High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors

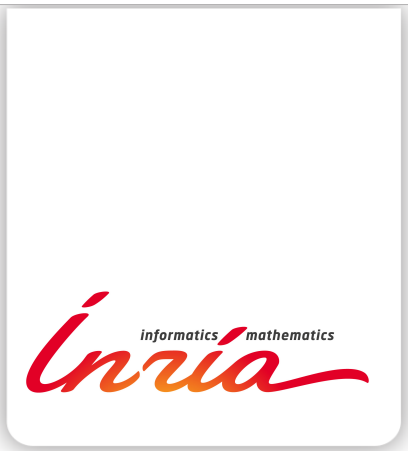
Oguz Kaya, Bora Uçar

High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors

Oguz Kaya, Bora Uçar

RESEARCH REPORT
N° 8801
October 2015
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8801--FR+ENG



High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors

Oguz Kaya*, Bora Uçar†

Project-Team ROMA

Research Report n° 8801 — October 2015 — 20 pages

Abstract: We investigate an efficient parallelization of a class of algorithms for the well-known Tucker decomposition of general N-dimensional sparse tensors. The targeted algorithms are iterative and use the alternating least squares method. At each iteration, for each dimension of an N-dimensional input tensor, the following operations are performed: (i) the tensor is multiplied with $(N-1)$ matrices (TTM step); (ii) the product is then converted to a matrix; and (iii) a few leading left singular vectors of the resulting matrix are computed (SVD step) to update one of the matrices for the next TTM step. We propose an efficient parallelization of these algorithms for current supercomputers comprised of compute nodes, where each node is a multi-core system. We reformulate the computation of N successive TTM-steps to increase the reuse of intermediate computation, which is of interest on its own. We discuss a set of preprocessing steps which takes all computational decisions out of the main iteration of the algorithm and provide an intuitive row-wise shared-memory parallelism for the TTM and SVD steps. We consider a coarse and a fine grain computational scheme, investigate their data dependencies, and identify efficient communication schemes. We demonstrate how the computation of singular vectors in the SVD step can be carried out efficiently following the TTM step. Finally, we develop a hybrid MPI-OpenMP based implementation of the overall algorithm and report speedup results on up to 2048 cores.

Key-words: sparse tensors, parallel tensor factorization

* Inria and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France
† CNRS and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes parallèles haute performance pour la décomposition Tucker des tenseurs creux

Résumé : Nous nous intéressons à la parallélisation efficace d'une classe d'algorithmes pour la décomposition Tucker de tenseurs creux en dimension N. Les algorithmes ciblés sont itératifs et utilisent la méthode alternée des moindres carrés. A chaque itération, pour chaque dimension d'un tenseur de dimension N, les opérations suivantes sont effectuées: (i) le tenseur est multiplié par N-1 matrices (étape TTM); (ii) le produit est converti en une matrice; et (iii) des vecteurs gauches singuliers de la matrice résultante sont calculés (étape SVD) afin de mettre à jour l'une des matrices pour la prochaine étape TTM. Nous proposons une parallélisation efficace de ces algorithmes pour les supercalculateurs dans lesquels chaque noeud est un système multi-coeurs. Nous reformulons le calcul de N étapes TTM successives afin d'améliorer la réutilisation de calculs intermédiaires. Nous discutons un ensemble d'étapes de pré-traitement qui sortent les décisions de calcul hors de l'itération principale de l'algorithme, et nous fournissons une parallélisation intuitive pour les étapes TTM et SVD. Nous considérons un schéma de calcul à grain fin et à gros grain, nous étudions les dépendances de données, et nous identifions des schémas de communication efficaces. Nous démontrons comment le calcul de vecteurs singuliers dans l'étape SVD peut être menée efficacement suite à l'étape TTM. Finalement, nous développons une implémentation hybride MPI-OpenMP de l'algorithme global et nous obtenons les facteurs d'accélération avec jusqu'à 2048 coeurs.

Mots-clés : tenseurs creux, factorizations des tenseurs creux d'ordre large

1 Introduction

Tensors or multi-dimensional arrays are used to represent data with higher dimensionality in many applications. Some of the most popular applications include analysis of Web graphs [27], knowledge bases [9], product reviews at online stores [8], chemometrics [2], signal processing [13], computer vision [37], forensics [31] and more. Tensor decomposition algorithms are used to find latent relations in the tensors. There are two prominent tensor decomposition formulations. CANDECOMP/PARAFAC (CP) formulation approximates a tensor as a sum of the product of rank-one tensors. *Tucker* formulation approximates a tensor with a core tensor multiplied by a matrix along each dimension or mode—see Fig. 1 for a simplistic view. Both of these formulations have uses in applications; in particular the CP formulation is deemed useful respectively for understanding latent factors, and the Tucker formulation is deemed more appropriate for compressions and identifying relation among the factors [21]. In this work, we investigate efficient computation of Tucker decomposition of sparse high dimensional tensors in common distributed memory systems.

There are variants of CP and Tucker decompositions, and different algorithms to compute them [12, 28]. The most common algorithms for both decompositions and their variants are based on the alternating least squares (ALS) method. The algorithms of this type are iterative, where the computational core of an iteration is a special operation among an N-mode tensor and N matrices. The key operation in the ALS-based CP decomposition (CP-ALS) case is called the matricized tensor times Khatri-Rao product (MTTKRP); we refer the reader to other resources for details [28]. The key operation in the ALS-based Tucker decomposition (Tucker-ALS) method is called the *tensor times matrix-chain* (TTMc) product. These two operations pose similar challenges for efficient computations. But there are distinct challenges and opportunities for efficiency as well.

The tensor times matrix-chain (TTMc) product in the Tucker-ALS method is performed for all modes of the N-mode input tensor at every ALS iteration. TTMc for a mode n involves tensor times matrix (TTM) products with $N-1$ different matrices, each of which is associated with one of the modes other than n. The TTM product can be considered as a higher dimensional variant of the matrix-vector product operation (Section 2 explains the TTM operation in detail). Techniques for efficiency of a single TTM are therefore akin to those in the matrix-vector case but require increased effort to overcome the difficulties associated with the higher dimensionality. The chain product TTMc adds another level of complexity; one needs to be careful about the memory requirements. Furthermore, one needs to take the content of an ALS iteration into account as N TTMc's will follow another.

The Tucker-ALS algorithm also computes a few singular vectors of a large, usually tall-and-skinny dense matrix for each mode at every ALS iteration. This matrix arises by a logical reorganization of the result of the TTMc associated with the corresponding mode. The cost of computing the singular vectors is not negligible and hence needs to be addressed in an efficient parallel algorithm for the Tucker decomposition. We refer to the computation of the desired singular vectors as the SVD-step.

