High Performance Parallel Algorithms for Tensor Decompositions

Oguz Kaya

Numéro National de Thèse : 2017LYSEN051

THÈSE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'École Normale Supérieure de Lyon

*École Doctorale N°*512
Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 15 Septembre 2017, par :

Oguz KAYA

High Performance Parallel Algorithms for Tensor Decompositions

Algorithmes Parallèles pour les Décompositions des Tenseurs

Devant la commission d'examen formée de :

Marc	BABOULIN	Professeur, Université de Paris-Sud	*Rapporteur*
Pierre	COMON	Directeur de recherche, CNRS et Grenoble INP	*Examinateur*
Camille	COTI	Maître de conférences, IUT de Villetaneuse	*Examinatrice*
Lieven	DE LATHAUWER	Professeur, KU Leuven	*Rapporteur*
Mariya	ISHTEVA	Professeur de recherche, Vrije Universiteit Brussel	*Examinatrice*
Yves	ROBERT	Professeur, ENS de Lyon	*Directeur de thèse*
Bora	UÇAR	Chargé de recherche (CR1), CNRS et ENS de Lyon	*Co-encadrant*
Richard	VUDUC	Professeur, Georgia Institute of Technology	*Rapporteur*

Contents

Table des Matières

Chapter 1

Introduction

In the age of big data, the volume of collected data grows rapidly not only due to sheer increase in the number of data sources, such as the number of users of social media, but also as a natural result of the insatiate need of obtaining a more precise and in-depth understanding of the big datasets arising in real-world applications. In parallel with the growth in data size, the number of data features also escalates, which in turn increases the dimensionality of data. In response to this, tensors have been increasingly employed in the recent past owing to their ability to naturally model such high dimensional datasets, and tensor decomposition methods have proven to be an effective tool for gaining insights from the analysis of such data.

Tensor decompositions have been used in a vast range of application domains, including the analysis of Web graphs [57], knowledge bases [15], recommender systems [85, 86, 103], signal processing [63], computer vision [105], health care [81], chemometrics [4], forensic data analysis [73], and many others [23, 58, 92]. In these applications, tensor decomposition are used to find latent relations or predict missing elements in data using its low rank structure. For this aptitude of tensor decompositions, there have been considerable efforts in designing numerical algorithms for different tensor decomposition problems [58], and algorithmic and software contributions go hand in hand with these efforts [3, 7, 22, 40, 50, 51, 96, 98]. These decompositions can be computed to express both dense and sparse tensors, while an efficient computation of the latter has only very recently drawn significant interest with the emerging applications using high dimensional big sparse datasets.

The two prominent tensor decomposition methods that are widely used in the literature are CANDECOMP/PARAFAC (CP) decomposition (or equivalently, canonical polyadic decomposition (CPD)) and Tucker decomposition. Both these formulations have uses in various applications; in particular, CP formulation is deemed useful for understanding latent components [57], whereas Tucker formulation is considered to be more appropriate for compression [5], identifying relations among the factors [39], and predicting missing data entries [86].

CP formulation approximates a given tensor as a sum of rank-one tensors. The standard algorithm for computing CP decomposition is CP-ALS [16, 35], which is based on the alternating least squares method, whereas other variants also exist [1]. Most of these algorithms are iterative, in which the computational core of each iteration involves a special operation called the matricized tensor-times Khatri-Rao product (MTTKRP). When the input tensor is sparse and N dimensional, the MTTKRP operation amounts to the element-wise multiplication of $N-1$ matrix row vectors and their scaled sum reduction according to the nonzero structure of the tensor. When the dimensionality of the ten-

sor increases, this operation gets computationally more expensive; hence, efficiently carrying out MTTKRP for higher dimensional tensors is of our particular interest due to the needs of emerging applications [81]. This operation has received recent attraction for efficient execution in different settings such as MATLAB [3, 7], MapReduce [40], shared memory [46, 52, 98], and distributed memory [22, 50, 52, 96].

Tucker formulation gives another decomposition that has been employed in many data analysis problems [86, 103, 110]. One of the most popular algorithms for computing Tucker decomposition an alternating least squares (ALS) based method called Higher Order Orthogonal Iteration (HOOI) [65]. A key operation in HOOI algorithm involves the multiplication of an N-dimensional tensor with $N-1$ matrices, which we call tensor-times-matrix chain product (TTMc). In many applications involving Tucker decomposition, the tensor formed from the data is sparse and big, both in terms of dimension sizes and the number of nonzero entries. The sparsity needs to be adequately exploited so as to compute Tucker decomposition efficiently.

The main goal of this thesis is to address such computational challenges in computing these two tensor decompositions from a multitude of perspectives, in an effort to render these tensor decomposition methods affordable to use in analyzing massive scale datasets. A particular focus is given on effectively carrying out MTTKRP and TTMc operations, which constitute the most costly step in many tensor decomposition algorithms. These approaches can be summarized as follows:

- Shared and distributed memory parallelizations with novel partitioning routines for load balancing and communication reduction.
- Tensor data structures for effectively carrying out tensor operations in parallel.
- New computational schemes and algorithms providing asymptotic computational gains in computing standard tensor decompositions.
- Faster numerical methods for handling high dimensional tensors.
- Complexity analysis pertaining to finding optimal computational schemes for tensor decompositions.

Other directions are also taken in the literature in accelerating the computation of tensor decompositions. The first approach is using randomization techniques that operate on a smaller sample of the tensor [60, 108]. The second direction involves new generalized tensor decomposition schemes and algorithms yielding faster convergence to the solution [99]. Additionally, there are also techniques that exploit an underlying structure on the factor matrices implied by the problem or application at hand [92, 106, 107]. Indeed, the focus of this thesis, using parallelization and HPC techniques for tensor decompositions, is an orthogonal research direction that can be coupled with any of these approaches, hence remains pertinent in all cases.

The approaches employed in the thesis for computing CP/Tucker decomposition of big sparse and dense tensors are organized in the following chapters as follows. In Chapter 3, we discuss novel techniques for a fast computation of the CP decomposition of big sparse tensors. This efficient computation is achieved through a multitude of approaches: a new computational scheme for carrying out the most expensive step (MTTKRP) of CP decomposition algorithms; a novel sparse tensor data structure that enables a scalable representation of high dimensional sparse tensors with significant computational and memory gains; a shared memory parallelization of CP decomposition algorithms using this

computational scheme and data structure; and an effective distributed memory parallelization together with scalable data partitioning schemes, obtained through novel hypergraph models, for the parallel algorithm. This chapter lays the foundations of this thesis, as these techniques are properly adopted by methods in the subsequent chapters. In Chapter 4, we discuss the computation and parallelization of a "modified" CP decomposition, in which we put the constraint of having only nonzero entries in CP decomposition. This enables a very fast parallel algorithm whose complexity does not increase with tensor dimensionality, and provides comparable results to the standard CP decomposition in approximation quality. Next, we investigate the parallelization of Tucker decomposition algorithms in shared and distributed memory environments for sparse tensors in Chapter 5, also adopt the new computational scheme to reduce the cost of the expensive TTMc step. Afterwards, Chapter 6 discusses the distributed memory parallelization of a related problem, non-negative matrix factorization, to which most of our parallelization and partitioning techniques in tensor decompositions gracefully apply. Chapter 7 investigates an optimal use of our new computational scheme in computing dense CP and Tucker decompositions with an in-depth theoretical analysis. Finally, Chapter 8 summarizes the available high performance sparse tensor software, and exemplifies the software developed during the thesis by highlighting its capabilities. All these chapters are summarized more in detail in what follows.

Thesis Overview

Chapter 2: Background. This chapter provides most of the background required by the subsequent chapters. It introduces the tensor notation, describes tensor decomposition algorithms, provides a formal description of a tree data structure and the hypergraph partitioning problem, and explains tensor datasets and parallel compute environments used in the experiments of most chapters.

Chapter 3: Parallel CP Decomposition of Sparse Tensors using Dimension Trees. This chapter focuses on computing the CP decomposition of sparse tensors, which has successfully been applied to many well-known problems in web search, graph analytics, recommender systems, health care data analytics, and many other domains. In these applications, computing the CP decomposition of sparse tensors efficiently is essential in order to be able to process and analyze data of massive scale. For this purpose, we investigate an efficient computation and parallelization of the CP decomposition of sparse tensors in this chapter. We provide a tree-based novel computational scheme using a data structure called dimension tree for significantly reducing the cost of MTTKRP in CP-ALS. We then effectively parallelize this computational scheme in the context of CP-ALS in shared and distributed memory environments, and propose data and task distribution models for better scalability. We compare our parallel implementations with a state-of-the-art parallel tensor factorization library using tensors formed from real-world and synthetic datasets. With our algorithmic contributions and implementations, we report up to 5.96x, 5.65x, and 3.9x speedup in sequential, shared memory parallel, and distributed memory parallel executions over the state of the art, and achieve strong scalability up to 4096 cores on an IBM BlueGene/Q supercomputer.

Chapter 4: Parallel Nonzero CP Decomposition of Sparse Tensors. To efficiently handle high dimensional big sparse tensors, we introduce in this chapter a shared memory parallel algorithm for computing a CP decomposition in which factor matrices are assumed to not contain any zero elements. This additional constraint enables a very efficient computation of the MTTKRP step in CP-ALS, and the performance gains increase with the dimensionality of tensor. For an N-dimensional tensor having k nonzero entries, our method provides $O(\log k)$ and $O(\log N \log k)$ faster preprocessing times, and

performs $O(N)$ and $O(\log N)$ less work in computing a CP decomposition over two efficient state-of-the-art methods, the latter being the algorithm introduced in Chapter 3. With these algorithmic contributions and a highly tuned parallel implementation, we achieve up to 16.7x speedup in sequential, and up to 10.5x speedup in parallel executions over these libraries on a 28-core workstation. In doing so, we incur with up to 24x less preprocessing time, and use up to $O(\log N)$ less memory for storing intermediate computations. We show using a real-world tensor that the accuracy of our method is comparable to the standard CP decomposition.

Chapter 5: Parallel Tucker Decomposition of Sparse Tensors. In this chapter, we investigate an efficient parallelization of an algorithm for computing the well-known Tucker decomposition of general N-dimensional sparse tensors. The algorithm is iterative and uses 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 (TTMc step); (ii) the product is then converted to a matrix; and (iii) a few leading left singular vectors of the resulting matrix are computed (truncated SVD step (TRSVD)) to update one of the matrices for the next TTMc step. We propose an efficient parallelization of these algorithms for the current parallel platforms with multicore nodes. We discuss a set of preprocessing steps which takes all computational decisions out of the main iteration of the algorithm and provides an intuitive shared-memory parallelism for the TTMc and TRSVD steps. We show how the dimension tree-based framework can be employed to perform the TTMC step faster. We propose a coarse- and a fine-grain distributed memory parallel algorithms, investigate their data dependencies, and identify efficient communication schemes. We demonstrate the way the computation of singular vectors in the TRSVD step can be carried out efficiently following the TTMc step. Finally, we develop a hybrid MPI-OpenMP implementation of the overall algorithm, and report scalability results up to 4096 cores of an IBM BlueGene/Q supercomputer.

Chapter 6: Parallel Non-negative Sparse Matrix Factorization. Non-negative matrix factorization (NMF) is the problem of finding two non-negative low rank factors $\mathbf{W}$ and $\mathbf{H}$ for a given input matrix $\mathbf{A}$ such that $\mathbf{A} \approx \mathbf{WH}$. NMF is a useful tool for many internet applications such as topic modeling in text mining and community detection in social networks. The algorithms for computing NMF have a notable similarity to tensor decomposition methods, which opens up the possibility to apply all our parallelization techniques to the computation of NMF for sparse matrices. To this end, the focus of this chapter is on scaling distributed NMF to very large sparse datasets by employing effective algorithms, communications, and partitioning schemes that leverage the sparsity of the input matrix $\mathbf{A}$. Our method employs an efficient point-to-point communication scheme, and removes any redundant communications that exist in state-of-the-art algorithms. Unlike the state-of-the-art, our method also permits arbitrary partitions of the sparse input matrix as well as factor matrices, which enables us to employ effective partitioning schemes for better scalability. We experiment on a cluster using up to 3072 cores, and report significant better scalability with respect to existing methods owing to our efficient communication and partitioning strategies. We also run our algorithms on an IBM BlueGene/Q supercomputer, and achieve scalability up to 32768 processors with a 32x speedup over the execution using 256 processors.

Chapter 7: Computing Dense Tensor Decompositions using Dimension Trees. The focus of this chapter is on computing CP and Tucker decompositions of dense tensors using the dimension tree-based computational scheme. We extend our earlier algorithms using dimension trees to the dense case, and propose efficient algorithms for computing both CP and Tucker decompositions. These algorithms are oblivious to the structure of the tree, which determines the computational cost in these algorithms. This raises the question of finding an optimal tree structure. We prove that finding an

optimal dimension tree for computing both CP and Tucker decompositions is NP-hard. For CP decomposition, we show that an optimal tree has to be binary, whereas in the Tucker case we provide a counter-example for which the optimal dimension tree is not binary. Finally, we introduce an optimal greedy algorithm that finds the ordering minimizing the cost of a series of TTMs in computing Tucker decomposition.

Chapter 8: High Performance Tensor Software. In this chapter, we give an overview of the existing high performance software for sparse tensor decompositions, and briefly discuss the capabilities of two software libraries developed during this thesis. The first software package is HYPERTENSOR, which involves parallel algorithms for computing sparse CP and Tucker decompositions, and employs an novel sparse tensor data structure and dimension tree-based computational scheme in these parallelizations. The second software package is PACOS (Partitioning and communication framework for sparse irregular applications) that drives all partitioning and communication routines of HYPERTENSOR for distributed memory parallelism. Even though PACOS is employed in HYPERTENSOR for parallel tensor factorization, the provided routines are generic, which enables using PACOS in the parallelization of many sparse (multi)-linear algebra routines including sparse matrix vector/matrix multiplication and various graph algorithms. We have recently integrated PACOS into NMFLIBRARY, a high performance parallel non-negative matrix factorization library, which provided much better scalability owing to effective partitioning strategies and efficient communication routines provided in PACOS.

À l'âge du "big data", le volume de données recueillies augmente rapidement non seulement en raison de l'augmentation du nombre de sources de données, comme le nombre d'utilisateurs des réseaux sociaux, mais aussi comme un résultat naturel du besoin insatiable d'obtenir une compréhension plus précise et plus complète des grands ensembles de données découlant d'applications du monde réel. Parallèlement à la croissance de la taille des données, le nombre de fonctions de données augmente également, ce qui augmente la dimensionnalité des données. En réponse à cela, les tenseurs ont été de plus en plus employés dans le passé récent en raison de leur capacité à modeler naturellement de tels ensembles de données de grande dimension, et les méthodes de décomposition de tenseur se sont révélées être un outil efficace pour tirer des idées de l'analyse de ces données.

