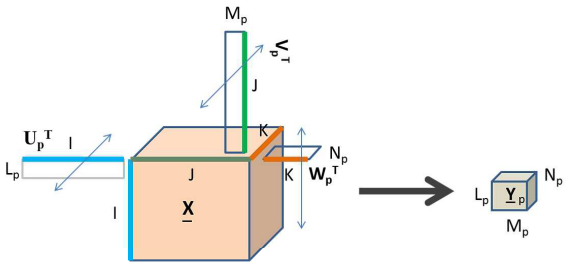

Fig. 1. Compressing $\underline{\mathbf{X}}$ $(I \times J \times K)$ to $\underline{\mathbf{Y}}_p$ $(L_p \times M_p \times N_p)$ by multiplying (every slab of) $\underline{\mathbf{X}}$ from the I-mode with $\mathbf{U}_p^T$, from the J-mode with $\mathbf{V}_p^T$, and from the K-mode with $\mathbf{W}_p^T$.

compression matrices $\mathbf{U}_p \in \mathbb{R}^{I \times L_p}$, $\mathbf{V}_p \in \mathbb{R}^{J \times M_p}$ and $\mathbf{W}_p \in \mathbb{R}^{K \times N_p}$, the compressed tensor $\underline{\mathbf{Y}}_p \in \mathbb{R}^{L_p \times M_p \times N_p}$ computed as $\underline{\mathbf{Y}}_p(l,m,n) =$

$$\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} \mathbf{U}_p(l,i)\mathbf{V}_p(m,j)\mathbf{W}_p(n,k)\underline{\mathbf{X}}(i,j,k) \qquad (12)$$

for $l \in \{1,\ldots,L_p\}$, $m \in \{1,\ldots,M_p\}$ and $n \in \{1,\ldots,N_p\}$. The compression operation in (12) can be thought of as multiplying (and compressing) the big tensor $\underline{\mathbf{X}}$ by $\mathbf{U}_p^T$ along the first mode, $\mathbf{V}_p^T$ along the second mode, and $\mathbf{W}_p^T$ along the third mode. This corresponds to compressing each of the dimensions of $\underline{\mathbf{X}}$ independently, i.e., reducing the dimension I to L_p, J to M_p and K to N_p. The result is a smaller "cube" of dimensions $L_p \times M_p \times N_p$. This process is depicted in Figure 1. This computation is repeated for $p = 1,\ldots,P$, for some P and

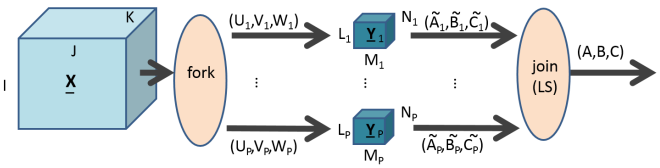

Fig. 2. The PARACOMP fork-join architecture. The fork creates P randomly compressed reduced-size 'replicas' $\left\{\underline{\mathbf{Y}}_p\right\}_{p=1}^{P}$, obtained by applying $(\mathbf{U}_p, \mathbf{V}_p, \mathbf{W}_p)$ to $\underline{\mathbf{X}}$, as in Fig. 1. Each $\underline{\mathbf{Y}}_p$ is independently factored. The join collects the estimated mode loading sub-matrices $\left(\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p, \tilde{\mathbf{C}}_p\right)$, anchors them all to a common permutation and scaling, and solves a linear least squares problem to estimate $(\mathbf{A}, \mathbf{B}, \mathbf{C})$.

$L_1,\ldots,L_P << I, M_1,\ldots,M_P << J$ and $N_1,\ldots,N_P << K$, to form P compressed tensors $\underline{\mathbf{Y}}_1,\ldots,\underline{\mathbf{Y}}_P$. This operation can be performed in parallel over P threads. Each of the P compressed tensors $\underline{\mathbf{Y}}_1,\ldots,\underline{\mathbf{Y}}_P$ is independently decomposed into CP factors denoted by $\tilde{\mathbf{A}}_p, \tilde{\mathbf{B}}_p$ and $\tilde{\mathbf{C}}_p$ with a rank of F, for $p = 1,\ldots,P$, using, for example, the ALS algorithm as described in Section II. Finally, the F factors are "joined" to compute $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$, the rank F decomposition of the big tensor $\underline{\mathbf{X}}$. This process is depicted in Figure 2.