Our contributions are as follows. First, we design efficient parallel algorithms for the TTMc operation. Here our analysis points that with proper task definitions, the computational and communication requirements are as in the MTTKRP operation. Having identified this, we make use of the hypergraph models from our earlier work on CP-ALS [24] for reducing communication and achieving load balance during each TTMc operation. Second, in order to increase the data reuse between the successive TTMc's, we propose a reformulation by generalizing exiting memory-

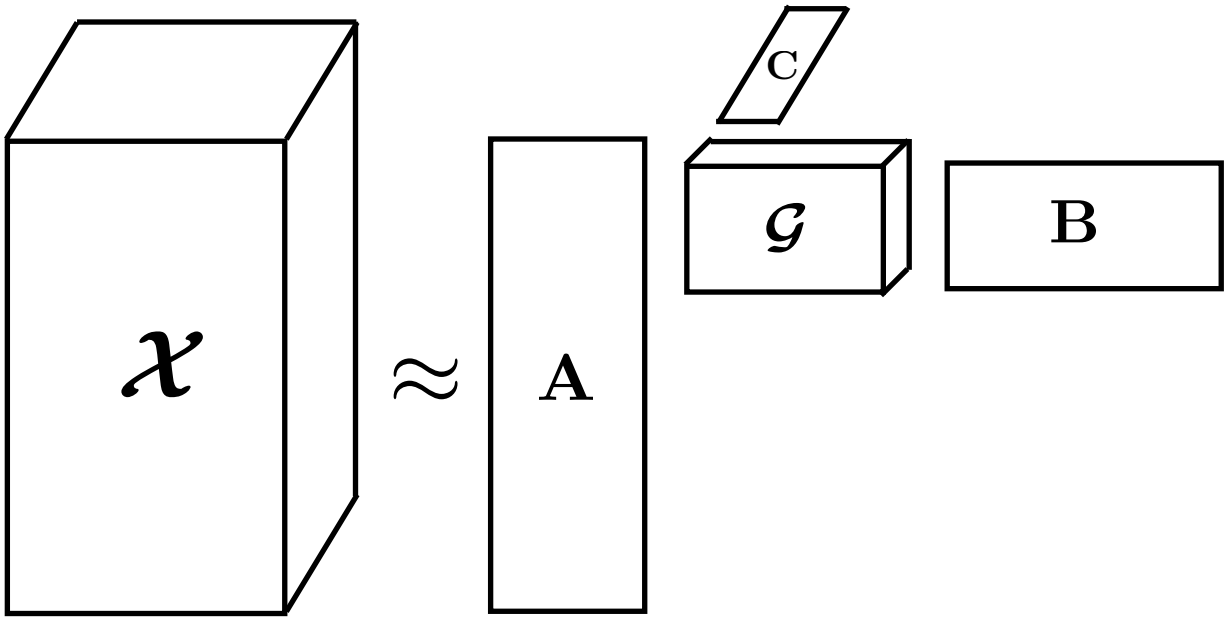


Figure 1 – Tucker decomposition of a 3rd mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ as a core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ multiplied by matrices $\mathbf{A} \in \mathbb{R}^{R_1 \times I_1}$, $\mathbf{B} \in \mathbb{R}^{R_2 \times I_2}$ and $\mathbf{C} \in \mathbb{R}^{R_3 \times I_3}$ in different modes. In CP-decomposition, $\mathcal{G}$ is a diagonal tensor having the same size along each dimension, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ have the same number of rows.

efficient implementations of TTMc's [6, 29]. This is of independent interest and can be used with dense or sparse tensors. Third, we propose efficient realization of the computation of the singular vectors by making use of existing libraries PETSc [5] and SLEPc [34]. We carefully designed this step so that the communication requirements in parallel iterative algorithms used for computing the singular vectors are reduced and the load balance is achieved by making use of the data decomposition of the TTMc's step. Fourth, we put every thing together and propose an OpenMP-MPI hybrid and complete implementation of Tucker-ALS algorithm and present speed-ups on a high-end parallel system where up to 742 speed-up on 2048 cores are reported on real world data.

The organization is as follows. We give background on the basic tensor operations in the next section. Then, in Section 3, we discuss the proposed parallel Tucker-ALS algorithm. Here, we first discuss the reorganization of TTMc's for efficient data reuse. We then discuss parallel TTMc's and SVD-steps. We then give a brief summary of related recent work in Section 4. Experimental results on a parallel system with compute nodes containing 32 cores are presented in Section 5. We conclude the paper with a few remarks in Section 6.

2 Background

We use bold, upper case Roman letters for matrices, as in $\mathbf{A}$. The matrix elements are shown with the corresponding lowercase letters, as in $a_{i,j}$. Matlab notation is used to refer to the entire rows and columns of a matrix, e.g., $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ refer to the ith row and jth column of $\mathbf{A}$ respectively.

We use calligraphic font to refer to tensors, e.g., $\mathcal{X}$. The *order* of a tensor is the number of its dimensions or modes, which we denote with N. For the sake of simplicity of the notation and the discussion, we sometimes discuss the case $N = 3$, even though our algorithms and implementations have no such restriction. We explicitly generalize the discussion to general order-N tensors whenever we find necessary. As in matrices, an element of a tensor is denoted by a lowercase letter and subscripts corresponding to the indices of the element, e.g., the element (i, j, k) of a third-order tensor is $x_{i,j,k}$. A *fiber* in a tensor is defined by fixing every index but one, e.g., if $\mathcal{X}$ is a third-order tensor, $\mathcal{X}_{:,j,k}$ is a mode-1 fiber and $\mathcal{X}_{i,j,:}$ is a mode-3 fiber.

We reproduce the following definitions from Kolda and Bader's survey [28]. The n-mode *matricization* of a tensor $\mathcal{X}$ is denoted by $\mathbf{X}_{(n)}$ and refers to the reordering of $\mathcal{X}$'s elements into a matrix by arranging the nth mode fibers as the columns of $\mathbf{X}_{(n)}$. For example, take $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. Then $\mathbf{X}_{(1)}$ denotes the mode-1 matricization of $\mathcal{X}$ where the rows of $\mathbf{X}_{(1)}$ correspond to the first mode of $\mathcal{X}$ and the columns correspond to the remaining modes. The tensor element $x_{i_1,\ldots,i_N}$ corresponds to the element $\left(i_1, 1 + \sum_{j=2}^{N} \left[(i_j - 1) \prod_{k=1}^{j-1} I_k\right]\right)$ of $\mathbf{X}_{(1)}$. Specifically, each column of the matrix $\mathbf{X}_{(1)}$ is a mode-1 fiber of the tensor $\mathcal{X}$.

The *n-mode product* of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$. This is also referred to as tensor times matrix (TTM) product. The result $\mathcal{Y}$ is a tensor of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$. A particular element of $\mathcal{Y}$ is given by

$$y_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} x_{i_1,i_2,\ldots,i_N} u_{j i_n} \ .$$

A tensor can be multiplied by a set of matrices along a given set S of modes. We use the notation $\text{TTMc}(\mathcal{X}, S, \{\mathbf{U}_n : \text{for } n \in S\})$ to refer to the tensor n-mode product of $\mathcal{X}$ with matrices $\mathbf{U}_n$ for $n \in S$. We use $\text{TTMc}(S)$ for clarity, as the tensor $\mathcal{X}$ and the matrices $\mathbf{U}_n$'s will be clear from the context.

The Tucker decomposition expresses a given tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as a core tensor $\mathcal{G}$ multiplied by a factor matrix $\mathbf{U}_n$ of size $I_n \times R_n$ at each mode n. Here, $R_1, \ldots, R_N$ are the requested rank of the decomposition for each mode. Formally, the Tucker decomposition $[\![\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_N]\!]$ writes $\mathcal{X}$ as $\mathcal{G} \times_1 \mathbf{U}_1 \times_2 \cdots \times_N \mathbf{U}_N$. For example, if $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, then its Tucker decomposition $[\![\mathcal{G}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$ writes $x_{i,j,k} \approx \sum_{p=1}^{I_1} \sum_{q=1}^{I_2} \sum_{r=1}^{I_3} g_{pqr} a_{ip} b_{jq} c_{kr}$.