Les décompositions de tenseur ont été utilisées dans une vaste gamme de domaines d'application, y compris l'analyse des graphiques Web [57], des bases de connaissances [15], les systèmes de recommandation [85, 86, 103], le traitement des signaux [63], la vision par ordinateur [105], les soins de santé [81], la chimiométri [4], l'analyse de données médico-légales [73] et bien d'autres [23, 58, 92]. Dans ces applications, la décomposition de tenseur sert à trouver des relations latentes ou à prédire les éléments manquants dans les données en utilisant sa structure de rang réduit. Pour cette aptitude des décompositions tensorielles, des efforts considérables ont été déployés dans la conception d'algorithmes numériques pour différents problèmes de décomposition de tenseur [58], et les contributions algorithmiques et logicielles vont de pair avec ces efforts [3, 7, 22, 40, 50, 51, 96, 98]. Ces décompositions peuvent être calculées pour exprimer des tenseurs denses et creux, alors qu'un calcul efficace de ce dernier n'a que très récemment suscité un intérêt significatif pour les applications émergentes utilisant des ensembles de données à grande dimension et à grande échelle.

Les deux méthodes de décomposition de tenseur proéminentes qui sont largement utilisées dans la littérature sont la décomposition CANDECOMP/PARAFAC (CP) (ou équivalent, la décomposition polyadique canonique (CPD)) et la décomposition Tucker. Les deux formulations ont des utilisations dans diverses applications; en particulier, la formulation de CP est jugée utile pour comprendre les composants latents, alors que la formulation Tucker est considérée plus appropriée pour la compression, l'identification des relations entre les facteurs et la prédiction des entrées de données manquantes.

La formulation CP se rapproche d'un tenseur donné en tant que somme des tenseurs de premier rang. L'algorithme standard pour le calcul de la décomposition CP est CP-ALS [16, 35], qui est basé sur la méthode des moindres carrés alternés, tandis que d'autres variantes existent également [1]. La plupart de ces algorithmes sont itératifs dans lesquels le noyau de calcul de chaque itération implique une opération spéciale appelée le produit de tenseur matricizé et le produit Khatri-Rao (MTTKRP). Lorsque le tenseur d'entrée est creux et N-dimensionnel, l'opération MTTKRP équivaut à la multiplication élémentaire des vecteurs de rangée de matrice $N-1$ et à leur réduction de somme à l'échelle selon la structure nonzéro du tenseur. Lorsque la dimensionnalité du tenseur augmente, cette opération devient plus coûteuse. Par conséquent, la réalisation efficace de MTTKRP pour des tenseurs dimensionnels plus élevés est de notre intérêt particulier en raison des besoins des applications émergentes [81]. Cette opération a reçu une attraction récente pour une exécution efficace dans différents domaines tels que MATLAB [3, 7], MapReduce [40], mémoire partagée [46, 52, 98] et mémoire distribuée [22, 50, 52, 96].

La formulation Tucker donne une autre décomposition qui a été utilisée dans de nombreux problèmes d'analyse de données [86, 103, 110]. L'un des algorithmes les plus populaires pour calculer la décomposition Tucker est une méthode basée sur les moindres carrés alternatifs (ALS) appelée iteration orthogonale à ordre supérieur (HOOI) [65]. Une opération clé dans l'algorithme HOOI implique

la multiplication d'un tenseur N-dimensionnel avec des matrices $N-1$, que nous appelons un produit à chaîne matricielle tensorielle (TTMc). Dans de nombreuses applications impliquant la décomposition Tucker, le tenseur formé à partir des données est creux et grand, tant en termes de dimensions que de nombre d'entrées nonzéro. La sparticité doit être exploitée de manière adéquate afin de calculer efficacement la décomposition Tucker.

L'objectif principal de cette thèse est d'aborder ces défis informatiques dans le calcul de ces deux décompositions tensorielles à partir d'une multitude de points de vue, dans le but de rendre ces méthodes de décomposition de tenseur abordables à utiliser dans l'analyse d'ensembles de données à grande échelle. Une attention particulière est accordée à la réalisation efficace des opérations MTTKRP et TTMc, qui constituent l'étape la plus coûteuse dans de nombreux algorithmes de décomposition de tenseur. Ces approches peuvent être résumées comme suit:

- Parallélisations en mémoire partagées et distribuées avec de nouvelles routines de partitionnement pour l'équilibrage de charge et la réduction de la communication.
- Structures de données de tenseur pour effectuer efficacement des opérations tensorielles en parallèle.
- Nouveaux schémas et algorithmes de calcul fournissant des gains de calcul asymptotiques dans le calcul des décompositions de tenseur standard.
- Méthodes numériques plus rapides pour la manipulation de tenseurs dimensionnels élevés.
- Analyse de complexité concernant la recherche de schémas informatiques optimaux pour les décompositions des tenseurs.

D'autres directions sont également prises dans la littérature pour accélérer le calcul des décompositions des tenseurs. La première approche consiste à utiliser des techniques de randomisation qui fonctionnent sur un échantillon plus petit du tenseur [60, 108]. La deuxième direction implique de nouveaux schémas généraux de décomposition de tenseur et des algorithmes donnant une convergence plus rapide à la solution [99]. En outre, il existe également des techniques qui exploitent une structure sous-jacente sur les matrices factorisées impliquées par le problème ou la demande en cours [92, 106, 107]. En effet, l'objectif de cette thèse, en utilisant la parallélisation et les techniques HPC pour les décompositions des tenseurs, est une direction de recherche orthogonale qui peut être couplée à l'une de ces approches, donc reste pertinente dans tous les cas.

Les approches employées dans la thèse pour le calcul de la décomposition CP/Tucker de grands tenseurs creux et denses sont organisées dans les chapitres suivants comme suit. Dans le Chapitre 3, nous discutons de nouvelles techniques pour un calcul rapide de la décomposition du CP de grands tensorants creux. Ce calcul efficace est réalisé grâce à une multitude d'approches: un nouveau schéma de calcul pour effectuer l'étape la plus coûteuse (MTTKRP) des algorithmes de décomposition du CP; une nouvelle structure de données de tenseur creux qui permet une représentation évolutive de tenseurs creux à haute dimension avec des gains significatifs de calcul et de mémoire; une parallélisation en mémoire partagée des algorithmes de décomposition du CP à l'aide de ce schéma de calcul et de la structure de données; et une parallélisation efficace en mémoire distribuée avec des schémas de partitionnement de données évolutifs, obtenus grâce à de nouveaux modèles d'hypergraphe, pour l'algorithme parallèle. Ce chapitre pose les bases pour les chapitre suivants car les techniques utilise dans ce chapitre sont adequatement adopte aux problèmes dans les chapitres suivants. Dans le

Chapitre 4, nous discutons du calcul et de la parallélisation d'une décomposition CP "modifiée", dans laquelle nous mettons la contrainte d'avoir seulement des entrées nonzéro dans la décomposition du CP. Ceci permet un algorithme parallèle très rapide dont la complexité n'augmente pas avec la dimensionnalité du tenseur et fournit des résultats comparables à la décomposition standard de CP en qualité d'approximation. Ensuite, nous étudions la parallélisation des algorithmes de décomposition Tucker dans les environnements de mémoire partagés et distribués pour les tensorants creux au Chapitre 5, adoptons également le nouveau schéma de calcul pour réduire le coût de l'étape TTMc coûteuse. Par la suite, le Chapitre 6 analyse la parallélisation en mémoire distribuée d'un problème associé, la factorisation des matrices non-négatives, à laquelle la plupart de nos techniques de parallélisation et de partitionnement des décompositions des tenseurs sont gracieusement appliquées. Le Chapitre 7 étudie une utilisation optimale de notre nouveau schéma informatique dans le calcul des décompositions denses de CP et de Tucker avec une analyse théorique approfondie. Enfin, le Chapitre 8 résume les logiciels haute performance disponibles de tenseurs creux et illustre le logiciel développé au cours de la thèse en mettant en évidence ses capacités. Tous ces chapitres sont résumés plus en détail dans ce qui suit.

Vue d'Ensemble de la Thèse

Chapitre 2: Contexte. Ce chapitre fournit la plupart du contexte requis par les chapitres suivants. Il introduit la notation tensorielle, décrit les algorithmes de décomposition de tenseur, fournit une description formelle d'une structure de données arborescentes et le problème de partitionnement des hypergraphes et explique les ensembles de données et les environnements de calcul parallèles utilisés dans les expériences de la plupart des chapitres.

Chapitre 3: Décomposition CP Parallèle des Tenseurs Creux Utilisant des Arbres de Dimension. Ce chapitre se concentre sur le calcul de la décomposition CP des tenseurs creux, qui a été appliquée avec succès à de nombreux problèmes bien connus dans la recherche sur le Web, l'analyse graphique, les systèmes de recommandation, les analyses de données sur les soins de santé et bien d'autres domaines. Dans ces applications, un calcul efficace de la décomposition CP des tenseurs creux est essentiel pour pouvoir traiter et analyser des données à grande échelle. Pour ce faire, nous étudions un calcul efficace et une parallélisation de la décomposition CP des tenseurs creux dans ce chapitre. Nous proposons un nouveau schéma de calcul basé sur un arbre en utilisant une structure de données appelée arbre de dimension pour réduire considérablement le coût du MTTKRP dans CP-ALS. Ensuite, nous parallélisons efficacement ce schéma de calcul dans le contexte CP-ALS dans des environnements de mémoire partagés et distribués, et proposons des modèles de distribution de données et de tâches pour une meilleure scalabilité. Nous comparons nos implémentations parallèles avec une bibliothèque de factorisation de tenseur parallèle courant en utilisant des tenseurs formés à partir d'ensembles de données du monde réel et synthétiques. Grâce à nos contributions et implémentations algorithmiques, nous rapportons jusqu'à 5,96x, 5,65x et 3,9x d'accélération dans l'exécution séquentiel et les exécutions parallèles en mémoire partagées en distribuée et obtenons une grande scalabilité allant jusqu'à 4096 cœurs sur un superordinateur IBM BlueGene/Q.

Chapitre 4: Décomposition CP Parallèle Nonzéro des Tenseurs Creux. Pour traiter efficacement les grands tenseurs à grande dimension, nous présentons dans ce chapitre un algorithme parallèle en mémoire partagée pour calculer une décomposition de CP dans laquelle les matrices de facteurs sont supposées ne contenir aucun élément zéro. Cette contrainte supplémentaire permet un calcul très efficace de l'étape MTTKRP dans CP-ALS, et les gains de performance augmentent avec la

dimensionnalité du tenseur. Pour un tenseur N-dimensionnel ayant k entrées nonzéro, notre méthode fournit $O(\log k)$ et $O(\log N \log k)$ des temps de prétraitement plus rapides et effectue $O(N)$ et $O(\log N)$ moins de travail dans le calcul d'une décomposition CP sur deux méthodes courants, ce dernier étant l'algorithme introduit au Chapitre 3. Avec ces contributions algorithmiques et une implémentation parallèle efficace, nous atteignons jusqu'à 16,7x d'accélération séquentielle et jusqu'à 10,5x d'accelération dans des exécutions parallèles sur ces bibliothèques sur une machine ayant à 28 cœurs. Ce faisant, nous avons un temps de prétraitement de jusqu'à 24x moins, et utilisons jusqu'à $O(\log N)$ moins de mémoire pour stocker des résultats intermédiaires. Nous montrons en utilisant un tenseur réel que la précision de notre méthode est comparable à la décomposition CP standard.

Chapitre 5: Décomposition Tucker Parallèle des Tenseurs Creux. Dans ce chapitre, nous étudions une parallélisation efficace d'un algorithme pour calculer la décomposition bien connue de Tucker des tenseurs creux N-dimensionnels généraux. L'algorithme est itératif et utilise la méthode des moindres carrés alternés. À chaque itération, pour chaque dimension d'un tenseur d'entrée N-dimensionnel, les opérations suivantes sont effectuées: (i) le tenseur est multiplié par des matrices $(N-1)$ (étape TTMc); (ii) le produit est ensuite converti en une matrice; et (iii) quelques vecteurs primaires de gauche de la matrice résultante sont calculés (étape SVD tronquée (TRSVD)) pour mettre à jour une des matrices pour la prochaine étape TTMc. Nous proposons une parallélisation efficace de ces algorithmes pour les plates-formes parallèles actuelles avec des nœuds multi-cœurs. Nous discutons d'un ensemble d'étapes de prétraitement qui supprime toutes les décisions de calcul de l'itération principale de l'algorithme et fournit un parallélisme intuitif en mémoire partagée pour les étapes TTMc et TRSVD. Nous montrons la manière dont le cadre arborescent des dimensions peut être utilisé pour effectuer l'étape TTMc plus rapidement. Nous proposons des algorithmes parallèles en mémoire distribuée, étudions leurs dépendances de données et identifions des systèmes de communication efficaces. Nous démontrons la manière dont le calcul des vecteurs singuliers dans l'étape TRSVD peut être effectué efficacement à la suite de l'étape TTMc. Enfin, nous développons une implémentation hybride MPI-OpenMP de l'algorithme et rapportons des résultats de scalabilité jusqu'à 4096 cœurs d'un superordinateur IBM BlueGene/Q.

Chapitre 6: Factorisation Non-Négative Parallèle des Matrices Creuses. La factorisation matricielle non négative (NMF) est le problème consistant à trouver deux facteurs négatifs non négatifs W et H pour une matrice d'entrée donnée A telle que A = WH. NMF est un outil utile pour de nombreuses applications Internet telles que la modélisation de sujet dans l'exploration de texte et la détection communautaire dans les réseaux sociaux. Les algorithmes de calcul de NMF ont une similitude notable avec les méthodes de décomposition de tenseur, ce qui ouvre la possibilité d'appliquer toutes nos techniques de parallélisation au calcul de NMF pour les matrices dispersées. À cette fin, l'accent est mis sur la mise à l'échelle de NMF distribué sur de très grands ensembles de données dispersés en utilisant des algorithmes efficaces, des communications et des schémas de partitionnement qui tirent parti de la matrice d'entrée A. Notre méthode utilise une communication point-à-point efficace Schéma, et supprime toutes les communications redondantes qui existent dans les algorithmes courant. Contrairement à l'état de l'art, notre méthode permet également des partitions arbitraires de la matrice d'entrée sparse ainsi que des matrices de facteurs, ce qui nous permet d'utiliser des schémas de partitionnement efficaces pour une meilleure évolutivité. Nous expérimentons sur un cluster en utilisant jusqu'à 3072 noyaux et signalons une meilleure évolutivité significative par rapport aux méthodes existantes grâce à nos stratégies efficaces de communication et de partitionnement. Nous exécutons également nos algorithmes sur un superordinateur IBM BlueGene/Q et réalisons une évolutivité jusqu'à 32768 processeurs avec une accélération de 32x sur l'exécution à l'aide de 256 processeurs.