PARACOMP is particularly attractive due to its natural parallel structure and the fact that the big tensor data does not need to be repeatedly accessed for the decomposition. The initial compression stage, however, can be a bottleneck. On the bright side, (12) can be performed "in place", requires only bounded intermediate memory, and it can exploit sparsity by summing only over the non-zero elements of $\underline{\mathbf{X}}$. On the other hand, complexity is $O(LMNIJK)$ for a dense tensor, and $O(LMN(\text{NNZ}))$ for a sparse tensor. The main issue with (12), however, is that it features a terrible memory access pattern which can really bog down computations. An alternative computation schedule comprises three steps:

$$\underline{\mathbf{T}}_1(l,j,k) = \sum_{i=1}^{I} \mathbf{U}(l,i)\underline{\mathbf{X}}(i,j,k),$$
$$\forall l \in \{1,\ldots,L_p\}, j \in \{1,\ldots,J\}, k \in \{1,\ldots,K\} \qquad (13)$$

$$\underline{\mathbf{T}}_2(l,m,k) = \sum_{j=1}^{J} \mathbf{V}(m,j)\underline{\mathbf{T}}_1(l,j,k),$$
$$\forall l \in \{1,\ldots,L_p\}, m \in \{1,\ldots,M_p\}, k \in \{1,\ldots,K\} \qquad (14)$$

$$\underline{\mathbf{Y}}(l,m,n) = \sum_{k=1}^{K} \mathbf{W}(n,k)\underline{\mathbf{T}}_2(l,m,k),$$
$$\forall l \in \{1,\ldots,L_p\}, m \in \{1,\ldots,M_p\}, n \in \{1,\ldots,N_p\} \qquad (15)$$

which has only $O(LIJK + MLJK + NLMK)$ complexity for a dense tensor, as opposed to $O(LMNIJK)$. This corresponds to compressing the I mode first, followed by the J mode, and finally the K mode. The reduction in complexity is achieved by storing intermediate results in (12) using tensors $\underline{\mathbf{T}}_1$ and $\underline{\mathbf{T}}_2$, instead of computing them multiple times.

The ordering of the multiplications can be changed in order to minimize the overall complexity. For three modes, there are 6 different orderings, each potentially yielding a different computational complexity. However, for $I \leq J \leq K$ with $\frac{I}{L} = \frac{J}{M} = \frac{K}{N}$, it can be shown that it is almost always preferable to perform the multiplications in the order shown in (13)-(15) for dense tensors, i.e., compress the smallest mode first, and proceed in ascending order of the size of the modes.

However, despite the substantial computational savings compared to (12), the computations (13)-(15) require very large amounts of intermediate memory. Further, for a sparse tensor $\underline{\mathbf{X}}$, the first mode compression can be performed for only the non-zero elements of the tensor – thus exploiting sparsity – but the intermediate tensors $\underline{\mathbf{T}}_1$ and $\underline{\mathbf{T}}_2$ are, in general, dense, potentially requiring even more memory than the original tensor! The implementation in (12) fully exploits sparsity and does not require any intermediate memory, and as observed in [13], there appears to be a tradeoff between complexity and memory savings, with and without sparsity.

In Algorithm 4, we propose a scheme that achieves the same flop count as (13)-(15) for dense tensors, and better flop count compared to (12) for sparse tensors, but requires only limited intermediate memory in the form of $\mathbf{T}'_1 \in \mathbb{R}^{L \times B}$ and $\mathbf{T}'_2 \in \mathbb{R}^{L \times M}$, for any choice of $B \leq J$. For example, $B = 1$ results in maximum memory savings – but in practice, setting B to a higher value to ensure maximum cache utilization may be a preferable option. The choice of B does not affect the computational complexity. For most processing systems, it should be possible to make B small enough such that the intermediate matrices are smaller than the final result $\underline{\mathbf{Y}}$ for small enough B, thus addressing the intermediate data explosion problem.

Algorithm 4 Compression of $\underline{\mathbf{X}}$ into $\underline{\mathbf{Y}}_p$

Input: $\underline{\mathbf{X}} \in \mathbb{R}^{I \times J \times K}, \mathbf{U}_p \in \mathbb{R}^{I \times L_p}, \mathbf{V}_p \in \mathbb{R}^{J \times M_p}, \mathbf{W}_p \in \mathbb{R}^{K \times N_p}$, Block size B
Output: $\underline{\mathbf{Y}}_p \in \mathbb{R}^{L_p \times M_p \times N_p}$