A well-known algorithm for computing the Tucker decomposition is based on the alternating least squares method which is given in Algorithm 1, which is also called higher-order orthogonal iteration (HOOI) [14]. In this algorithm, the factor matrices are initialized first. This initialization could be done randomly or using the higher-order SVD [14]. Then, the alternating least squares method is applied in the "repeat-until" loop. Here, for each mode n, the tensor matrix-chain multiplication $\text{TTMc}(\{1, \ldots, N\} \setminus \{n\})$ at Line 1 is computed. This yields a tensor of size $R_1 \times R_2 \times \cdots R_{n-1} \times I_n \times R_{n+1} \times \cdots \times R_N$, which is then matricized along the nth mode yielding $\mathbf{Y}_{(n)} \in \mathbb{R}^{I_n \times \Pi_{i \neq n} R_i}$, and the left R_n leading singular vectors of $\mathbf{Y}_{(n)}$ are computed at Line 2 and used as the columns of $\mathbf{U}_n$.

The algorithms for computing the singular vectors are well documented [16, Ch. 8.6]. The TTMc operation at Line 1 needs attention. This is because of the fact that there are many ways to compute the product [29], as the TTM's can be performed in any order [28]. One extreme, which is called the standard way, is to compute one n-mode product at a time, e.g., for a 3rd order

Algorithm 1: Tucker-ALS for N-mode tensors

Input : $\mathcal{X}$: An N-mode tensor
$R_1, \ldots, R_N$:
The rank of the decomposition for each mode
Output: Tucker decomposition $[\![\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_N]\!]$

Initialize the matrix $\mathbf{U}_n \in \mathbb{R}^{I_n \times R_n}$ for $n = 1, \ldots, N$
repeat
  for $n = 1, \ldots, N$ **do**
1 $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{U}_1^T \cdots \times_{n-1} \mathbf{U}_{n-1}^T \times_{n+1} \mathbf{U}_{n+1}^T \cdots \times_N \mathbf{U}_N^T$
2 $\mathbf{U}_n \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
until *no improvement or maximum iterations reached*
$\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{U}_1^T \cdots \times_N \mathbf{U}_N^T$
return $[\![\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_N]\!]$

tensor and $n = 1$,

$$\mathcal{Y} = ((\mathcal{X} \times_2 \mathbf{U}_2^T) \times_3 \mathbf{U}_3^T) \,. \tag{1}$$

The other extreme, which is called the element-wise, is to compute the fibers of $\mathcal{Y}$ at a time, e.g., with the same example as above,

$$y_{:,j,k} = \mathcal{X} \times_2 \mathbf{U}_2(j,:) \times_3 \mathbf{U}_3(k,,:) \,, \tag{2}$$

for $j = 1, \ldots I_2$ and $k = 1, \ldots, I_3$. In the element-wise computation (2), the n-mode products are with the row-vectors of the factor matrices, where the unit size in the nth mode of the result is not indexed. Kolda and Sun also discuss schemes for higher order tensors, where some of the modes are treated element-wise and some others in the standard way and present formulas to decide this for a given tensor.

We re-write the elementary-way (2) in Algorithm 2, as it better suits for our presentation. In this algorithm, $\mathbf{U}_2(j,:) \circ \mathbf{U}_3(k,:)$ refers to the outer product of the vectors $\mathbf{U}_2(j,:)^T$ and $\mathbf{U}_3(k,:)$.

Algorithm 2: Elementary-way of multiplication in TTMc for 3rd order tensors and $n = 1$

foreach $x_{i,j,k} \in \mathcal{X}$ **do**
  $\mathcal{Y}(i,:,:) \leftarrow \mathcal{Y}(i,:,:) + x_{i,j,k} \, (\mathbf{U}_2(j,:) \circ \mathbf{U}_3(k,:))$

3 Parallel Tucker Factorization

Here we propose a computational scheme to compute a set of tensor times matrix chain products efficiently. We then discuss our parallelization approach for TTMc's and for computing the singular vectors.

3.1 TTM-tree

In an iteration of the Tucker-ALS, N TTMc's are performed one after another, each one for a mode potentially having overlaps. For example, TTMc($\{1, \ldots N\} \setminus \{N\}$) and TTMc($\{1, \ldots N\} \setminus \{N-1\}$)

have $N-2$ TTM's in common. This has been exploited before [6] to increase data reuse (see our discussion in Section 4). Here, we generalize this idea and increase the data reuse considerably.

We propose a computational scheme that we call *TTM-tree*. This tree represents N TTMc's compactly and enable the re-use of the common TTM's. Each node in a TTM-tree corresponds to a tensor with a set of *multiplied* and a set of *unmultiplied* modes which partition $\{1,\ldots,N\}$. The tensor associated with a tree node corresponds to the result of the TTMc of the input tensor $\mathcal{X}$ with the node's multiplied modes. A TTM-tree always has $\mathcal{X}$ as its root, with an empty set of multiplied modes. Starting from the root, the unmultiplied modes of the children of each node *partition* the node's unmultiplied modes. A TTM-tree has exactly N leaves where the n^{th} leaf has only the mode n in its unmultiplied mode list; its result is TTMc($\{1,\ldots,N\} \setminus \{n\}$) and hence the tensor $\mathcal{Y}$ in Line 1 of Algorithm 1. We note that the TTM-tree is a general computational scheme that can be applied to both sparse and dense Tucker computations.

We perform two types of operations on a TTM-tree while computing TTMc($\{1,\ldots,N\} \setminus \{n\}$) for all n in an iteration of Tucker-ALS. In an *update* operation on a particular tree node, we update the TTMc result stored in the node by performing relevant TTM operations along the tree. In doing so, we first perform an update operation on the parent in case its result is not computed, or needs to be re-computed due to a recent update to a matrix $\mathbf{U}_k$ in one of its multiplied modes k. Note that the parent might also recursively trigger another update on its parent; thereby, all results for the nodes from the leaf to the root might potentially be updated. After ensuring that the parent's result is valid, we perform a TTMc with the parent's tensor and the relevant "missing" multiplied modes of the child node to update the tensor at the child node. When $\mathbf{U}_n$ is updated, we perform an *invalidate* operation. This operation frees all tensors at tree nodes whose value depends on the old value of $\mathbf{U}_n$, and mark those nodes as invalid. Note that these are exactly the nodes who have n in their multiplied modes.

We investigate two types of TTM-trees we call **flat** and **binary**. A *flat* TTM-tree consists of the root and the leaves, with no intermediate result. This corresponds to the "element-wise" product (2). This approach minimizes the memory consumption at the cost of eliminating the potential reuse of intermediate results. In a *binary* TTM-tree, starting from the root, every node has two children, except the leaves. Unmultiplied modes of a node is divided into left and right halves, and the children are obtained by performing TTM in these modes.

We note the following two theorems whose relatively straightforward proofs are in the accompanying technical report [25].

Theorem 1. *Let $\mathcal{X}$ be a tensor with $N = 2^k$ modes, where $k \in \mathbb{Z}_{\geq 0}$. The total number of n-mode matrix multiplications in each iteration of the Tucker-ALS algorithm using binary TTM-tree is $N \log_2 N$.*

Proof. In each iteration of the Tucker-ALS, at the k^{th} step, the binary TTM-tree updates its k^{th} leaf, then invalidates all its nodes whose set of multiplied modes involves k. Note that by the construction of the binary TTM-tree, each ancestor of the leaf node k involves k in its set of unmultiplied nodes; hence, it does not get invalidated by the update to k, nor to any of its other descendants. Also, descendant leaves of each interior *parent* node are consecutive, as the unmultiplied modes of each node consists of a set of consecutive modes, which is implied by the construction procedure described above. Therefore, as the modes are updated from 1 to N within a Tucker-ALS iteration, the *parent* node gets updated when its first descendant leaf is updated, retains a valid result while updates are performed on all its consecutive descendant leaves, then

gets invalidated and never needed for the remaining modes; because, the leaf nodes corresponding to the remaining modes are not descendants of the *parent* node, and they neither incur an update operation on it nor need its result. As a result, in each iteration of Tucker-ALS, each internal node is updated exactly once. Similarly, binary TTM-tree updates each leaf node once per iteration in the main Tucker-ALS loop.