Chapitre 7: Calcule de Décompositions des Tenseurs Denses Utilisant des Arbres de Dimension. L'objectif de ce chapitre est de calculer les décompositions CP et Tucker des tenseurs denses en utilisant le schéma de calcul basé sur les arborescences. Nous étendons nos algorithmes antérieurs à l'aide d'arbres de dimension dans le cas dense et proposons des algorithmes efficaces pour calculer les décompositions de CP et de Tucker. Ces algorithmes sont indifférents á la structure de l'arbre, ce qui détermine le coût de calcul de ces algorithmes. Cela soulève la question de trouver une structure arborescente optimale. Nous prouvons que trouver un arbre de dimension optimale pour calculer les décompositions CP et Tucker est NP-difficile. Pour la décomposition du CP, nous montrons qu'un arbre optimal doit être binaire, alors que dans le cas de Tucker, nous fournissons un contre-exemple pour lequel l'arborescence des dimensions optimales n'est pas binaire. Enfin, nous introduisons un algorithme glouton optimal qui trouve l'ordre minimisant le coût d'une série de TTM dans le calcul de la décomposition Tucker.

Chapitre 8: Logiciels Haute Performance de Tenseur. Dans ce chapitre, nous donnons un aperçu du logiciel de haute performance existant pour les décompositions des tenseurs creux et discutons brièvement les capacités de deux bibliothèques de logiciels développées au cours de cette thèse. Le premier paquet de logiciels est HYPERTENSOR, qui implique des algorithmes parallèles pour calculer des décompositions CP et Tucker des tenseurs creux, et utilise une nouvelle structure de données tensorielle et un schéma de calcul basé sur les arborescences dans ces parallélisations. Le deuxième logiciel est PACOS (partitionnement et cadre de communication pour les applications irrégulières creuses) qui déploie toutes les routines de partitionnement et de communication de HYPERTENSOR pour le parallélisme en mémoire distribuée. Même si PACOS est utilisé dans HYPERTENSOR pour la factorisation du tenseur en parallèle, les routines fournies sont génériques, ce qui permet l'utilisation de PACOS dans la parallélisation de nombreuses routines d'algèbre linéaire ou multilinéaire, y compris le produit des matrices creuses et vecteurs et divers algorithmes des graphes. Nous avons récemment intégré PACOS dans NMFLIBRARY, une bibliothèque parallèle haute performance de factorisation des matrices non-négative, qui a permis une meilleure évolutivité grâce à des stratégies de partitionnement efficaces et à des routines de communication efficaces fournies dans PACOS.

Chapter 2

Background

2.1 Tensor notation and operations

We denote the set $\{1, \ldots, N\}$ of integers as $\mathbb{N}_N$ for $N \in \mathbb{Z}^+$. For vectors, we use bold lowercase Roman letters, as in $\mathbf{x}$. For matrices, we use bold uppercase Roman letters, e.g., $\mathbf{X}$. For tensors, we generally follow the notation in Kolda and Bader's survey [58]. We represent tensors using bold calligraphic fonts, e.g., $\boldsymbol{\mathcal{X}}$. The *order* of a tensor is defined as the number of its *dimensions*, or equivalently, *modes*, which we denote by N. We use italic lowercase letters with corresponding indices to represent vector, matrix, and tensor elements, e.g., x_i for a vector x, $x_{i,j}$ for a matrix $\boldsymbol{\mathcal{X}}$, and $x_{i,j,k}$ for a 3-dimensional tensor $\boldsymbol{\mathcal{X}}$. For the column vectors of a matrix, we use the same letter in lowercase and with a subscript corresponding to the column index, e.g., $\mathbf{x}_i$ to denote $\boldsymbol{\mathcal{X}}(:, i)$. A *slice* of a tensor in the nth mode is a set of tensor elements obtained by fixing the index only along the nth mode. A *fiber* of a tensor is defined by fixing every index but one, and a *slice* of a tensor is obtained by fixing only one index. We use the MATLAB notation to refer to matrix rows and columns as well as tensors fibers and slices, e.g., $\boldsymbol{\mathcal{X}}(i, :)$ and $\boldsymbol{\mathcal{X}}(:, j)$ are the ith row and the jth column of $\boldsymbol{\mathcal{X}}$, whereas $\boldsymbol{\mathcal{X}}(i, j, :)$ and $\boldsymbol{\mathcal{X}}(:, :, k)$ represent a fiber and a slice of $\boldsymbol{\mathcal{X}}$ in the third dimension, respectively.

The multiplication of an N-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ along a dimension $n \in \mathbb{N}_N$ with a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is a tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times 1 \times I_{n+1} \times \cdots \times I_N}$ with elements

$$y_{i_1,\ldots,i_{n-1},1,i_{n+1},\ldots,i_N} = \sum_{j=1}^{I_n} v_j x_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N}. \tag{2.1}$$

This operation is called *tensor-times-vector multiply (TTV)* and is denoted by $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{v}$. The cost of this operation is $O(\prod_{i \in \mathbb{N}_N} I_i)$. $\boldsymbol{\mathcal{X}}$ can also be multiplied with a matrix $\mathbf{U} \in \mathbb{R}^{K \times I_n}$ in a mode n, which results in a tensor $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{U}$, $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times K \times I_{n+1} \times \cdots \times I_N}$ with elements

$$y_{i_1,\ldots,i_{n-1},k,i_{n+1},\ldots,i_N} = \sum_{j=1}^{I_n} u_{k,j} x_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N}. \tag{2.2}$$

In other words, TTV of $\boldsymbol{\mathcal{X}}$ with each row vector of $\mathbf{U}$ forms the kth slice of $\boldsymbol{\mathcal{Y}}$ in the nth dimension. This operation is called *tensor-times-matrix multiply (TTM)* or *n-mode product* and has the cost $O(K \prod_{i \in \mathbb{N}_N} I_i)$. The order of TTVs or TTMs in a set of distinct modes is irrelevant, i.e., $\boldsymbol{\mathcal{X}} \times_i \mathbf{u} \times_j \mathbf{w} = \boldsymbol{\mathcal{X}} \times_j \mathbf{w} \times_i \mathbf{u}$ for $\mathbf{u} \in \mathbb{R}^{I_i}$, $\mathbf{w} \in \mathbb{R}^{I_j}$, $i \neq j$, and $i, j \in \mathbb{N}_N$.

A tensor $\boldsymbol{\mathcal{X}}$ can be *matricized* in some modes, meaning that a matrix $\mathbf{X}$ can be associated with $\boldsymbol{\mathcal{X}}$ by identifying a subset of its modes to correspond to the rows of $\mathbf{X}$, and the rest of the modes to correspond to the columns of $\boldsymbol{\mathcal{X}}$. This involves a mapping of the elements of $\boldsymbol{\mathcal{X}}$ to those of the matricization $\mathbf{X}$ of the tensor. We will be exclusively dealing with the matricizations of tensors along a single mode, meaning that a single mode is mapped to the rows of the resulting matrix, and the rest of the modes correspond to its columns. We use $\mathbf{X}_{(n)}$ to denote matricization along a mode n, e.g., for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, the matrix $\mathbf{X}_{(1)} \in \mathbb{R}^{I_1 \times I_2 I_3 \ldots I_N}$ denotes the mode-1 matricization of $\boldsymbol{\mathcal{X}}$. Specifically, the tensor element $x_{i_1,\ldots,i_N}$ corresponds to the element $\left(i_1, i_2 + \sum_{j=3}^{N} \left[(i_j - 1) \prod_{k=2}^{j-1} I_k\right]\right)$ of $\mathbf{X}_{(1)}$ in this matricization. Matricizations in other modes are defined likewise.

The *Hadamard product* of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^I$ is a vector $\mathbf{w} = \mathbf{u} * \mathbf{v}, \mathbf{w} \in \mathbb{R}^I$ with elements $w_i = u_i \cdot v_i$. The *outer product* of $N > 1$ vectors $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(N)}$ of respective sizes $I_1, \ldots, I_N$ is denoted by $\boldsymbol{\mathcal{X}} = \mathbf{u}^{(1)} \circ \cdots \circ \mathbf{u}^{(N)}$ where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is a N-dimensional tensor with elements $x_{i_1,\ldots,i_N} = \prod_{t \in \mathbb{N}_N} u_{i_t}^{(t)}$. The *Kronecker product* of vectors $\mathbf{u} \in \mathbb{R}^I$ and $\mathbf{v} \in \mathbb{R}^J$ results in a vector $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$, $\mathbf{w} \in \mathbb{R}^{IJ}$, defined as

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

For matrices $\mathbf{U} \in \mathbb{R}^{I \times K}$ and $\mathbf{V} \in \mathbb{R}^{J \times K}$, their *Khatri-Rao product* corresponds to the Kronecker product of their corresponding columns, i.e.,

$$\mathbf{W} = \mathbf{U} \odot \mathbf{V} = \left[\mathbf{u}_1 \otimes \mathbf{v}_1, \ldots, \mathbf{u}_K \otimes \mathbf{v}_K\right], \tag{2.3}$$

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

We use the shorthand notation $\circ_{i \neq n} \mathbf{u}^{(i)}$ to denote operation $\circ$ on a set $\{\mathbf{u}_1^{(1)}, \ldots, \mathbf{u}^{(N)}\}$ of operands, e.g., $\boldsymbol{\mathcal{X}} \times_{i \neq n} \mathbf{u}^{(i)}$ denotes $\boldsymbol{\mathcal{X}} \times_1 \mathbf{u}^{(1)} \times_2 \cdots \times_{n-1} \mathbf{u}^{(n-1)} \times_{n+1} \mathbf{u}^{(n+1)} \times_{n+2} \cdots \times_N \mathbf{u}^{(N)}$, and $*_{n \in \mathbb{N}_N} \mathbf{u}^{(n)}$ gives $\mathbf{u}^{(1)} * \cdots * \mathbf{u}^{(N)}$.

We next describe the algorithms for computing Tucker and CP decompositions. Throughout the rest of the manuscript, we assume that the input tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times_1 \cdots \times_N I_N}$ has dimensions $I_n \geq 2$ for $n \in \mathbb{N}_N$, as we can otherwise remove all dimensions of size 1, and execute the algorithms on the lower dimensional tensor.

2.2 CP decomposition

The rank-R CP decomposition of a tensor $\boldsymbol{\mathcal{X}}$ expresses or approximates $\boldsymbol{\mathcal{X}}$ as a sum of R rank-1 tensors. For instance, a rank-R CP decomposition yields $\boldsymbol{\mathcal{X}} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ for $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ with $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$, and $\mathbf{c}_r \in \mathbb{R}^K$. This decomposition provides an element-wise approximation (or equality) $x_{i,j,k} \approx \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr}$, as shown in Figure 2.1. The minimum R value rendering this approximation an equality is called the *rank* (or CP-rank) of the tensor $\boldsymbol{\mathcal{X}}$, and computing this rank is NP-hard [36]. Here, the matrices $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_R]$, $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_R]$, and $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_R]$ are called *factor matrices*, or *factors*. For N-mode tensors, we use $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$ to refer to factor matrices having $I_1, \ldots, I_N$ rows and R columns, $\mathbf{u}_j^{(n)}$ to refer to the jth column of $\mathbf{U}^{(n)}$, and $u_{i,j}^{(n)}$ to denote the element of $\mathbf{U}^{(n)}$ with the index (i, j).

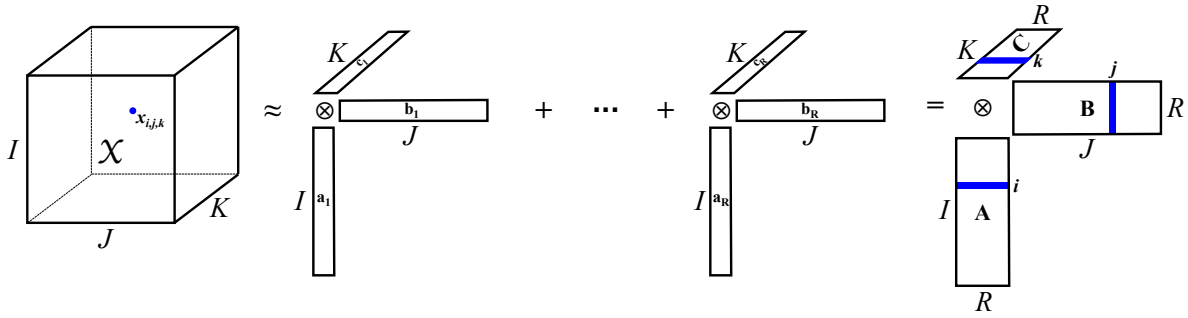


Figure 2.1: CP decomposition of a 3rd order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$.

The traditional algorithm for computing CP decomposition is CP-ALS; an iterative algorithm, shown in Algorithm 1, that progressively updates factors $\mathbf{U}^{(n)}$ in an alternating fashion starting from an initial guess. CP-ALS continues until it can no longer improve the solution, or it reaches the allowed maximum number of iterations. Factor matrices can be initialized randomly, or using the truncated SVD of the matricizations of $\boldsymbol{\mathcal{X}}$ [58]. Each iteration of CP-ALS consists of N *subiterations*, where in the nth subiteration $\mathbf{U}^{(n)}$ is updated using $\boldsymbol{\mathcal{X}}$ as well as the current values of all other factor matrices.