1: **for** $k = 1, \ldots, K$ **do**
2: $\quad \mathbf{T}'_2 \leftarrow \mathbf{0}$
3: $\quad$ **for** $b = 1, B+1, 2B+1, 3B+1, \ldots, J$ **do**
4: $\quad\quad \mathbf{T}'_1 \leftarrow \mathbf{U}_p^T \underline{\mathbf{X}}(:, b : (b+B-1), k)$
5: $\quad\quad \mathbf{T}'_2 \leftarrow \mathbf{T}'_2 + \mathbf{T}'_1 \mathbf{V}_p(b : (b+B-1), :)$
6: $\quad$ **end for**
7: $\quad$ **for** $n = 1, \ldots, N_p$ **do**
8: $\quad\quad \underline{\mathbf{Y}}_p(:, :, n) \leftarrow \underline{\mathbf{Y}}_p(:, :, n) + W(n, k)\mathbf{T}'_2$
9: $\quad$ **end for**
10: **end for**

This algorithm can be generalized to tensors with more than 3 modes as follows: the computations in steps 4 and 5 can be used to compress a pair of modes at a time, while the computation in step 8 can be used for the last mode in the case of an odd number of modes. The main result that the intermediate memory required is typically less than the final result will continue to hold.

Next, it is important to consider the exploitation of sparsity. Clearly, the matrix multiplication in Step 4 can fully exploit sparsity in the tensor data. Further, if we assume that the empty columns of $\mathbf{T}_1$ can be identified, then these columns need not participate while updating $\mathbf{T}_2$ in Step 5. Hence, the overall complexity for sparse data is $O(L(\text{NNZ}) + LM(\text{NNZ}_2) + LMNK$. Note that (12) has a higher complexity of $O(LMN(\text{NNZ}))$, since $\text{NNZ} > K, \text{NNZ}_2$. Hence, the above algorithm exploits sparsity better than the method of looping over all the non-zero elements of $\underline{\mathbf{X}}$ using (12). However, despite this, it is clear sparsity is only fully exploited while compressing the first mode. Only fully empty columns after compressing the first mode can be exploited while compressing the second mode. Hence it is tempting to believe that further optimizations of Algorithm 4 for sparse data are possible, which is currently the subject of ongoing work.

Finally, we note that there is more than one way of parallelizing Algorithm 4. The straightforward method of parallelizing the P compression steps over P threads requires that each thread accesses the entire tensor once. However, if the parallelization is done over the for loop in Step 1 in Algorithm 4, i.e., over as many as K threads, where each thread handles $p = 1, \ldots, P$, only a "slice" of the tensor data, $\underline{\mathbf{X}}(:, :, k)$, is required at each thread. Similar to algorithms 1, 2, and 3, this architecture favors situations where the tensor data is stored in a distributed fashion, with only a portion of the data locally available to each thread. Analyzing the communication overhead associated with various parallel implementations of Algorithm 4 will be explored in other work.

IV. Conclusions

New memory-efficient algorithms for streamlining the computation of intermediate products appearing in tensor decomposition using ALS and PARACOMP were proposed. These are often the bottleneck of the entire computation, especially for big tensors. The new algorithms have favorable memory access patterns without requiring big tensor data replication. They also feature reduced computational complexity relative to the prior art, in many cases of practical interest. Another notable feature is that they are naturally amenable to parallelization. Performance optimization for specific high-performance computing architectures is currently being explored.

References

[1] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka Jr., and T. Mitchell, "Toward an Architecture for Never-Ending Language Learning," in *AAAI*, 2010.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] B. Bader and T. Kolda, "Efficient MATLAB Computations with Sparse and Factored Tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2008. [Online]. Available: http://dx.doi.org/10.1137/060676489

[8] T. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *Proc. IEEE ICDM 2008*, pp. 363–372.

[9] B. W. Bader, T. G. Kolda, *et al.*, "Matlab tensor toolbox version 2.5," Available online, January 2012. [Online]. Available: http://www.sandia.gov/ tgkolda/TensorToolbox/

[10] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," in *Proc. ACM SIGKDD*, 2012, pp. 316–324.

[11] E. Papalexakis, C. Faloutsos, and N. Sidiropoulos, "Parcube: Sparse parallelizable tensor decompositions," in *Proc. European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases 2012 (ECML-PKDD)*. Springer, 2012, pp. 521–536.

[12] N. D. 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.

[13] N. Sidiropoulos, E. Papalexakis, and C. Faloutsos, "PArallel RAndomly COMPressed Cubes: A Scalable Distributed Architecture for Big Tensor Decomposition," *IEEE Signal Processing Magazine (Special Issue on Signal Processing for Big Data)*, pp. 57–70, Sep. 2014.