The number of n-mode matrix multiplications to be performed in each mode's update in a TTM-tree equals to the number of missing unmultiplied modes with respect to its parent. Note that in the construction of the binary TTM-tree, each node partitions its unmultiplied modes into two halves, each of which constitutes the set of unmultiplied modes of its children. Therefore, each child node misses the other half of the unmultiplied modes of its parent, hence the number of missing unmultiplied modes equals to the cardinality of the childs set of unmultiplied modes.

After these two observations, we can finalize the proof. In a Tucker-ALS iteration, each node gets updated once, and performs n-mode matrix product per its missing unmultiplied mode with respect to its parent. Note that by the construction of a binary TTM-tree, the unmultiplied modes of the nodes in its each level partition the set $\{1, \ldots, N\}$; therefore, the total number of n-mode matrix multiplications performed per level is N. The binary TTM-tree has $k = log_2 N$ levels; therefore, the total number of n-mode matrix multiplications performed within a Tucker-ALS iteration is $N log_2 N$. □

Theorem 2. *Let $\mathcal{X}$ be a tensor with $N = 2^k$ modes, where $k \in \mathbb{Z}_{\geq 0}$. Then, in any iteration of the Tucker-ALS algorithm, the number of nodes with valid results in the binary TTM-tree is at most $2 \log_2 N$.*

Proof. At the k^{th} step of an iteration of the Tucker-ALS, after the invalidate is performed, the results of all nodes in the binary TTM-tree whose multiplied mode involves k (or equivalently, unmultiplied modes do not involve k) are invalidated and destroyed. As stated in the Theorem 1, the unmultiplied modes of the nodes at each level of the binary TTM-tree partition the set $\{1, \ldots, N\}$; therefore, only one node at each level involves k in its unmultiplied modes, hence do not get invalidated. Hence, after the invalidate is performed at any step k, the number of nodes whose results remain valid is $k = log_2 N$. Note also that an update operation can generate at most $k = log_2 N$ results in the binary TTM-tree; therefore, at any instant of Tucker-ALS, the total number of nodes with valid results in the binary TTM-tree cannot exceed $2 log_2 N$. □

In contrast to the binary TTM-tree, while using the flat TTM-tree, one needs to perform TTM's for $N - 1$ matrices in each mode, which results in a total of $N(N - 1)$ TTM's in one iteration of the Tucker-ALS algorithm. Therefore, Theorem 1 shows that the binary TTM-tree asymptotically reduces the number of TTM's, which becomes particularly effective for higher-order tensors. The disadvantage is the required memory to store the intermediate results; Theorem 2 puts a nice upper bound on the maximum number of intermediate results at any instant of the algorithm.

3.2 TTM-tree and symbolic TTM with semi-dense tensors

We use the semi-dense tensor data structure proposed by Baskaran et al. [6] for storing tensors in the TTM-tree. A semi-dense tensor consists of sparse and dense modes, namely *sm* and *dm*, which partition $\{1, \ldots, N\}$. Each entry of the semi-dense tensor consists of coordinates in sparse modes stored in *idx*, and a value which is a dense tensor of order $|dm|$. A semi-dense tensor contains

symbolic information such as sparse indices of elements and the set of sparse and dense modes that would remain the same through multiple TTM calls, particularly when the tensor corresponds to a node in a TTM-tree. Therefore, it is viable to precompute and store all such symbolic information before the main Tucker-ALS iteration for efficiency. For this reason, we introduce Algorithm 3, which computes all the symbolic information needed for the numerical TTM's when executed within Tucker-ALS for many iterations.

In the symbolic TTM, from an input semi-dense tensor $\mathcal{X}_p$, which we call as the *parent*, we create the *child* semi-dense tensor $\mathcal{X}_c$ holding the symbolic information for the numerical TTM of $\mathcal{X}_p$ in the given *modes*. We first set the dense and sparse modes of $\mathcal{X}_c$ in Lines 1 and 2. Note that dense modes of $\mathcal{X}_c$ consist of the dense modes of $\mathcal{X}_p$ and the *modes*, which correspond to the sparse modes of $\mathcal{X}_p$ that become dense in $\mathcal{X}_c$ as a result of TTM. In Line 3, the nonzero entries of $\mathcal{X}_c$ are determined by collapsing the parent's *idx* array in the columns corresponding to the set of *modes*, which will be dense in $\mathcal{X}_c$, and then eliminating duplicates. Finally in Line 4, for each element i of the parent $\mathcal{X}_p$, we find the element j of $\mathcal{X}_c$ to which it makes a contribution in the numerical TTM, and add it to the reduction list $\mathcal{X}_c.rl(i)$. Thereby, we can easily perform the numerical TTM by going over the parent's elements in this update list, performing their outer product with the corresponding rows of the matrices, and sum-reducing the results in a similar fashin to the scheme provided in Algorithm 2. Note that the sparse modes of a semi-dense tensor at a node of the TTM-tree are the unmultiplied modes, whereas its dense modes are the multiplied modes. All numerical computation later is performed using the update and invalidate operations. We do not provide further details for the numerical multiplication.

Algorithm 3: Symbolic TTM for constructing semi-dense tensor $\mathcal{X}_c$ at a child node from $\mathcal{X}_p$ at a parent node in the TTM-tree.

Input : $\mathcal{X}_p$: An N-mode semi-dense tensor
modes: List of sparse modes of $\mathcal{X}_p$ to perform TTM
Output: $\mathcal{X}_c$: The result of the TTM of $\mathcal{X}_p$ in modes *modes*

1 $\mathcal{X}_c.dm \leftarrow \mathcal{X}_p.dm \cup modes$
2 $\mathcal{X}_c.sm \leftarrow \{1, \ldots, N\} \setminus \mathcal{X}_c.dm$
3 $\{\mathcal{X}_c.idx, \mathcal{X}_c.nz\} \leftarrow collapse(\mathcal{X}_p.idx(:, \mathcal{X}_c.sm))$
for $i = 1, \ldots, \mathcal{X}_p.nz$ **do**
4 $\quad j \leftarrow findIndex(\mathcal{X}_c.idx, \mathcal{X}_p.idx(i, \mathcal{X}_c.sm))$
5 $\quad \mathcal{X}_c.rl(j) \leftarrow \mathcal{X}_c.rl(j) \cup \{i\}$
return $\mathcal{X}_c$

3.3 Parallelization

In Algorithm 4, we provide the high-level description of the parallel Tucker-ALS algorithm at an MPI-rank p. The first step of the algorithm is to form the TTM-tree for the input tensor $\mathcal{X}_p$. At the beginning of each iteration, we assume that p has all the rows of $\mathbf{U}$ matrices that are needed to perform the TTMc operations. Having all needed data available locally, for each mode n, p performs the *update* operation on the leaf tensor which corresponds to rows of $\mathbf{Y}_{(n)}$ owned by the MPI-rank p. In the next step, SVD is performed on the row-wise distributed matrix to obtain the new values of $\mathbf{U}_n$. Then, the updated rows of the $\mathbf{U}_n$ are communicated to make the needed data available. Finally, TTM-tree invalidates and deallocates all its results that become outdated after

Algorithm 4: Parallel TTM-tree based Tucker-ALS

Input : $\mathcal{X}$: An N-mode tensor
$R_1, \ldots, R_N$:
The rank of the decomposition for each mode
Output: Tucker decomposition $[\![\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_N]\!]$
$tree \leftarrow formTTMtree(\mathcal{X})$
repeat
  for $n = 1, \ldots, N$ **do**
    $tree.update(tree.leaf(n))$
    $\mathbf{U}_n \leftarrow R_n$ leading left sing. vectors of $tree.leaf(n)$
    Send/receive the updated rows of $\mathbf{U}_n$ $tree.invalidate(n)$
until *no improvement or maximum iterations reached*;
$\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{U}_1^T \cdots \times_N \mathbf{U}_N^T$
return $[\![\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_N]\!]$

$\mathbf{U}_n$ changes. While performing updates numerically for a child $\mathcal{X}_c$, the $i$the element is updated using the reduction lists $\mathcal{X}_c.rl(i)$, the destinations are disjoint and hence parallel loops can be used easily to achieve node parallelism.