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

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

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

Computing the matrix $\mathbf{M}^{(n)} \in \mathbb{R}^{I_n \times R}$ at Line 5 of Algorithm 1 is the sole part involving the tensor $\boldsymbol{\mathcal{X}}$, and it is the most expensive computational step of CP-ALS. The operation $\boldsymbol{\mathcal{X}}_{(n)}(\odot_{i \neq n} \mathbf{U}^{(i)})$ is called the *matricized tensor-times Khatri-Rao product* (MTTKRP). The Khatri-Rao product of the involved $\mathbf{U}^{(n)}$s defines a matrix of size $(\prod_{i \neq n} I_i) \times R$ according to (2.3), and can get very costly in terms of computational and memory requirements when I_i or N is large, which is the case for many real-world sparse tensors. To alleviate this, various methods are proposed in the literature that enable performing MTTKRP without forming the Khatri-Rao product. One such formulation [6] expresses MTTKRP in terms of a series of TTVs, and computes the resulting matrix $\mathbf{M}^{(n)}$ column by column. With this formulation, the rth column of $\mathbf{M}^{(n)}$ can be computed using $N - 1$ TTVs as in

$$\mathbf{M}^{(n)}(:, r) \leftarrow \boldsymbol{\mathcal{X}} \times_{i \neq n} \mathbf{u}_r^{(i)}. \tag{2.4}$$

Another equivalent formulation computes MTTKRP as a summed reduction of Hadamard products of factor matrix rows scaled with tensor entries, which is given in Algorithm 2. For a sparse tensor

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

Input: $\mathcal{X}$: An N-dimensional tensor, $\mathcal{X} \in \mathbb{R}^{I_1,\ldots,I_N}$
$\mathbf{U}^{(1)} \ldots \mathbf{U}^{(N)}$: Factor matrices
n: The dimension of matricization for MTTKRP
Output: $\mathbf{M}^{(n)}$: MTTKRP result matrix, $\mathbf{M}^{(n)} \in \mathbb{R}^{I_n \times R}$
1: $\mathbf{M}^{(n)} \leftarrow 0$
2: **for** $x_{i_1,\ldots,i_n,\ldots,i_N} \in \mathcal{X}$ **do**
3: $\quad \mathbf{M}^{(n)}(i_n,:)$ += $x_{i_1,\ldots,i_n,\ldots,i_N} \left[*_{j \neq n} \left(\mathbf{U}^{(j)}(i_j,:)\right)\right]$

having nnz($\mathcal{X}$) nonzero entries, the overall cost of this algorithm is $O(\text{nnz}(\mathcal{X})NR)$ as each nonzero induces an Hadamard product of $N-1$ row vectors followed by an addition of a row vector of size R. This algorithm is used is commonly used by many existing implementations [6, 40].

Once $\mathbf{M}^{(n)}$ is obtained, the Hadamard product of the matrices ${\mathbf{U}^{(i)}}^T\mathbf{U}^{(n)}$ of size $R \times R$ is computed at Line 6 to form the matrix $\mathbf{H}^{(n)} \in \mathbb{R}^{R \times R}$. Note that within the subiteration n, only $\mathbf{U}^{(n)}$ is updated among all factor matrices. Therefore, for efficiency, one can precompute all matrices $\mathbf{W}^{(i)} = {\mathbf{U}^{(i)}}^T\mathbf{U}^{(i)}$ of size $R \times R$ for $1 \leq i \leq N$, performed at Line 2, then update ${\mathbf{U}^{(n)}}^T\mathbf{U}^{(n)}$ once $\mathbf{U}^{(n)}$ changes as in Line 9. As the rank R of approximation is typically much smaller than tensor dimensions I_n in practice, performing these Hadamard products to compute $\mathbf{H}^{(n)}$ and the matrix-matrix multiplication to compute ${\mathbf{U}^{(n)}}^T\mathbf{U}^{(n)}$ is cheaper compared with the TTV step. Once both $\mathbf{M}^{(n)}$ and $\mathbf{H}^{(n)}$ are computed, another matrix-matrix multiplication is performed using $\mathbf{M}^{(n)}$ and the pseudoinverse of $\mathbf{H}^{(n)}$ in order to update the matrix $\mathbf{U}^{(n)}$, which is not very expensive for the same reason. Finally, $\mathbf{U}^{(n)}$ is normalized column-wise, and the norms of column vectors are stored in a vector $\lambda \in \mathbb{R}^R$. Convergence is achieved when the relative improvement in the error norm, i.e., $|\mathcal{X} - \sum_{r=1}^{R} \lambda_r(\mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)})|$, is deemed small. The cost of this computation is insignificant; hence, we skip its details.

2.3 Tucker decomposition

Tucker decomposition expresses a given tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ with a smaller core tensor $\mathcal{G}$ multiplied by a factor matrix $\mathbf{U}^{(n)}$ of size $I_n \times R_n$ in each mode $n \in \mathbb{N}_N$. Here, the tuple $(R_1, \ldots, R_N)$ forms the requested rank of the decomposition across different modes. This decomposition is denoted by $[\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$, which expresses (or approximates) $\mathcal{X}$ as $\mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \cdots \times_N \mathbf{U}^{(N)}$. For example, an (R_1, R_2, R_3)-rank Tucker decomposition $[\![\mathcal{G}; \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \mathbf{U}^{(3)}]\!]$ of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ provides $x_{i,j,k} \approx \sum_{p=1}^{R_1} \sum_{q=1}^{R_2} \sum_{r=1}^{R_3} g_{p,q,r} u_{i,p}^{(1)} u_{j,q}^{(2)} u_{k,r}^{(3)}$. Figure 2.2 illustrates the Tucker decomposition of a 3-dimensional tensor.

A well-known algorithm for computing Tucker decomposition is Higher Order Orthogonal Iteration (HOOI) [64], which is shown in Algorithm 3. In this algorithm, factor matrices are initialized first. This initialization can be done randomly or using the higher-order SVD of $\mathcal{X}$ [64]. Then, the "repeat-until" loop applies the ALS method. Here, in each mode n, $\mathcal{X} \times_{i \neq n} {\mathbf{U}^{(i)}}^T$ is computed at Line 4, and we call this operation *tensor-times-matrix chain product* (TTMc). This produces a tensor of size $R_1 \times R_2 \times \cdots \times R_{n-1} \times I_n \times R_{n+1} \times \cdots \times R_N$, which is then matricized along the nth mode

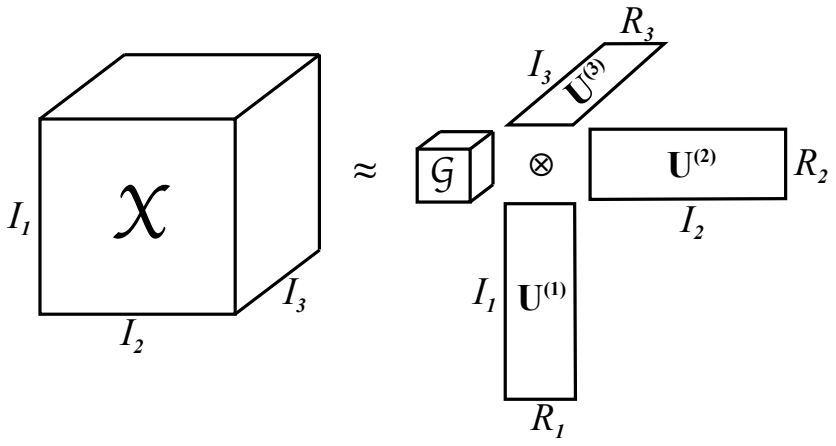


Figure 2.2: Tucker decomposition of a 3rd order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ as a core tensor $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ multiplied by matrices $\mathbf{U}^{(1)} \in \mathbb{R}^{I_1 \times R_1}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{I_2 \times R_2}$ and $\mathbf{U}^{(3)} \in \mathbb{R}^{I_3 \times R_3}$ in different modes. In the CP-decomposition, $\boldsymbol{\mathcal{G}}$ is a diagonal tensor having the same size $R_1 = R_2 = R_3 = R$ along each dimension, and $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$ and $\mathbf{U}^{(3)}$ have the same number of columns.

into the matrix $\mathbf{Y}_{(n)} \in \mathbb{R}^{I_n \times \Pi_{i \neq n} R_i}$. Next, the leading R_n left singular vectors of $\mathbf{Y}_{(n)}$ are computed to form the new $\mathbf{U}^{(n)}$ at Line 5. After all matrices $\mathbf{U}^{(n)}$ are updated, the core tensor $\boldsymbol{\mathcal{G}}$ is formed at Line 6, and the fit $(|\boldsymbol{\mathcal{X}}| - |\boldsymbol{\mathcal{G}}|)/|\boldsymbol{\mathcal{X}}|$ is measured to check convergence at the end of each iteration.

Algorithm 3 HOOI: ALS algorithm for computing Tucker decomposition

Input: $\boldsymbol{\mathcal{X}}$: An N-dimensional tensor, $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$
$R_1, \ldots, R_N$: The rank of Tucker decomposition in each dimension
Output: $[\![\boldsymbol{\mathcal{G}}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$: A rank-$(R_1, \ldots, R_N)$ Tucker decomposition of $\boldsymbol{\mathcal{X}}$
1: Initialize the matrix $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ for $n \in \mathbb{N}_N$.
2: **repeat**
3: **for** $n = 1, \ldots, N$ **do**
4: $\boldsymbol{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{X}} \times_{i \neq n} \mathbf{U}^{(i)^T}$
5: $\mathbf{U}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
6: $\boldsymbol{\mathcal{G}} \leftarrow \boldsymbol{\mathcal{X}} \times_1 \mathbf{U}^{(1)^T} \times_2 \cdots \times_N \mathbf{U}^{(N)^T}$
7: **until** convergence or the maximum number of iterations
8: **return** $[\![\boldsymbol{\mathcal{G}}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

2.4 Dimension tree

A *dimension tree* is a data structure that partitions the mode indices of an N-dimensional tensor in a hierarchical manner for computing tensor decompositions efficiently. It was first used in the hierarchical Tucker format representing the hierarchical Tucker decomposition of a tensor [31], which was introduced as a computationally feasible alternative to the original Tucker decomposition for higher order tensors. We provide the formal definition of a dimension tree along with some basic properties as follows.

Definition 1. *A dimension tree $\mathcal{T}$ for N dimensions is a tree with a root, denoted by* ROOT($\mathcal{T}$)*, and N leaf nodes, denoted by the set* LEAVES($\mathcal{T}$)*. In a dimension tree $\mathcal{T}$, each non-leaf node has at least*

two children, and each node $t \in \mathcal{T}$ is associated with a ***mode set*** *$\mu(t) \subseteq \mathbb{N}_N$ satisfying the following properties:*

1. $\mu(\text{ROOT}(\mathcal{T})) = \mathbb{N}_N$.
2. *For each non-leaf node $t \in \mathcal{T}$, the mode sets of its children partition $\mu(t)$.*
3. *The nth leaf node, denoted by $\mathcal{L}_n \in \text{LEAVES}(\mathcal{T})$, has $\mu(\mathcal{L}_n) = \{n\}$.*

For the simplicity of the presentation, we assume w.l.g that the sequence $\mathcal{L}_1, \ldots, \mathcal{L}_N$ corresponds to a relative ordering of the leaf nodes in a post-order traversal of the dimension tree. If this is not the case, we can relabel tensor modes accordingly. We define the *inverse mode set* of a node t as $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$. For each node t with a parent $P(t)$, $\mu(t) \subset \mu(P(t))$ holds due to the second property, which in turn yields $\mu'(t) \supset \mu'(P(t))$.

2.5 Hypergraph partitioning

A hypergraph $H = (V, E)$ consists of a set V of vertices and a set E of hyperedges. Each hyperedge is a subset of V. The vertices of a hypergraph can be associated with weights denoted by $\mathsf{w}[\cdot]$, and the hyperedges can be associated with costs denoted by $\mathsf{c}[\cdot]$. For a given integer $K \geq 2$, a K-way vertex partition of a hypergraph $H = (V, E)$ is denoted by $\Pi = \{V_1, \ldots, V_K\}$, where the parts are non-empty, i.e., $V_k \neq \emptyset$ for $k \in \mathbb{N}_K$; mutually exclusive, i.e., $V_k \cap V_\ell = \emptyset$ for $k \neq \ell$; and collectively exhaustive, i.e., $V = \bigcup V_k$.

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

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k \in \mathbb{N}_K, \tag{2.5}$$

we say that Π is *balanced* where ε represents the allowed maximum imbalance ratio.

In a partition Π, a hyperedge that has at least one vertex in a part is said to *connect* that part. The number of parts connected by a hyperedge h is called its *connectivity*, and is denoted by λ_h. Given a vertex partition Π of a hypergraph $H = (V, E)$, one can measure the *cutsize* metric induced by Π as

$$\chi(\Pi) = \sum_{h \in E} \mathsf{c}[h](\lambda_h - 1) \ . \tag{2.6}$$

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

Given $\varepsilon > 0$ and an integer $K > 1$, the standard hypergraph partitioning problem is defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [66].

A common variant of the above problem is the *multi-constraint hypergraph partitioning* [20, 44]. In this variant, each vertex has an associated vector of weights. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint for each weight. Let $\mathsf{w}[v, i]$ denote the C weights of a vertex v for $i \in \mathbb{N}_C$. In this variant, the balance criterion (2.5) is rewritten as

$$W_{k,i} \leq W_{avg,i}\,(1 + \varepsilon) \text{ for } k \in \mathbb{N}_K \text{ and } i \in \mathbb{N}_C, \tag{2.7}$$

Table 2.1: Real-world tensors used in the experiments.

Tensor	I_1	I_2	I_3	I_4	#nonzeros
Amazon	6.6M	2.4M	23K	-	1.3B
Delicious	1.4K	532K	17M	2.4M	140M
Flickr	731	319K	28M	1.6M	112M
Netflix	480K	17K	2K	-	100M
NELL	3.2M	301	638K	-	78M
NELL2	1.6M	297	338K	-	2M

where the ith weight $W_{k,i}$ of a part V_k is defined as the sum of the ith weights of the vertices in that part, i.e., $W_{k,i} = \sum_{v \in V_k} \mathsf{w}[v,i]$, and $W_{avg,i}$ represents the average part weight for the ith weight of all vertices, i.e., $W_{avg,i} = \sum_{v \in V} \mathsf{w}[v,i]/K$.

2.6 Tensor datasets

Most experiments throughout the manuscript involve six real-world sparse tensors whose sizes are shown in Table 2.1. Netflix tensor has $\text{user} \times \text{movie} \times \text{time}$ dimensions, which we formed from the data of the Netflix Prize competition [11]. In this tensor, nonzeros correspond to the user reviews for movies, and the 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 tensor from the Never Ending Language Learning (NELL) knowledge database of the "Read the Web" project [15], which consists of tuples of form $(\text{entity}, \text{relation}, \text{entity})$ 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. There is also a smaller version of this dataset from which we construct another tensor named NELL2. Delicious and Flickr are the datasets for the web-crawl of Delicious.com and Flickr.com during 2006 and 2007, which are formed by Görlitz et al. [30]. These datasets consist of tuples of form $(\text{time} \times \text{users} \times \text{resources} \times \text{tags})$; hence, we form 4-mode tensors out of these tuples. We obtained the Amazon review dataset from SNAP [67], which contains product review texts by users. We first processed this dataset with the standard text processing routines. We used the `nltk` package [13] in Python to tokenize the review text, to discard the stop words, to apply Porter stemmer, and to keep the words that are in the US, GB, or CA dictionaries. Afterwards, we retained only the words with at least five occurrences in the whole review set. Then, we created a three dimensional tensor whose dimensions correspond to the users, products, and retained words. Numerical values are set to the frequency of a word in a review.