Note that after the TTMc operation, the resulting $\mathcal{Y}$ tensor is matricized into

$$\mathbf{Y}_{(n)} \in \mathbb{R}^{I_n \times (R_1 \times \cdots R_{n-1} \times R_{n+1} \times \cdots \times R_N)}$$

whose dimensions can be large. Therefore a parallel algorithm should be run for computing the singular vectors. SLEPc [18] provides required algorithms. Among different alternatives [34, Ch. 4.], we chose the SVD solver based on the Lanczos bidiagonalization [19, 20]. This method is iterative where the computational core of each iteration is the matrix-vector (MxV) and matrix transpose-vector (MtxV) multiplications (there are some other vector operations as well) with $\mathbf{Y}_{(n)}$. By making use of PETSc's shell matrix type [5], SLEPc offers implementation choices where the user provides MxV and MtxV multiplication subroutines to be called within the SVD solvers. We parallelize the local MxV and MtxV operations at a node level by relying on the multithreaded `gemv` implementation in MKL. There were a few particularities to reduce the communication in the SVD solver which we discuss below at each task definition.

For the sake of the simplicity, some of the task definitions are for 3-mode tensors; however, they naturally generalize to higher dimensions.

3.3.1 Coarse-grain task definition

In an iteration of the Tucker-ALS, we perform two main operations for each mode n: a TTMc step to obtain a matricized tensor $\mathbf{Y}_{(n)}$, and an SVD step to obtain the matrix $\mathbf{U}_n$ once $\mathbf{Y}_{(n)}$ is ready.

In the coarse-grain task decomposition, we define computing each row i of $\mathbf{U}_n$ as an atomic task, and hold the owner of this task responsible for computing the i^{th} row of $\mathbf{Y}_{(n)}$. We denote this task by t_i^n.

For $n = 1$, as provided in the Algorithm 2, a row i of the matricized tensor $\mathbf{Y}_{(n)}$ receives a contribution $x_{i,j,k}\,(\mathbf{U}_2(j,:) \circ \mathbf{U}_3(k,:))$ for each nonzero $x_{i,j,k}$. Therefore, t_i^1 needs all nonzeros in the tensor slice $\mathcal{X}(i,:,:)$, as well as the rows $\mathbf{U}_2(j,:)$ and $\mathbf{U}_3(k,:)$ to perform the outer product. Therefore, t_i^1 may require up to $|\mathcal{X}(i,:,:)|$ outer products for completion (with a flat TTM-tree). Also, for each nonzero $x_{i,j,k}$, t_i^1 needs the data owned by t_j^2 and t_k^3, which defines the data dependencies.

In our recent work [24], we used a similar task definition in the context of CP-ALS, which we naturally represented with a hypergraph model. We can use the same model to reduce the total communication volume and to achieve load balance during the TTM computations. There is a difference though. The computational weights $|\mathcal{X}(i,:,:)|$, $|\mathcal{X}(:,j,:)|$, and $|\mathcal{X}(:,:,k)|$ for the tasks t_i^1, t_j^2, and t_k^3, respectively are exact with a flat TTM-tree, because each nonzero performs an outer product with $N-1$ vectors. If a binary TTM-tree is used, a nonzero may perform an outer product in all $N-1$ modes, or a partial outer product which gets merged with another partial outer product in an intermediate semi-dense tensor. Therefore, in this scheme, the weight becomes an upper-bound on the actual computational load.

At the end of the coarse grain TTMc, each MPI-rank has the entirety of the rows of $\mathbf{Y}_{(n)}$ corresponding to its n-mode indices. The MxV multiplication $y \leftarrow \mathbf{Y}_{(n)}x$ can be easily implemented by gathering all x entries to all processors as $\mathbf{Y}_{(n)}$ is a dense matrix. The partition on y induces a partition on the left singular vectors of $\mathbf{Y}_{(n)}$ and hence on the rows of the new $\mathbf{U}_n$. We aligned the partition on y with that of the rows of $\mathbf{U}_n$ so that the owner of $\mathbf{U}_n(i,:)$ holds y_i. The MtxV $x^T \leftarrow y^T\mathbf{Y}_{(n)}$ can be implemented by an all-to-all communication on the partial results of local $y^T\mathbf{Y}_{(n)}$ computations. If the MPI-ranks have almost equal number of n-mode indices, the computational load would be balanced. In our current implementation, the n-mode indices are distributed for the TTMc step among the MPI-ranks with hypergraph models to achieve load balance in terms of the number of nonzero tensor elements. This is likely to result in load balance during the MxV and MtxV operations in the SVD step.

3.3.2 Fine-grain task definition

We have the task u_i^n of computing and owning the row i of $\mathbf{U}_n$ as in the coarse grain formulation. However, in this task definition we use a finer granularity in the TTMc operations, and define the computation of each outer product $x_{i,j,k}\,(\mathbf{U}_2(j,:) \circ \mathbf{U}_3(k,:))$ as the atomic task, which we denote as $z_{i,j,k}$.

For $n = 1$, each process which owns a task $z_{i,j,k}$ generates a partial result for $\mathbf{Y}_{(1)}$. One might consider sending these partial results to the owner of the task u_i^1; however, we avoid this communication because of the fact that each row of the matrix $\mathbf{Y}_{(1)}$ has $R_2 \times R_3$ columns, which may get quite costly even for $N = 3$. For $N > 3$, this number becomes $\prod_{i\neq 1} R_i$, which can easily get large for high order tensors and/or ranks of approximation. Therefore, we keep the partial results in the sum-distributed form across many processors and handle this difficultly during computing singular vectors.

Modeling the tasks and their dependencies yields the same fine grain hypergraph model we proposed before [24]. Therefore, we can reduce the total communication volume and achieve load balance during the TTM computations by making use of the results from the previous study. This hypergraph model also useful in assigning almost equal number of mode indices per MPI-rank for each mode. Note that not combining the partial results of $\mathbf{Y}_{(n)}$ can increase the total computational load of matrix-vector multiplications in the SVD step. This increase is equal to the $\lambda - 1$ cut size metric of the corresponding hypergraph and hence is reduced. That is why the previously proposed hypergraph model is also useful for the SVD step.

At the end of the fine grain TTMc, each MPI-rank can have partial results on many nonzero elements in $\mathcal{Y}$ according to the distribution of the nonzeros of $\mathcal{X}$. In other words, MPI-ranks share some of the rows of $\mathbf{Y}_{(n)}$ where the exact value of the elements in a row can only be computed

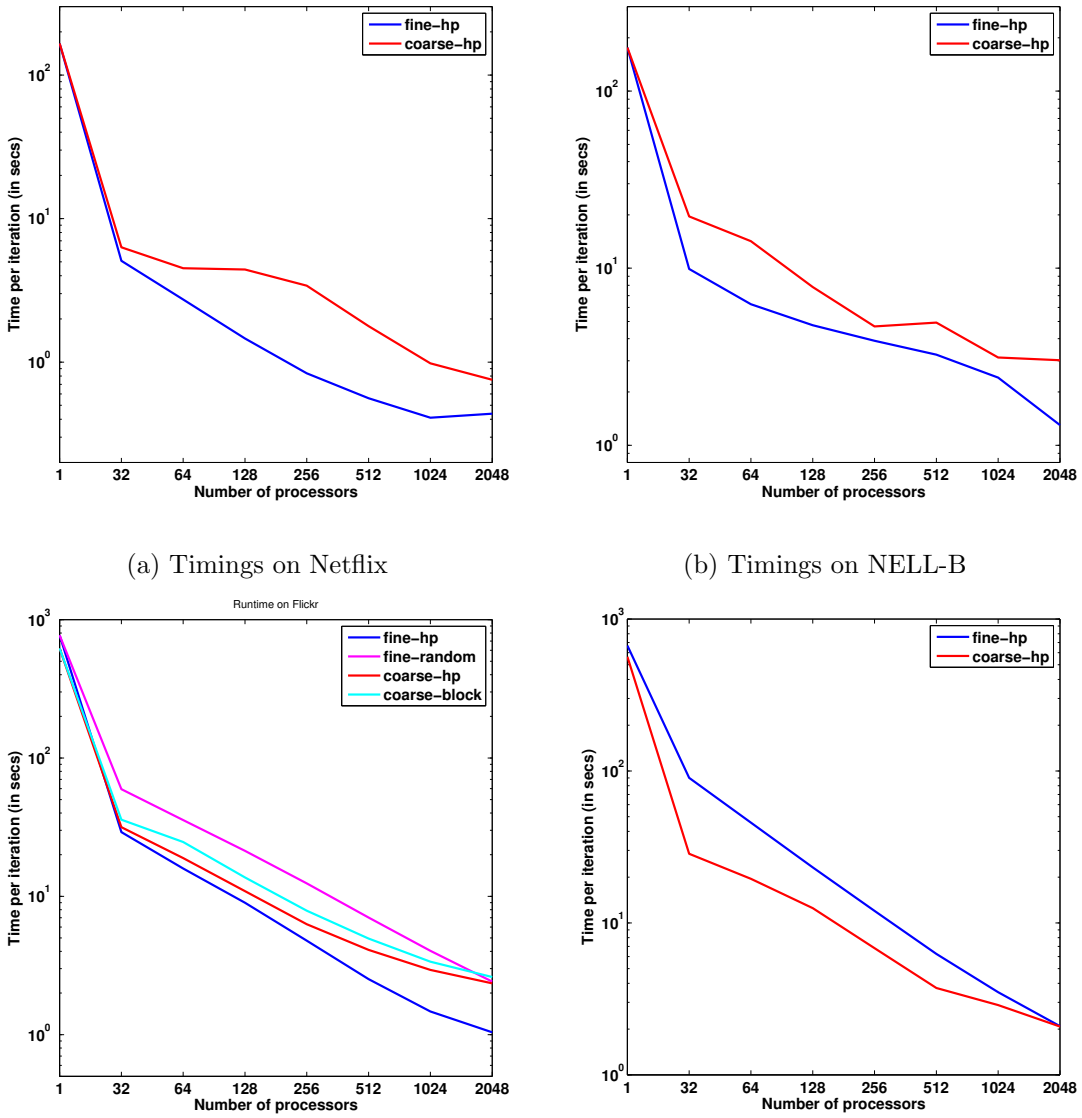
if the shared rows are added up. Since the MxVs does not really need the value of the elements of $\mathcal{Y}$ but $\mathbf{Y}_{(n)}$'s action on a vector, we do not accumulate those partial results on $\mathcal{Y}$. Rather, we implement the MxV $y \leftarrow \mathbf{Y}_{(n)}x$ for this case by reducing the partial results on y. The amount of communication on y is closely related to the communication in the fine-grain TTMc operations. Assuming a common decomposition rank of R for each mode, the total communication volume for reducing y is equal to $1/R$ times the total volume of communication in the TTMc, if one again aligns the partition on y with the n-mode index partition. Note that if instead one builds the exact values of $\mathbf{Y}_{(n)}$, then the same amount of communication as in the TTMc will be required. A dual communication scheme is necessary for computing the MtxVs. As in the coarse grain algorithm, the load balance in the local MxV and MtxV operations can be achieved if the MPI-ranks have equal number of rows in $\mathbf{Y}_{(n)}$. This is one of the complex partitioning problems where the total computational load can only be determined after a partition [23, 33]. We do not explicitly address this problem and hope that assigning equal amount of n-mode indices will again lead to load balance. We note that the number of rows of $\mathbf{Y}_{(n)}$ in which many MPI-ranks have contributions is equal to the total communication volume for reducing y, and hence again is a linear function of the TTMc's volume of communication.

4 Related work

We give a brief overview of mostly recent progress on efficient tensor decomposition algorithms. These can be categorized into four classes: (i) toolboxes for Matlab and similar environments [1, 3, 4, 29, 32]; (ii) implementations for shared memory systems [6, 7, 30, 36]; (iii) implementations based on MapReduce paradigm [21, 22]; (iv) distributed memory systems [11, 24, 35]. The implementations in the first group are very useful tools that enable fast prototyping. Those in the second group and similar work are helpful when data fits into the memory of a single machine, which is nowadays large enough to accommodate tensors from many applications. Those in the third and fourth groups enable computations on tensors that do not fit into the memory. The ones in the third group are not designed for high performance, as MapReduce paradigm is meant to perform multiple passes over out-of-core data and perform global communication shuffling the input data.

The the best of our knowledge, there is no high-performance distributed memory implementation of algorithms for the sparse Tucker decomposition. Among the cited references above, Haten2 [21] is a MapReduce based Tucker-ALS implementation. Li et al. [30] investigate efficient shared memory execution of tensor times matrix products and as a future work mention how this can be used to perform intra-node TTM computations in a distributed memory setting.

Our data structures and the proposed reorganization of the repeated TTMc's are based on two of the cited work [6, 29]. In particular, the different tensor times matrix product schemes of Kolda and Sun is used in building the TTM-tree. We make use of the data structures of Baskaran et al. [6] to represent the tensors at the nodes of the TTM-trees. Baskaran et al. also discuss a simple yet very effective scheme to increase the data reuse. This scheme computes half of the tensor matrix-chain multiplies in increasing order of modes, and the other in decreasing order of modes. While computing the first half, the factor matrices corresponding to the second half are not modified and hence one can accumulate and re-use them. A similar observation holds for the second half. Our TTM-tree representation takes this scheme one step ahead and increase data reuse at the expense of extra storage.



(a) Timings on Netflix

(b) Timings on NELL-B

(c) Timings on Flickr

(d) Timings on Delicious

Figure 2 – Time for parallel Tucker-ALS iteration on Netflix and NELL-B, and speedups on Flickr and Delicious.