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

2.7 Parallel compute environments

Most of the experiments in the manuscript were conducted on a shared memory and a separate distributed memory system, which are described in what follows. The shared memory system has two CPU sockets (Intel(R) Xeon(R) E5-2695 v3) each having 14 cores at a clock speed of 2.30GHz with Turbo Boost disabled. The system has a total memory of size 768GB and L1, L2, L3 caches of sizes of 32KB, 256KB, and 35MB, respectively. We call this system **Grunch** throughout the manuscript. The distributed memory system is an IBM Blue Gene/Q cluster. This system consists of 6 racks of 1024 nodes with each node having 16 GBs of memory and a 16-core IBM PowerPC A2 processor running at 1.6 GHz. This system is referred to as **Blue Gene/Q** in the rest of the document. Other compute environments used specific to a chapter are described within that chapter.

Part I

Parallel Sparse Tensor and Matrix Decompositions

Chapter 3

Parallel CP Decomposition of Sparse Tensors Using Dimension Trees

In this chapter, we investigate the parallelization of CP-ALS algorithm in shared and distributed memory environments for sparse tensors. For the shared memory computations, we propose a novel computational scheme that significantly reduces the computational cost of MTTKRP while providing an effective parallelization. We then perform theoretical analyses corresponding to the computational gains and the memory utilization of this scheme, which are also validated with experiments. Next, we propose a fine-grain distributed memory parallel CP-ALS algorithm, and compare it against a medium-grain variant [96]. Finally, we discuss effective partitioning routines for these algorithms.

The organization of the rest of the chapter is as follows. We explain the use of dimension trees to carry out MTTKRPs within CP-ALS in the next section. In Section 3.2, we discuss an effective shared and distributed memory parallelization of CP-ALS iterations. Afterwards, we introduce partitioning problems pertaining to distributed memory parallel performance, and use these partitioning methods in our experiments using real-world tensors. Section 3.3 provides an overview of the existing literature. Finally, we present experimental results to demonstrate the performance gains of our algorithms for shared and distributed memory parallelism over an efficient state of the art implementation in Section 3.4.

An earlier version of this work is published in [50], and the final version is currently under review [53].

3.1 Computing CP decomposition using dimension trees

As introduced in Section 2.4, a dimension tree partitions the mode indices of an N-dimensional tensor in a hierarchical manner for computing tensor decompositions efficiently. It was first used in the hierarchical Tucker format representing the hierarchical Tucker decomposition and SVD of a tensor [31]. Here, we propose a novel way of using dimension trees for computing the *standard* CP decomposition of tensors with a formulation that asymptotically reduces the computational cost. In doing so, we do not alter the original CP decomposition in any way. The reduction in the computational cost is made possible by storing partial TTV results, and hence by trading off more memory. A similar idea of reusing partial results without the use of a tree framework was moderately explored by Baskaran et

al. [9] for computing the Tucker decomposition of sparse tensors and by Phan et al. for computing the CP decomposition of dense tensors [82]. We adopt the dimension tree data structure to fully utilise the potential of reducing the cost of MTTKRP operations in computing the CP decomposition. If a dimension tree has the height $\lceil \log(N) \rceil$ with its first $\lfloor \log(N) \rfloor$ levels forming a complete binary tree, we call it a balanced binary dimension tree (BBDT). In Figure 3.1, we show a BBDT for 4 dimensions (associated with a sparse tensor described later).

3.1.1 Using dimension trees to perform successive TTVs

At each subiteration of the CP-ALS algorithm, using (2.4), $\boldsymbol{\mathcal{X}}$ is multiplied with the column vectors of matrices in $N-1$ modes in performing the MTTKRP. Some of these TTVs involve the same matrices as the preceding subiterations. As a series of TTVs can be done in any order, this opens up the possibility to factor out and reuse TTV results that are common in consecutive subiterations for reducing the computational cost. For instance, in the first subiteration of CP-ALS using a 4-mode tensor $\boldsymbol{\mathcal{X}}$, we compute $\boldsymbol{\mathcal{X}} \times_2 \mathbf{u}_r^{(2)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and eventually update $\mathbf{u}_r^{(1)}$ for each $r \in \mathbb{N}_R$, whereas in the second subiteration we compute $\boldsymbol{\mathcal{X}} \times_1 \mathbf{u}_r^{(1)} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and update $\mathbf{u}_r^{(2)}$. In these two subiterations, the matrices $\mathbf{U}^{(3)}$ and $\mathbf{U}^{(4)}$ remain unchanged, and both TTV steps involve the TTV of $\boldsymbol{\mathcal{X}}$ with $\mathbf{u}_r^{(3)}$ and $\mathbf{u}_r^{(4)}$. Hence, we can compute $\boldsymbol{\mathcal{Y}}_r = \boldsymbol{\mathcal{X}} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$, then reuse it in the first and the second subiterations as $\boldsymbol{\mathcal{Y}}_r \times_2 \mathbf{u}_r^{(2)}$ and $\boldsymbol{\mathcal{Y}}_r \times_1 \mathbf{u}_r^{(1)}$ to obtain the final TTV results for these modes.

We use dimension trees to systematically detect and reuse such partial results by associating a tree $\boldsymbol{\mathcal{T}}$ with an N-dimensional tensor $\boldsymbol{\mathcal{X}}$ as follows. With each node $t \in \boldsymbol{\mathcal{T}}$, we associate R tensors $T_1(t), \ldots, T_R(t)$. $T_r(t)$ corresponds to the TTV result $T_r(t) = \boldsymbol{\mathcal{X}} \times_{m_1} \mathbf{u}_r^{(m_1)} \times_{m_2} \cdots \times_{m_{|\mu'(t)|}} \mathbf{u}_r^{(m_{|\mu'(t)|})}$, where $m_i \in \mu'(t)$ are distinct elements of the inverse mode set $\mu'(t)$. Therefore, the inverse mode set $\mu'(t)$ corresponds to the set of modes in which TTV is performed on $\boldsymbol{\mathcal{X}}$ to form $T_r(t)$. For the root of the tree, $\mu'(\text{ROOT}(\boldsymbol{\mathcal{T}})) = \emptyset$; thus, all tensors of the root node correspond to the original tensor $\boldsymbol{\mathcal{X}}$. Since $\mu'(t) \supset \mu'(P(t))$ for a node t and its parent $P(t)$, tensors of $P(t)$ can be used as partial results to update the tensors of t. Let $\delta(t) = \mu'(t) \setminus \mu'(P(t))$. We can then compute each tensor of t from its parent's as $T_r(t) = T_r(P(t)) \times_{d_1} \mathbf{u}_r^{(d_1)} \times_{d_2} \cdots \times_{d_{|\delta(t)|}} \mathbf{u}_r^{(d_{|\delta(t)|})}$ for $\delta(t) = \{d_1, \ldots, d_{|\delta(t)|}\}$. This procedure is called DTREE-TTV and is shown in Algorithm 4. DTREE-TTV first checks if the tensors of t are already computed, and immediately returns if so. This happens, for example, when two children of t call DTREE-TTV on t consecutively, in which case for the second DTREE-TTV call, the tensors of t would be already computed. If the tensors of t are not already computed, DTREE-TTV on $P(t)$ is called first to make sure that $P(t)$'s tensors are up-to-date. Then, each $T_r(t)$ is computed by performing a TTV on the corresponding tensor $T_r(P(t))$ of the parent. We use the notation $T_:(t)$ to denote all R tensors of a node t.

Algorithm 4 DTREE-TTV: Dimension tree-based TTV with R vectors in each mode

Input: t: A node of the dimension tree
Output: Tensors $T_:(t)$ of t are computed.
1: **if** EXISTS($T_:(t)$) **then**
2: **return** ▶ Tensors $T_:(t)$ are already computed.
3: DTREE-TTV($P(t)$) ▶ Compute the parent's tensors $T_:(P(t))$ first.
4: **for** $r = 1, \ldots, R$ **do** ▶ Now update all tensors of t using parent's.
5: $T_r(t) \leftarrow T_r(P(t)) \times_{d_1} \mathbf{u}_r^{(d_1)} \times_{d_2} \cdots \times_{d_{|\delta(t)|}} \mathbf{u}_r^{(d_{|\delta(t)|})}$

Algorithm 5 DTREE-CP-ALS: Dimension tree-based CP-ALS algorithm

Input: $\mathcal{X}$: An N-mode tensor
R: The rank of CP decomposition
Output: $[\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\mathcal{X}$

1: $\mathcal{T} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\mathcal{X}$) ▶ The tree has leaves $\{\mathcal{L}_1, \ldots, \mathcal{L}_N\}$.
2: **for** $n = 2 \ldots N$ **do**
3: $\quad \mathbf{W}^{(n)} \leftarrow {\mathbf{U}^{(n)}}^T \mathbf{U}^{(n)}$
4: **repeat**
5: $\quad$ **for** $n = 1, \ldots, N$ **do**
6: $\quad\quad$ **for all** $t \in \mathcal{T}$ **do**
7: $\quad\quad\quad$ **if** $n \in \mu'(t)$ **then**
8: $\quad\quad\quad\quad$ DESTROY($T_:(t)$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
9: $\quad\quad$ DTREE-TTV($\mathcal{L}_n$) ▶ Perform the TTVs for the leaf node tensors.
10: $\quad\quad$ **for** $r = 1, \ldots, R$ **do** ▶ Form $\mathbf{M}^{(n)}$ column-by-column (done implicitly).
11: $\quad\quad\quad \mathbf{M}^{(n)}(:, r) \leftarrow T_r(\mathcal{L}_n)$ ▶ $T_r(\mathcal{L}_n)$ is a vector of size I_n for the leaf node $\mathcal{L}_n$.
12: $\quad\quad \mathbf{H}^{(n)} \leftarrow *_{i \neq n} \mathbf{W}^{(i)}$
13: $\quad\quad \mathbf{U}^{(n)} \leftarrow \mathbf{M}^{(n)} {\mathbf{H}^{(n)}}^\dagger$
14: $\quad\quad \lambda \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
15: $\quad\quad \mathbf{W}^{(n)} \leftarrow {\mathbf{U}^{(n)}}^T \mathbf{U}^{(n)}$
16: **until** converge is achieved or the maximum number of iterations is reached.
17: **return** $[\![\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

3.1.2 Dimension tree-based CP-ALS algorithm

At the nth subiteration of Algorithm 1, we need to compute $\mathcal{X} \times_{i \neq n} \mathbf{u}_r^{(i)}$ for all $r \in \mathbb{N}_R$ in order to form $\mathbf{M}^{(n)}$. Using a dimension tree $\mathcal{T}$ with leaves $\mathcal{L}_1, \ldots, \mathcal{L}_N$, we can perform this simply by executing DTREE-TTV($\mathcal{L}_n$), after which the rth tensor of $\mathcal{L}_n$ provides the rth column of $\mathbf{M}^{(n)}$. Once $\mathbf{M}^{(n)}$ is formed, the remaining steps follow as before. We show the whole CP-ALS using a dimension tree in Algorithm 5. At Line 1, we construct a dimension tree $\mathcal{T}$ with the leaf order $\mathcal{L}_1, \ldots, \mathcal{L}_N$ obtained from a post-order traversal of $\mathcal{T}$. This tree can be constructed in any way that respects the properties of a dimension tree described in Section 3.1; but for our purposes we assume that it is formed as a BBDT. At Line 8 within the nth subiteration, we destroy all tensors of a node t if its set of multiplied modes $\mu'(t)$ involve n, as in this case its tensors involve multiplication using the old value of $\mathbf{U}^{(n)}$ which is about to change. Note that this step destroys the tensors of all nodes not lying on a path from $\mathcal{L}_n$ to the root. Afterwards, DTREE-TTV is called at Line 9 for the leaf node $\mathcal{L}_n$ to compute its tensors. This step computes (or reuses) the tensors of all nodes from the path from $\mathcal{L}_n$ to the root. Next, the rth column of $\mathbf{M}^{(n)}$ is formed using $T_r(\mathcal{L}_n)$ for $r \in \mathbb{N}_R$. Once $\mathbf{M}^{(n)}$ is ready, $\mathbf{H}^{(n)}$ and $\mathbf{U}^{(n)}$ are computed as before, after which $\mathbf{U}^{(n)}$ is normalized.