Table 1 – Size of tensors used in the experiments

Tensor	I_1	I_2	I_3	I_4	#nonzeros
Netflix	480K	17K	2K	-	100M
NELL-B	3.2M	301	638K	-	78M
Delicious	1.4K	532K	17M	2.4M	140M
Flickr	731	319K	28M	1.6M	112M

5 Experimental Results

We conducted our experiments on the IDRIS Ada cluster, which consists of 304 nodes each having 128 GBs of memory and four 8-core Intel Sandy Bridge E5-4650 processors running at 2.7 GHz. We ran our experiments up to 2048 cores (64 nodes), which is the maximum allowed in the cluster. We tried different configurations of 32 cores within a node for the MPI/OpenMP rank/thread assignments, and found the assignment of 4 MPI ranks (and 8 OpenMP threads per MPI-rank) per node to give the best results. All codes we used in our benchmarks were compiled using the Intel C++ compiler (version 14.0.1.106) with -O3 option for compiler optimizations, -openmp flag to enable OpenMP, and -mkl option to use the Intel MKL library (version 11.1.1) for LAPACK, and BLAS routines. To obtain the sequential runtimes, we allocated 1 MPI rank with 1 threads, disabled the multithreading within MKL, and used a high-memory nodes in the cluster having 256 GB memory.

We used PaToH [10] (version 3.2) with default options to partition the hypergraphs that we formed. We created all partitions offline, and ran our experiments on these partitioned tensors on the cluster. We do not report timings for partitioning hypergraphs with PaToH, which is quite costly, for two reasons. First, in most applications, the tensors from the real-world data are built incrementally; hence, a partition for the updated tensor can be formed by refining the partition of the previous tensor. Also, one can decompose a tensor multiple times with different ranks of approximation [26], potentially amortizing the time spent in partitioning across multiple runs. Second, we had to partition the tensors on a system different from the one used for the experiments. It would not be very useful to compare the runtimes from different systems and repeating the same runs on different system would not add much to the presented results. We also note that the proposed algorithms are independent from the partitioning method, and hence any hypergraph partitioning method can be used.

We used the same four tensors as in previous work [24]. These tensors are from real world data. We provide the dimensions of these tensors in Table 1. Netflix tensor has user $\times$ movie $\times$ time dimensions, which we formed from the Netflix Prize competition [8]. In this tensor, the nonzeros correspond to the user reviews of movies, and review date extends the data to the third dimension. The values of the nonzeros are determined by the corresponding review scores given by the users. We obtained The NELL-B tensor from the Never Ending Language Learning (NELL) knowledge database of the "Read the Web" project [9], which consists of tuples of the form $(entity, relation, entity)$ such as ('Chopin','plays musical instrument','piano'). The nonzeros of this tensor correspond to these entries discovered by NELL from the web, and the values are set to be the "belief" scores given by the learning algorithms used in NELL. Delicious and Flickr are the datasets for the web-crawl of Delicious.com and Flickr.com during 2006 and 2007, which is formed by Görlitz et al. [17]. These datasets consist of tuples of the form (users $\times$ resources $\times$ tags $\times$ time); hence we naturally form 4-mode tensors out of these tuples.

Table 2 – Comparison of the sequential and parallel run times (in secs) with fine-hp using flat and binary TTM-trees

TTM-tree/#procs	Netflix	NELL-B	Delicious	Flickr
flat/1	166	176	669	772
btree/1	123	129	560	678
flat/2048	0.44	1.3	2.09	1.04
btree/2048	0.39	1.2	0.97	0.90

In our experiments, we set the rank of approximation $R_1 = R_2 = R_3 = 10$ for 3-mode tensors, and $R_1 = R_2 = R_3 = R_4 = 5$ for the 4-mode tensors, ran the Tucker-ALS for 10 iteration, and reported the average time spent per Tucker-ALS iteration. We first show the effects of flat TTM-tree and the binary TTM-tree. Then we investigate the scalability.

Table 2 gives the actual runtimes (in seconds) with both flat and binary TTM-trees for comparison. The choice of TTM-tree only pertains to the local TTM operations and is independent from the distributed memory parallelism; hence we provide the runtimes for the sequential as well as parallel execution times over 2048 cores of both schemes. As seen in this table, binary TTM-tree improves the runtime for all tensors during both sequential and 2048-way parallel executions. In the sequential execution, using binary TTM-tree reduced the overall execution by %27, %26, %17, and %13 for NELL-B, Netflix, Delicious, and Flickr, respectively. Similarly in the parallel execution, the overall time for using the binary TTM-tree was reduced by %8, %12, %54, and %14, respectively. These results demonstrate that even for 3 and 4-mode tensors binary TTM-tree can significantly reduce the operation count by utilizing the overlaps in the partial results of TTMS. We expect this benefit to increase even further for higher dimensional tensors.

We investigate the scalability of the proposed algorithms using flat TTM-trees in order to be able provide runtime results for runs with smaller number of processors, for which memory availability becomes a problem for the binary TTM-tree scheme. In Figure 2 we give the scalability results of our algorithm using different partitions in our dataset. For all tensors, we report the results for fine-hp and coarse-hp, which stand for using the hypergraph partitioning on fine-grain and coarse-grain task definitions, respectively. Additionally, to evaluate the hypergraph partitioning, on the Flickr tensor we provide results for random partitioning of tasks in fine-grained task scheme, which we denoted as fine-random, and block partitioning of the tasks corresponding to the consecutive rows of the $\mathbf{U}$ matrices, which we called coarse-block.

In our results, we first observe in Figure 2 that in all test cases our methods could scale up to 2048 cores (or 256 MPI ranks), except for fine-hp on Netflix tensor, for which we observed a slight slowdown at 2048 cores. Using the fine-hp, we were able to obtain speedups values 742x, 377x, 320x, and 135x for Flickr, Netflix, Delicious, and NELL-B tensors, respectively. This is a very promising result, particularly considering the fact that on the same dataset, in our recent work in the CP-ALS setting [24], we could only scale to 512 cores. We observed two reasons for this relative improvement in scalability. First, our CP-ALS implementation did not employ any shared-memory parallelism, and assigned one MPI thread per core. This resulted in a high number of messages that needed to be sent/received by each MPI rank; as a result, the communication latency became a bottleneck beyond 512 cores. Here we also employ shared memory parallelism, and therefore use significantly less number of MPI-threads per node (4 vs 32), which alleviates this problem simply by employing less MPI ranks. Second, the communication pattern and the volume of CP-ALS and Tucker-ALS iterations are similar; however, Tucker-ALS involves significantly more computational

Table 3 – Statistics for the computation and communication requirements with different partitionings of the Flickr tensor in one Tucker-ALS iteration for 2048-way parallel run with 256 MPI ranks.

Mode	Comp. load		Comm. volume		Num. msg.	
	nzmax	nrmax	Max	Avg	Max	Avg
fine-hp						
1	441K	590	2218	2029	510	509
2	441K	8778	24K	17K	507	491
3	441K	221K	166K	11K	265	109
4	441K	32K	77K	53K	510	508
fine-random						
1	443K	668	5884	2597	490	464
2	443K	98K	409K	385K	510	510
3	443K	545K	1744K	1735K	510	510
4	443K	110K	432K	413K	510	510
coarse-hp						
1	718K	22	4910	1213	255	254
2	810K	2700	118K	66K	255	255
3	798K	197K	3187K	810K	255	255
4	2368K	13K	170K	102K	255	255
coarse-block						
1	958K	252	18K	907	220	162
2	756K	5401	126K	80K	254	252
3	441K	130K	3324K	1250K	254	223
4	2518K	60K	296K	138K	246	237

load for the same amount of communication. This increase in the computational density made our algorithms less communication-bound, and more scalable.