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

Theorem 1. *Let $\mathcal{X}$ be an N-mode tensor. The total number of TTVs at each iteration of Algorithm 5 using a BBDT is at most $RN\lceil \log N \rceil$.*

Proof. As we assume that the sequence $\mathcal{L}_1, \ldots, \mathcal{L}_N$ is obtained from a post-order traversal of $\mathcal{T}$, for each internal node t, the subtree rooted at t has the leaves $\mathcal{L}_i, \mathcal{L}_{i+1}, \ldots, \mathcal{L}_{i+k-1}$ corresponding to k

consecutive mode indices for some positive integers i and k. As we have $\mu(\mathcal{L}_i) = i$ for the ith leaf node, we obtain $\mu(t) = \{i, i+1, \ldots, i+k-1\}$ due to the second property of dimension trees. As a result, for each leaf node $\mathcal{L}_{i+k'}$ for $0 \leq k' < k$, we have $i + k' \in \mu(t)$; hence $i + k' \notin \mu'(t)$ as $\mu'(t) = \mathbb{N}_N \setminus \mu(t)$. Therefore, within an iteration of Algorithm 5 the tensors of t get computed at the ith subiteration, stay valid (not destroyed) and get reused until the $i+k-1$th subiteration, and finally get destroyed in the following subiteration. Once destroyed, the tensors of t are never recomputed in the same iteration, as all the modes associated with the leaf tensors in its subtree are processed (which are the only nodes that can reuse the tensors of t). As a result, in one CP-ALS iteration, tensors of every tree node (except the root) get to be computed and destroyed exactly once. Therefore, the total number of TTVs in one iteration becomes the sum of the number of TTVs performed to compute the tensors of each node in the tree once. In computing its tensors, every node t has R tensors, and for each tensor it performs TTVs for each dimension in the set $\delta(t)$ in Algorithm 4, except the root node, whose tensors all are equal to $\boldsymbol{\mathcal{X}}$ and never change. Therefore, we can express the total number of TTVs performed within a CP-ALS iteration due to one of these R tensors as

$$\sum_{t \in \boldsymbol{\mathcal{T}} \setminus \{\textsc{Root}(\boldsymbol{\mathcal{T}})\}} |\delta(t)| = \sum_{t \in \boldsymbol{\mathcal{T}} \setminus \{\textsc{Root}(\boldsymbol{\mathcal{T}})\}} |\mu(P(t)) \setminus \mu(t)|. \tag{3.1}$$

Since every non-leaf node t in a BBDT has exactly two children, say t_1 and t_2, we obtain $|\mu(t) \setminus \mu(t_1)| + |\mu(t) \setminus \mu(t_2)| = |\mu(t)|$, as $\mu(t)$ is partitioned into two sets $\mu(t_1)$ and $\mu(t_2)$. With this observation, we can reformulate (3.1) as

$$\sum_{t \in \boldsymbol{\mathcal{T}} \setminus \{\textsc{Root}(\boldsymbol{\mathcal{T}})\}} |\mu(P(t)) \setminus \mu(t)| = \sum_{t \in \boldsymbol{\mathcal{T}} \setminus \textsc{Leaves}(\boldsymbol{\mathcal{T}})} |\mu(t)| \,. \tag{3.2}$$

Note that in constructing a BBDT, at the root node we start with the mode set $\mu(\textsc{Root}(\boldsymbol{\mathcal{T}})) = \mathbb{N}_N$. Then, at each level $l > 0$, we form the mode sets of the nodes at level l by partitioning the mode sets of their parents at level $l-1$. As a result, at each level l, each dimension $n \in \mathbb{N}_N$ can appear in only one set $\mu(t)$ for a node t belonging to the level l of the BBDT. With this observation in mind, rewriting (3.2) by iterating over nodes by levels of the BBDT yields

$$\begin{aligned} \sum_{t \in \boldsymbol{\mathcal{T}} \setminus \textsc{Leaves}(\boldsymbol{\mathcal{T}})} |\mu(t)| &= \sum_{l=1}^{\lceil \log N \rceil} \sum_{t \in \boldsymbol{\mathcal{T}} \setminus \textsc{Leaves}(\boldsymbol{\mathcal{T}}), \textsc{Level}(t) = l} |\mu(t)| \\ &\leq \sum_{l=1}^{\lceil \log N \rceil} N = N \lceil \log N \rceil \,. \end{aligned}$$

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

Note in comparison that the traditional scheme [98] incurs $R(N-1)$ TTVs in each mode, and $RN(N-1)$ TTVs in total in a CP-ALS iteration. This yields a factor of $(N-1)/\log N$ reduction in the number performed of TTVs using dimension trees. We would like to stress that in terms of the actual computational cost, this corresponds to a lower bound on the expected speedup for the following reason. As the tensor is multiplied in different dimensions, resulting tensors are expected to have many index overlaps, effectively reducing their number of nonzeros and rendering subsequent

TTVs significantly cheaper. This renders using a BBDT much more effective as it avoids repeating such expensive TTVs at the higher levels of the dimension tree by reusing partial results. In the light of this observation, we argue the actual obtained speedup the following way. On one extreme, multiplying tensor in dimensions might create no or few index overlaps, which makes the cost of each TTV approximately equal, yielding the speedup factor of approximately $N/\log N$. On the other extreme, the first TTVs performed the original tensor may drastically reduce the number of tensor elements so that the cost of the subsequent TTVs becomes negligible. The traditional scheme multiplies the original tensor N times, once per dimension, wheres a BBDT suffices with 2 such TTVs as the root node has only two children. Therefore, we expect in practice the actual speedup to be between these two extremes depending on the sparsity of the tensor, and having more speedup with higher index overlap after multiplications.

For sparse tensors, one key idea we use for obtaining high performance is performing TTVs for all R tensors $T_{:}(t)$ of a node $t \in \mathcal{T}$ in a *vectorized* manner. We illustrate this on a 4-dimensional tensor $\mathcal{X}$ and a BBDT, and for clarity, we put the mode set $\mu(t)$ of each tree node t in the subscript, as in t_{1234}. Let t_{1234} represent the root of the BBDT with $T_r(t_{1234}) = \mathcal{X}$ for all $r \in \mathbb{N}_R$. The two children of t_{1234} are t_{12} and t_{34} with the corresponding tensors $T_r(t_{12}) = T_r(t_{1234}) \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and $T_r(t_{34}) = T_r(t_{1234}) \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$, respectively. Since $T_r(t_{1234})$s are identical for all $r \in \mathbb{N}_R$, the nonzero pattern of all tensors $T_r(t_{12})$ are identical for all $r \in \mathbb{N}_R$. The same is true for $T_r(t_{34})$s. The same argument applies to the children t_1 and t_2 of t_{12}, as well as t_3 and t_4 of t_{34}. As a result, each node in the tree involves R tensors with identical nonzero patterns. This opens up two possibilities in terms of efficiency. First, it is sufficient to compute only one set of nonzero indices for each node $t \in \mathcal{T}$ to represent the nonzero structure of all of its tensors, which reduces the computational and memory cost by a factor of R. Second, we can perform the TTVs for all tensors at once in a *vectorized* manner by modifying (2.1) to perform R TTVs of the form $\mathcal{Y}^{(r)} \leftarrow \mathcal{X}^{(r)} \times_d \mathbf{v}_r$ for $\mathbf{V} = [\mathbf{v}_1 | \dots | \mathbf{v}_R] \in \mathbb{R}^{I_d \times R}$ as follows:

$$\mathbf{y}^{(:)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N} = \sum_{j=1}^{I_d} \mathbf{V}(j,:) * \mathbf{x}^{(:)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N}, \tag{3.3}$$

where $\mathbf{y}^{(:)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N}$ and $\mathbf{x}^{(:)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N}$ are vectors of size R with elements $y^{(r)}_{i_1,\dots,i_{d-1},1,i_{d+1},\dots,i_N}$ and $x^{(r)}_{i_1,\dots,i_{d-1},j,i_{d+1},\dots,i_N}$ in $\mathcal{Y}^{(r)}$ and $\mathcal{X}^{(r)}$, for all $r \in \mathbb{N}_R$. We call this operation tensor-times-multiple-vector multiplication (TTMV) as R column vectors of $\mathbf{V}$ are multiplied simultaneously with R tensors of identical nonzero patterns. We can similarly extend this formula to the multiplication $\mathcal{Z}^{(r)} \leftarrow \mathcal{Y}^{(r)} \times_{d_2} \mathbf{W} = (\mathcal{X}^{(r)} \times_{d_1} \mathbf{v}_r) \times_{d_2} \mathbf{w}_r$ in two modes d_1 and d_2, $d_1 < d_2$, with matrices $\mathbf{V} \in \mathbb{R}^{I_{d_1} \times R}$ and $\mathbf{W} \in \mathbb{R}^{I_{d_2} \times R}$ as

$$\begin{aligned}\mathbf{z}^{(:)}_{i_1,\dots,i_{d_1-1},1,i_{d_1+1},\dots,i_{d_2-1},1,i_{d_2+1},\dots,i_N} &= \sum_{j_2=1}^{I_{d_2}} \mathbf{W}(j,:) * \mathbf{y}^{(:)}_{i_1,\dots,i_{d_1-1},1,i_{d_1+1},\dots,i_{d_2-1},j_2,i_{d_2+1},\dots,i_N} \\ &= \sum_{(j_1,j_2)=(1,1)}^{(I_{d_1},I_{d_2})} \mathbf{V}(j_1,:) * \mathbf{W}(j_2,:) * \mathbf{x}^{(:)}_{i_1,\dots,i_{d_1-1},j_1,i_{d_1+1},\dots,i_{d_2-1},j_2,i_{d_2+1},\dots,i_N}. \end{aligned} \tag{3.4}$$

The formula similarly generalizes to d dimensions for $d > 2$ where each dimension adds another Hadamard product with a corresponding matrix row. This "thick" mode of operation provides a significant performance gain thanks to the increase in locality. Also, performing TTVs in this manner in CP-ALS effectively reduces the $RN\lceil \log N \rceil$ TTVs required in Theorem 1 to $N\lceil \log N \rceil$ TTMV

calls within an iteration. In our approach, for each node $t \in \boldsymbol{\mathcal{T}}$, we store a single set $\mathcal{I}_t$ containing index tuples of the form $(i_1, \ldots, i_N)$ representing the nonzeros $x^{(r)}_{i_1,\ldots,i_N} \in T_r(t)$ for all $r \in \mathbb{N}_R$. Also, for each such index tuple we hold a vector of size R corresponding to the values of this nonzero in each one of R tensors, and we denote this value vector as $\mathcal{V}_t(i_1, \ldots, i_N)$.

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

Symbolic TTV

For simplicity, we proceed with describing how to perform the symbolic TTV using the same 4-dimensional tensor $\boldsymbol{\mathcal{X}}$ and BBDT. However, the approach naturally generalizes to any N-dimensional tensor and dimension tree.

The first information we need is the set of nonzero indices $\mathcal{I}_t$ for each node t in a dimension tree, which we determine as follows. As mentioned, the two children t_{12} and t_{34} of t_{1234} have the corresponding tensors $T_r(t_{12}) = T_r(t_{1234}) \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$ and $T_r(t_{34}) = T_r(t_{1234}) \times_1 \mathbf{u}_r^{(1)} \times_2 \mathbf{u}_r^{(2)}$, respectively. Using (3.4), the nonzero indices of $T_r(t_{12})$ and $T_r(t_{34})$ take the form $(i, j, 1, 1)$ and $(1, 1, k, l)$, respectively (in practice, tensor indices in the multiplied dimensions are always 1; hence we omit storing them), and such a nonzero index exist in these tensors only if there exists a nonzero $x_{i,j,k,l} \in T_r(t_{1234})$. Determining the set $\mathcal{I}_{t_{12}}$ (or $\mathcal{I}_{t_{34}}$) can simply be done by starting with a list $\mathcal{I}_{t_{1234}}$ of tuples, then replacing each tuple (i, j, k, l) in the list with the tuple (i, j) (or with (k, l) for $\mathcal{I}_{t_{34}}$). This list may contain duplicates, which can efficiently be eliminated by sorting. Once the set of nonzeros for t_{12} (or t_{34}) is determined, we proceed to detect the nonzero patterns of its children t_1 and t_2 (or, t_3 and t_4).

To be able to carry out the numerical calculations (3.3) and (3.4) for each nonzero at a node t, we need to identify the set of nonzeros of the parent node $P(t)$'s tensors that contribute to this nonzero. Specifically, at t_{12}, for each nonzero index $(i, j) \in \mathcal{I}_{t_{12}}$ we need to bookmark all nonzero index tuples of t_{1234} of the form $(i, j, k, l) \in \mathcal{I}_{t_{1234}}$, as it is this set of nonzeros of $T_r(t_{1234})$ that contribute to the nonzero of $T_r(t_{12})$ with index $(i, j, 1, 1)$ in (3.4). Therefore, for each such index tuple of t_{12} we need a *reduction set* $\mathcal{R}_{t_{12}}(i, j)$ which keeps track of all such index tuples of the parent. We determine these sets simultaneously with $\mathcal{I}_t$. In Figure 3.1, we illustrate a sample BBDT for a 4-dimensional sparse tensor with $\mathcal{I}_t$, shown with arrays, and $\mathcal{R}_t$, shown using arrows.

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

Theorem 2. *Let $\boldsymbol{\mathcal{X}}$ be an N-mode sparse tensor. The total number of index arrays in a BBDT of $\boldsymbol{\mathcal{X}}$ is at most $N(\lceil \log N \rceil + 1)$.*

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

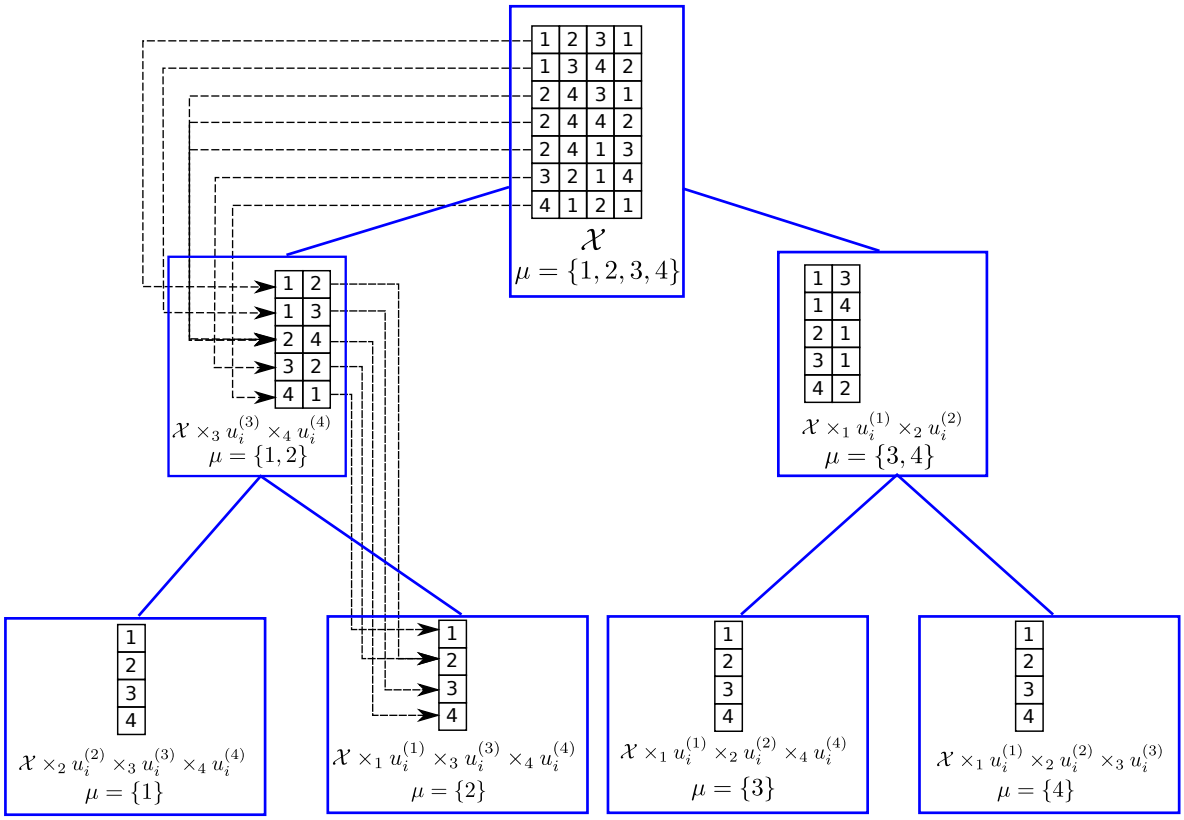


Figure 3.1: BBDT of a 4-dimensional sparse tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{4\times4\times4\times4}$ having 7 nonzeros. Each closed box refers to a tree node. Within each node, the index array and the mode set corresponding to that node are given. The reduction sets of two nodes in the tree are indicated with the dashed lines.

Theorem 2 shows that the storage requirement for the tensor indices of a BBDT cannot exceed $(\lceil \log N \rceil + 1)$-times the size of the original tensor in the coordinate format (which has N index arrays of size $\text{nnz}(\boldsymbol{\mathcal{X}})$), yielding the overall worst-case memory cost $O(\text{nnz}(\boldsymbol{\mathcal{X}})N(\lceil \log N \rceil + 1))$. Indeed, this is a pessimistic estimate for real-world tensors, as with significant index overlap in non-root tree tensors, the total memory cost could reduce to as low as $O(\text{nnz}(\boldsymbol{\mathcal{X}})N)$. This renders the approach very suitable for higher dimensional tensors. In computing the symbolic TTV, we sort $|\mu(t)|$ index arrays for each node $t \in \boldsymbol{\mathcal{T}}$. In addition, for each node t of a BBDT, we sort an extra array to determine the reduction set $\mathcal{R}_t$. In the worst case, each array can have up to $\text{nnz}(\boldsymbol{\mathcal{X}})$ elements. Therefore, combining with the number of index arrays as given in Theorem 2, the overall worst case cost of sorting becomes $O(N(\lceil \log N \rceil + 1) + 2N - 1)\text{nnz}(\boldsymbol{\mathcal{X}}) \log(\text{nnz}(\boldsymbol{\mathcal{X}})) = O(N \log N \ \text{nnz}(\boldsymbol{\mathcal{X}}) \log(\text{nnz}(\boldsymbol{\mathcal{X}}))$. We note, however, that both the total index array size and sorting cost are pessimistic overestimates, since the nonzero structure of real-world tensors exhibits significant locality in indices. For example, on two tensors from our experiments (Delicious and Flickr), we observed a reduction factor of 2.57 and 5.5 in the number of nonzeros of the children of the root of the BBDT. Consequently, the number of nonzeros in a node's tensors reduces dramatically as we approach to the leaves. In comparison, existing approaches [95] sort the original tensor once with a cost of $O(N\text{nnz}(\boldsymbol{\mathcal{X}}) \log(\text{nnz}(\boldsymbol{\mathcal{X}})))$ at the expense of computing TTVs from scratch in each CP-ALS iteration.

Symbolic TTV is a one-time computation whose cost is amortized. Normally, choosing an appropriate rank R for a sparse tensor $\boldsymbol{\mathcal{X}}$ requires several executions of CP-ALS. Also, CP-ALS is known to be sensitive to the initialization of factor matrices; therefore, it is often executed with multiple initializations [58]. In all of these use cases, the tensor $\boldsymbol{\mathcal{X}}$ is fixed; therefore, the symbolic TTV is required only once. Moreover, CP-ALS usually have a number of iterations which involve many costly numeric TTV calls. As a result, the cost of the subsequent numeric TTV calls over many iterations and many CP-ALS executions easily amortizes that of this symbolic preprocessing. Nevertheless, in case of need, this step can efficiently be parallelized in multiple ways. First, symbolic TTV is essentially a sorting of multiple index arrays; hence, one can use parallel sorting methods. Second, the BBDT structure naturally exposes a coarser level of parallelism; once a node's symbolic TTV is computed, one can proceed with those of its children in parallel, and process the whole tree in this way. Finally, in a distributed memory setting where we partition the tensor to multiple processes, each process can perform the symbolic TTV on its local tensor in parallel. We benefit only from this parallelism in our implementation.

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

Theorem 3. *For an N-mode tensor $\boldsymbol{\mathcal{X}}$, the total number of tree nodes whose values are not destroyed is at most $\lceil \log N \rceil$ at any instant of Algorithm 5 using a BBDT.*

Proof. Note that at the beginning of the nth subiteration of Algorithm 5, the tensors of each node $t \in \boldsymbol{\mathcal{T}}$ involving n in $\mu'(t)$ are destroyed at Line 8. These are exactly the tensors that do not lie in the path from the leaf $\mathcal{L}_n$ to the root, as they do not involve n in their mode set μ. The TTMV result for $\mathcal{L}_n$ depends only on the nodes on this path from $\mathcal{L}_n$ to the root; therefore, at the end of the nth subiteration, only the tensors of the nodes on this path will be computed using Algorithm 4. As this path length cannot exceed $\lceil \log N \rceil$ in a BBDT, the number of nodes whose values are not destroyed cannot exceed $R\lceil \log N \rceil$ at any instant of Algorithm 5. □

Theorem 3 puts a nice upper bound of $\lceil \log N \rceil$ on the maximum number of allocated value arrays, thus on the maximum memory utilization due to tensor values, in DTREE-CP-ALS. Note that in the worst case, each tensor value array may have up to $\text{nnz}(\mathcal{X})$ elements, requiring $\text{nnz}(\mathcal{X})R\lceil \log N \rceil$ memory to store its values in total. Indeed, having significant nonzero index overlaps after TTVs may significantly reduce this cost, as the size of a value array is proportional to the number of nonzero elements of the associated node's tensors.

3.2 Parallel CP-ALS for sparse tensors using dimension trees

We first present shared memory parallel algorithms involving efficient parallelization of the dimension tree-based TTMVs in Section 3.2.1. Later, in Section 3.2.2, we present distributed memory parallel algorithms that use this shared memory parallelization.

3.2.1 Shared memory parallelism

Algorithm 6 SMP-DTREE-TTV

Input: t: A dimension tree node/tensor
Output: Numerical values $\mathcal{V}_t$ are computed.

```
1:  if EXISTS(V_t) then
2:      return                                   ▶ Numerical values V_t are already computed.
3:  SMP-DTREE-TTV(P(t))                          ▶ Compute the parent's values V_{P(t)} first.
4:  parallel for all (i_1, ..., i_N) ∈ I_t do    ▶ Compute each V_t(i_1, ..., i_N) in parallel.
5:      V_t(i_1, ..., i_N) ← zeros(1, R)         ▶ Initialize with a zero vector of size 1 × R.
6:      for all (j_1, ..., j_N) ∈ R(i_1, ..., i_N) do   ▶ Perform updates using elements in R.
7:          r ← V_{P(t)}(j_1, ..., j_N)
8:          for all d ∈ δ(t) do                  ▶ Multiply the vector r with corresponding matrix rows.
9:              r ← r * U^(d)(j_d, :)
10:         V_t(i_1, ..., i_N) ← V_t(i_1, ..., i_N) + r   ▶ Do the update due to element (j_1, ..., j_N).
```

After forming the dimension tree $\mathcal{T}$ for a tensor $\mathcal{X}$ with symbolic structures $\mathcal{I}_t$ and $\mathcal{R}_t$ for all tree nodes, we can perform numeric TTMV computations in parallel. In Algorithm 6, we provide the shared memory parallel TTMV algorithm, called SMP-DTREE-TTV, for a node t of a dimension tree. The goal of SMP-DTREE-TTV is to compute the tensor values $\mathcal{V}_t$ for a given node t. Similar to Algorithm 4, it starts by checking if $\mathcal{V}_t$ is already computed, and returns immediately in that case. Otherwise, it calls SMP-DTREE-TTV on the parent node $P(t)$ to make sure that parent's tensor values $\mathcal{V}_{P(t)}$ are available. Once $\mathcal{V}_{P(t)}$ is ready, the algorithm proceeds with computing $\mathcal{V}_t$ for each nonzero index $(i_1, \ldots, i_N) \in \mathcal{I}_t$. As for each index $(i_1, \ldots, i_N) \in \mathcal{I}_t$, the reduction set is defined during the symbolic TTV, the $\mathcal{V}_t$ can independently be updated in parallel. In performing this update, for each element $\mathcal{V}_{P(t)}(j_1, \ldots, j_N)$ of the parent, the algorithm multiplies this vector with the rows of the corresponding matrices of the TTMV in $\delta(t)$ of t, then adds it to $\mathcal{V}_t(i_1, \ldots, i_N)$.

For shared memory parallel CP-ALS, we replace Line 9 with the call SMP-DTREE-TTV$(\mathcal{L}_n)$ in Algorithm 5. The parallelization of the rest of the computations is trivial. In computing the matrices $\mathbf{W}^{(n)}$ and $\mathbf{U}^{(n)}$ at Lines 3, 13 and 15, we use parallel dense BLAS kernels. Computing the matrix $\mathbf{H}^{(n)}$ at Line 12 and normalizing the columns of $\mathbf{U}^{(n)}$ are embarrassingly parallel element-wise matrix operations. We skip the details of the parallel convergence check whose cost is negligible.

3.2.2 Distributed memory parallelism

Parallelizing CP-ALS in a distributed memory setting involves defining unit parallel tasks, data elements, and their interdependencies. Following to this definition, we partition and distribute tensor elements as well as factor matrices to all available processes. We provide a *fine-grain* and a *medium-grain* parallel task model together with the associated distributed memory parallel algorithms. We start the discussion with the following straightforward lemma that enables us to distribute tensor nonzeros for parallelization.

Lemma 1 (Distributive property of tensor-times-vector multiplies)**.** *Let $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ be tensors in $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with $\mathcal{X} = \mathcal{Y} + \mathcal{Z}$. Then, for any $n \in \mathbb{N}_N$ and $\boldsymbol{u} \in \mathbb{R}^{I_n}$ $\mathcal{X} \times_n \boldsymbol{u} = \mathcal{Y} \times_n \boldsymbol{u} + \mathcal{Z} \times_n \mathbf{u}$ holds.*

Proof. Using (2.1) we express the element-wise result of $\mathcal{X} \times_n \mathbf{u}$ as

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

□

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

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

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

Fine-grain parallelism

Corollary 1 allows us to partition the tensor $\mathcal{X}$ in the sum form $\mathcal{X}_1 + \cdots + \mathcal{X}_P$ for any $P > 1$, then perform TTMVs on each tensor partition $\mathcal{X}_p$ independently, finally sum up these results to obtain the TTMV result for $\mathcal{X}$ multiplied in $N - 1$ modes. As $\mathcal{X}$ is sparse, an intuitive way to achieve this decomposition is by partitioning its nonzeros to P tensors where P is the number of available distributed processes. This way, for any dimension n, we can perform the TTMV of $\mathcal{X}_p$ with the columns of the set of factor matrices $\{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\}$ in all modes except n. This yields a "local" matrix $\mathbf{M}_p^{(n)}$ at each process p, and all these local matrices must subsequently be "assembled" by summing up their rows corresponding to the same row indices. In order to perform this assembling of rows, we also partition the rows of matrices $\mathbf{M}^{(n)}$ so that each row is "owned" by a process that is

responsible for holding the final row sum. We represent this partition with a vector $\boldsymbol{\sigma}^{(n)} \in \mathbb{R}^{I_n}$ where $\sigma^{(n)}(i) = p$ implies that the final value of $\mathbf{M}^{(n)}(i,:)$ resides at the process p. We assume the same partition $\boldsymbol{\sigma}^{(n)}$ on the corresponding factor matrices $\mathbf{U}^{(n)}$, as this enables each process to compute the rows of $\mathbf{U}^{(n)}$ that it owns using the rows of $\mathbf{M}^{(n)}$ belonging to that process without incurring any communication.

Algorithm 7 DMP-DTREE-CP-ALS: Dimension tree-based CP-ALS algorithm

Input: $\boldsymbol{\mathcal{X}}_p$: An N-mode tensor
$I_p^{(n)}$: The index set with elements $i \in I_p^{(n)}$ where $\sigma^{(n)}(i) = p$
$F_p^{(n)}$: The set of all unique indices of $\boldsymbol{\mathcal{X}}_p$ in mode n
$\mathbf{U}^{(1)}(F_p^{(1)},:), \ldots, \mathbf{U}^{(N)}(F_p^{(N)},:)$: Distributed initial matrices (rows needed by process p)
R: The rank of CP decomposition
Output: $[\lambda; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]$: Rank-$R$ CP decomposition of $\boldsymbol{\mathcal{X}}_p$ with distributed $\mathbf{U}^{(n)}$

1: $\boldsymbol{\mathcal{T}} = \{\mathcal{L}_1, \ldots, \mathcal{L}_N, \mathcal{I}_1, \ldots\} \leftarrow$ CONSTRUCT-DIMENSION-TREE($\boldsymbol{\mathcal{X}}_p$)
2: **for** $n = 2 \ldots N$ **do**
3: $\quad \mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($[\mathbf{U}^{(n)}(I_p^{(n)},:)]^T\ \mathbf{U}^{(n)}(I_p^{(n)},:)$)
4: **repeat**
5: $\quad$ **for** $n = 1 \ldots N$ **do**
6: $\quad\quad$ **for all** $t \in \boldsymbol{\mathcal{T}}$ **do** ▶ Invalidate tree tensors that are multiplied in mode n.
7: $\quad\quad\quad$ **if** $n \in \mu'(t)$ **then**
8: $\quad\quad\quad\quad$ DESTROY($\mathcal{V}_t$) ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
9: $\quad\quad$ SMP-DTREE-TTV($\mathcal{L}_n$) ▶ Perform the TTVs for the leaf node tensors.
10: $\quad\quad$ $\mathbf{M}^{(n)}(F_p^{(n)},:) \leftarrow \mathcal{V}_{\mathcal{L}_n}(:)$ ▶ Form $\mathbf{M}^{(n)}$ using leaf tensors (done implicitly).
11: $\quad\quad$ COMM-FACTOR-MATRIX($\mathbf{M}^{(n)}$, "*fold*", $F_p^{(d)}, I_p^{(d)}, \sigma^{(d)}$) ▶ Assemble $\mathbf{M}^{(n)}(I_p^{(n)},:)$.
12: $\quad\quad$ $\mathbf{H}^{(n)} \leftarrow *_{i \neq n} \mathbf{W}^{(i)}$
13: $\quad\quad$ $\mathbf{U}^{(n)}(I_p^{(n)},:) \leftarrow \mathbf{M}^{(n)}(I_p^{(n)},:)\mathbf{H}^{(n)\dagger}$
14: $\quad\quad$ $\lambda \leftarrow$ COLUMN-NORMALIZE($\mathbf{U}^{(n)}$)
15: $\quad\quad$ COMM-FACTOR-MATRIX($\mathbf{U}^{(n)}$, "*expand*", $F_p^{(d)}, I_p^{(d)}, \sigma^{(d)}$) ▶ Send/Receive $\mathbf{U}^{(n)}$.
16: $\quad\quad$ $\mathbf{W}^{(n)} \leftarrow$ ALL-REDUCE($[\mathbf{U}^{(n)}(I_p^{(n)},:)]^T\ \mathbf{U}^{(n)}(I_p^{(n)},:)$)
17: **until** converge is achieved or the maximum number of iterations is reached.