We given Table 3 to investigate computation and communication requirements with different partitions of the Flickr tensor in one Tucker-ALS iteration for 2048-way parallel run with 256 MPI ranks. In this table `nzmax` and `nrmax` refer, respectively, to the maximum tensor nonzeros per MPI-rank and the maximum number of rows on which an MPI-rank has nonzeros. The first observation is that the number of tensor nonzeros per MPI-rank is always well balanced with the fine grain computations. This results in load balanced TTMc computations across MPI-ranks. On the other hand, with the coarse grain formulation, the nonzeros per MPI-rank is not well balanced. This is mostly due to feasibility; some $N-1$ dimensional sub-tensors (slices, when $N=3$) contain much more nonzeros than others, and the coarse grain formulation cannot simply achieve better balance. The second observation is that the fine grain achieves balance on the number of rows per MPI-rank as well. For example, the total number of rows in fine-hp per each mode is 130636,1450259, 28916044, and 5035767 (compare the increases with respect to the sizes in Table 1). This yields 16%, 55%, 96%, and 63% load imbalance during SVD step. Coarse-hp obtains smaller number of rows per MPI-rank (no publication) but achieves worse imbalance than the fine-hp (about 700%, 216%, 180%, 208%). As we also observed before [24], fine-hp is more effective in reducing the communication cost than the coarse-hp.

6 Conclusion

We discussed efficient parallelization of the alternating least squares based algorithms (Tucker-ALS also called HOOI) for Tucker decomposition of sparse tensors in current distributed memory systems. We introduced a tree-based computation scheme called TTM-tree for tensor matrix multiplication (TTM) operations which enables data reuse in the computational core of the targeted algorithms. We discussed coarse and fine grain algorithms for implementing the TTMs. The granularity in TTMs also affects the design choices for the computing the singular vectors. We discussed a careful implementation with popular libraries in which the communication cost reduction and load balance are implicitly achieved by aligning the data partition for this step with that of TTMs. We put all of these together and implemented a complete Tucker decomposition algorithm with shared (via OpenMP) and distributed memory (with MPI) parallelism. Using this implementation, we reported speedups up on up to 2048 cores.

This tensor matrix multiplication tree is of independent interest as it can also be used in sequential computing environments with dense tensors. There are many possible trees, but we explored only two of them in this study, and left further investigation as a future work. In particular, automated means to decide the shape of the tree according to a given sparse tensor need to be developed. Also, tensor times matrix-chain products are used in other algorithms [15] for Tucker decomposition. The proposed computational tools, including the task definitions and the TTM-tree can be used in those algorithms.

References

[1] C. A. Andersson and R. Bro. The N-way toolbox for MATLAB. *Chemometrics and Intelligent Laboratory Systems*, 52(1):1–4, 2000.

[2] C. J. Appellof and E. Davidson. Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents. *Analytical Chemistry*, 53(13):2053–2056, 1981.

[3] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[4] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6. Available online, February 2015.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2015.

[6] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *IEEE Conference on High Performance Extreme Computing (HPEC),*, pages 1–6, Sept 2012. doi: 10.1109/HPEC.2012.6408676.

[7] M. M. Baskaran, B. Meister, and R. Lethin. Low-overhead load-balanced scheduling for sparse tensor computations. In *IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–6, Waltham, MA, USA, Sept. 2014. IEEE.

[8] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[9] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[10] Ü . V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[11] J. H. Choi and S. V. N. Vishwanathan. DFacTo: Distributed factorization of tensors. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *27th Advances in Neural Information Processing Systems*, pages 1296–1304, Montreal, Quebec, Canada, 2014.

[12] P. Comon. Tensors: A brief introduction. *IEEE Signal Processing Magazine*, 31(3):44–53, May 2014.

[13] L. De Lathauwer and B. De Moor. From matrix to tensor: Multilinear algebra and signal processing. In *Institute of Mathematics and Its Applications Conference Series*, volume 67, pages 1–16, 1998.

[14] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[15] L. Eldén and B. Savas. A Newton–Grassmann method for computing the best multilinear rank-(r_1, r_2, r_3) approximation of a tensor. *SIAM Journal on Matrix Analysis and Applications*, 31 (2):248–271, 2009.

[16] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[17] O. Görlitz, S. Sizov, and S. Staab. Pints: Peer-to-peer infrastructure for tagging systems. In *Proceedings of the 7th International Conference on Peer-to-peer Systems*, IPTPS'08, page 19, Berkeley, CA, USA, 2008. USENIX Association.

[18] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, 2005.

[19] V. Hernández, J. E. Román, and A. Tomás. Restarted Lanczos bidiagonalization for the SVD in SLEPc. Technical Report STR-8, Universitat Politècnica de València, 2007.

[20] V. Hernández, J. E. Román, and A. Tomás. A robust and efficient parallel SVD solver based on restarted Lanczos bidiagonalization. *Electronic Transactions on Numerical Analysis*, 31: 68–85, 2008.

[21] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos. Haten2: Billion-scale tensor decompositions. In *IEEE 31st International Conference on Data Engineering (ICDE)*, pages 1047–1058, 2015. doi: 10.1109/ICDE.2015.7113355.

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

[23] K. Kaya, F.-H. Rouet, and B. Uçar. On partitioning problems with complex objectives. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *LNCS*, pages 334–344. Springer Berlin / Heidelberg, 2012.

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

[25] O. Kaya and B. Uçar. High performance parallel algorithms for Tucker decompositions of higher order sparse tensors. Technical report, Inria, Grenoble-Rhône-Alpes, 2015.

[26] H. A. L. Kiers and A. der Kinderen. A fast method for choosing the numbers of components in Tucker3 analysis. *British Journal of Mathematical and Statistical Psychology*, 56:119–125, 2003.

[27] T. Kolda and B. Bader. The TOPHITS model for higher-order web link analysis. In *Proceedings of Link Analysis, Counterterrorism and Security 2006*, 2006.

[28] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[29] 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*, pages 363–372, Pisa, Italy, 2008.

[30] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 76:1–76:12, Austin, Texas, 2015. ACM, New York, NY, USA.

[31] K. Maruhashi, F. Guo, and C. Faloutsos. MultiAspectForensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 203–210, July 2011. doi: 10.1109/ASONAM.2011.80.

[32] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. ParCube: Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition. *ACM Transactions on Knowledge Discovery from Data*, 10(1):3:1–3:25, July 2015.

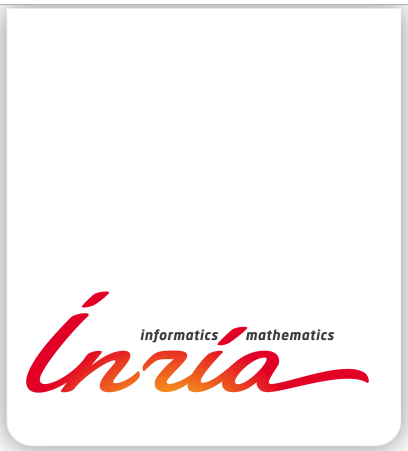
[33] A. Pinar and B. Hendrickson. Partitioning for complex objectives. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (CDROM)*, page 121, Washington, DC, USA, 2001. IEEE Computer Society.

[34] J. E. Roman, C. Campos, E. Romero, and A. Tomas. SLEPc users manual. Technical Report DSIC-II/24/02 - Revision 3.6, D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2015.

[35] S. Smith and G. Karypis. DMS: Distributed sparse tensor factorization with alternating least squares. Technical Report 15-007, Department of Computer Science and Engineering, University of Minnesota, May 2015.

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

[37] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Computer Vision—ECCV 2002*, pages 447–460. Springer, 2002.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399