This approach amounts to a fine-grain parallelism where each fine-grain computational task corresponds to performing TTMV operations due to a nonzero element $x_{i_1,\ldots,i_N} \in \boldsymbol{\mathcal{X}}$. Specifically, according to (3.4), the process p needs the matrix rows $\mathbf{U}^{(1)}(i_1,:), \ldots, \mathbf{U}^{(N)}(i_N,:)$ for each nonzero $x_{i_1,\ldots,i_N}$ in its local tensor $\boldsymbol{\mathcal{X}}_p$ in order to perform its local TTMVs. For each dimension n, we represent the union of all these "required" row indices for the process p by $F_p^{(n)}$. Similarly, we represent the set of "owned" rows by the process p by $I_p^{(n)}$. In this situation, the set $F_p^{(n)} \setminus I_p^{(n)}$ correspond to the rows of $\mathbf{M}^{(n)}$ for which the process p generates a partial TTMV result, which need to be sent to their owner processes. Equally, it represents the set of rows of $\mathbf{U}^{(n)}$ that are not owned by the process p and are needed in its local TTMVs according to (3.4). These rows of $\mathbf{U}^{(n)}$ are similarly to be received from their owners in order to carry out the TTMVs at process p. Hence, a "good" partition in general involves a significant overlap of $F_p^{(n)}$ and $I_p^{(n)}$ to minimize the cost of communication.

In Algorithm 7, we describe the fine-grain parallel algorithm that operates in this manner at process p. The elements $\boldsymbol{\mathcal{X}}_p$, $I_p^{(n)}$, and $F_p^{(n)}$ are determined in the partitioning phase, and are provided as input to the algorithm. Each process starts with the subset $F_p^{(n)}$ of rows of each factor matrix $\mathbf{U}^{(n)}$ that it needs for its local computations. Similar to Algorithm 5, at Line 1 we start by forming the dimension tree for the local tensor $\boldsymbol{\mathcal{X}}_p$. We then compute the matrices $\mathbf{W}^{(n)}$ corresponding to $\mathbf{U}^{(n)T}\mathbf{U}^{(n)}$ using

the initial factor matrices. We do this step in parallel in which each process computes the local contribution $[\mathbf{U}^{(n)}(I_p^{(n)},:)]^T\ \mathbf{U}^{(n)}(I_p^{(n)},:)$ due to its owned rows. Afterwards, we perform an ALL-REDUCE communication to sum up these local results to obtain a copy of $\mathbf{W}^{(n)}$ at each process. The cost of this communication is typically negligible as $\mathbf{W}^{(n)}$ is a small matrix of size $R \times R$. The main CP-ALS subiteration for mode n begins with destroying tensors in the tree that will become invalid after updating $\mathbf{U}^{(n)}$. Next, we perform SMP-DTREE-TTV on the leaf node $\mathcal{L}_n$, and obtain the "local" matrix

Algorithm 8 COMM-FACTOR-MATRIX: Communication routine for factor matrices

Input: $\mathbf{M}$: Distributed factor matrix to be communicated
$comm$: The type of communication. "*fold*" sums up the partial results in owner processes, whereas "*expand*" communicates the final results from owners to all others.
F_p: The rows used by process p
I_p: The rows owned by process p
σ: The ownership of each row of $\mathbf{M}$
Output: Rows of $\mathbf{M}$ are properly communicated.
1: **if** $comm =$ "*fold*" **then**
2: **for all** $i \in F_p \setminus I_p$ **do** ▶ Send non-owned rows to their owners.
3: Send $\mathbf{M}(i,:)$ to the process $\sigma(i)$.
4: **for all** $i \in I_p$ **do** ▶ Gather all partial results of owned rows together.
5: Receive and sum up all partial results for $\mathbf{M}(i,:)$.
6: **else if** $comm =$ "*expand*" **then**
7: **for all** $i \in I_p$ **do** ▶ Send owned rows to all processes in need.
8: Send $\mathbf{M}(i,:)$ to the all processes p' with $i \in F_{p'}$.
9: **for all** $i \in F_p \setminus I_p$ **do** ▶ Receive rows that are needed locally.
10: Receive $\mathbf{M}(i,:)$ from the process $\sigma(i)$.

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

The expand and the fold communications at Lines 11 and 15 constitute the most expensive communication steps. We outline these communications in Algorithm 8. In the expand communication, the process p sends the partial results for the set $F_p \setminus I_p$ of rows to their owner processes, while similarly receiving all partial results for its set I_p of owned rows and summing them up. Symmetrically, in the fold communication, the process p sends the rows with indices I_p, and receives the rows with indices $F_p \setminus I_p$. The exact set of row indices that needs to be communicated in fold and expand steps depends on the partitioning of $\boldsymbol{\mathcal{X}}$ and the factor matrices. As this partition does not change once determined, the communicated rows between p and q stays the same in CP-ALS iterations. Therefore, in our implementation we determine this row set once outside the main CP-ALS iteration, and reuse it at each iteration.

Medium-grain parallelism

For an N-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and using $P = \prod_{i=1}^{N} P_i$ processes, the medium-grain decomposition imposes a partition with $P_1 \times \cdots \times P_N$ Cartesian topology on the dimensions of $\boldsymbol{\mathcal{X}}$. Specifically, for each dimension n, the index set $\mathbb{N}_{I_n}$ is partitioned into P_n sets $\mathcal{I}_1^{(n)}, \ldots, \mathcal{I}_{P_n}^{(n)}$. With this partition, the process with the index $(p_1, \ldots, p_N) \in P_1 \times \cdots \times P_N$ gets $\boldsymbol{\mathcal{X}}(\mathcal{I}_{p_1}^{(1)}, \ldots, \mathcal{I}_{p_N}^{(n)})$ as its local tensor. Each factor matrix $\mathbf{U}^{(n)}$ is also partitioned following this topology where the set of rows $\mathbf{U}^{(n)}(\mathcal{I}_j^{(n)}, :)$ is owned by the processes with index $(p_1, \ldots, p_N)$ where $j \in \mathbb{N}_{P_N}$ and $p_n = j$, even though these rows are to be further partitioned among the processes having $p_n = j$. As a result, one advantage of the medium-grain partition is that only the processes with $p_n = j$ need to communicate with each other in mode n. This does not necessarily reduce the volume of communication, but it can reduce the number of communicated messages by a factor of P_n in the nth dimension.

One can design an algorithm specifically for the medium-grain decomposition [96]. However, using the fine-grain algorithm on a medium-grain partition effectively provides a medium-grain algorithm. For this reason, we do not need nor provide a separate algorithm for the medium-grain task model, and use the fine-grain algorithm with a proper medium-grain partition instead, which equally benefits from the topology.

Partitioning

The distributed memory algorithms that we described require partitioning the data and the computations, as in any distributed memory algorithm. In order to reason about their computational load balance and communication cost, we use hypergraph models. Once the models are built, different hypergraph partitioning methods can be used to partition the data and the computations. We discuss a few partitioning alternatives.

Partitioning for the fine-grain parallelism

We propose a hypergraph model to capture the computational load and the communication volume of the fine-grain parallelization given in Algorithm 7. For the simplicity of the discussion, we present the model for a 3rd order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and factor matrices $\mathbf{U}^{(1)} \in \mathbb{R}^{I_1 \times R}$, $\mathbf{U}^{(2)} \in \mathbb{R}^{I_2 \times R}$, and $\mathbf{U}^{(3)} \in \mathbb{R}^{I_3 \times R}$. For these inputs, we construct a hypergraph $H = (V, E)$ with the vertex set V and the hyperedge set E. The generalization of the model to higher order tensors should be clear from this construction.

The vertex set $V = V^{(1)} \cup V^{(2)} \cup V^{(3)} \cup V^{(\boldsymbol{\mathcal{X}})}$ of the hypergraph involves four types of vertices. The first three types correspond to the rows of the matrices $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$, and $\mathbf{U}^{(3)}$. In particular, we have vertices $v_i^{(1)} \in V^{(1)}$ for $i \in \mathbb{N}_{I_1}$, $v_j^{(2)} \in V^{(2)}$ for $j \in \mathbb{N}_{I_2}$, and $v_k^{(3)} \in V^{(3)}$ for $k \in \mathbb{N}_{I_3}$. These vertices represent the "ownership" of the corresponding matrix rows, and we assign unit weight to each such vertex. The fourth type of vertices are denoted by $v_{i,j,k}^{(\boldsymbol{\mathcal{X}})}$, which we define for each nonzero $x_{i,j,k} \in \boldsymbol{\mathcal{X}}$. This vertex type relates to the number of operations performed in TTMV due to the nonzero element $x_{i,j,k} \in \boldsymbol{\mathcal{X}}$ in using (3.4) in all modes. In the N-dimensional case, this includes up to N vector Hadamard products involving the value of the nonzero $x_{i,j,k}$, and the corresponding matrix rows. The exact number of performed Hadamard products depends on how nonzero indices coincide as TTVs are carried out, and cannot be determined before a partitioning takes place. In our earlier work [50],

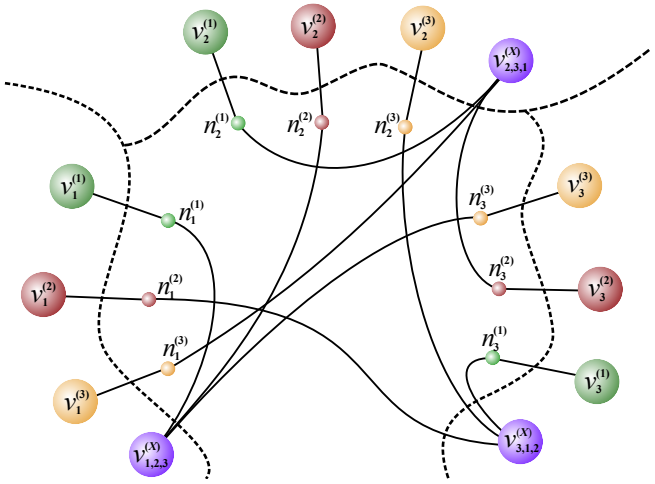


Figure 3.2: Fine-grain hypergraph model for the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$ and a 3-way partition of the hypergraph. The objective is to minimize the cutsize of the partition while maintaining a balance on the total part weights corresponding to each vertex type (shown with different colors).

this cost was exactly N Hadamard products per nonzero, as the MTTKRPs were computed without reusing partial results and without index compression after each TTMV. In the current case, we assign a cost of N to each vertex $v_{i,j,k}^{(\mathcal{X})}$ to represent an upper bound on the computational cost, and expect this to lead to a good load balance in practice. With these vertex definitions, one can use multi-constraint partitioning (2.7) with one constraint per vertex type. In this case, the first, the second, and the third types have unit weights in the first, second, and third constraints, respectively, and zero weight in all other constraints. The fourth vertex type also gets a unit weight (N, or equivalently, 1) in the fourth constraint, and zero weight for others. Here, balancing the first three constraints corresponds to balancing the number of matrix rows at each process (which provides the memory balance as well as the computational balance in dense matrix operations), whereas balancing the fourth type corresponds to balancing the computational load due to TTMVs.

As TTMVs are carried out using (3.4), data dependencies to the rows of $\mathbf{U}^{(1)}(i,:)$, $\mathbf{U}^{(2)}(j,:)$, and $\mathbf{U}^{(3)}(k,:)$ take place when performing Hadamard products due to each nonzero $x_{i,j,k}$. We introduce three types of hyperedges in $E = E^{(1)} \cup E^{(2)} \cup E^{(3)}$ to represent these dependencies as follows: $E^{(1)}$ contains a hyperedge $n_i^{(1)}$ for each matrix row $\mathbf{U}^{(1)}(i,:)$, $E^{(2)}$ contains a hyperedge $n_j^{(2)}$ for each row $\mathbf{U}^{(2)}(j,:)$, and $E^{(3)}$ contains a hyperedge $n_k^{(3)}$ for each row $\mathbf{U}^{(3)}(k,:)$. Initially, $n_i^{(1)}$, $n_j^{(2)}$ and $n_k^{(3)}$ contain the corresponding vertices $v_i^{(1)}$, $v_j^{(2)}$, and $v_k^{(3)}$, as the owner of a matrix row has a dependency to it by default. In computing the MTTKRP using (3.4), each nonzero $x_{i,j,k}$ requires access to $\mathbf{U}^{(1)}(i,:)$, $\mathbf{U}^{(2)}(j,:)$, and $\mathbf{U}^{(3)}(k,:)$. Therefore, we add the vertex $v_{i,j,k}^{(\mathcal{X})}$ to the hyperedges $n_i^{(1)}$, $n_j^{(2)}$ and $n_k^{(3)}$ to model this dependency. In Figure 3.2, we demonstrate this fine-grain hypergraph model on a sample tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$, yet we exclude the vertex weights for simplicity. Each vertex type and hyperedge type is shown using a different color in the figure.

Consider now a P-way partition of the vertices of $H = (V,E)$ where each part is associated with a unique process to obtain a P-way parallel execution of Algorithm 7. We consider the first subiteration of Algorithm 7 that updates $\mathbf{U}^{(1)}$, and assume that each process already has all data

elements to carry out the local TTMVs at Line 9. Now suppose that the nonzero $x_{i,j,k}$ is owned by the process p and the matrix row $\mathbf{U}^{(1)}(i,:)$ is owned by the process q. Then, the process p computes a partial result for $\mathbf{M}^{(1)}(i,:)$ which needs to be sent to the process q at Line 3 of Algorithm 8. By construction of the hypergraph, we have $v_i^{(1)} \in n_i^{(1)}$ which resides at the process q, and due to the nonzero $x_{i,j,k}$ we have $v_{i,j,k}^{(\boldsymbol{\mathcal{X}})} \in n_i^{(1)}$ which resides at the process p; therefore, this communication is accurately represented in the *connectivity* $\lambda_{n_i^{(1)}}$ of the hyperedge $n_i^{(1)}$. In general, the hyperedge $n_i^{(1)}$ incurs $\lambda_{n_i^{(1)}} - 1$ messages to transfer the partial results for the matrix row $\mathbf{M}^{(1)}(i,:)$ to the process q at Line 3. Therefore, the *connectivity-1 cutsize* metric (2.6) over the hyperedges exactly encodes the total volume of messages sent at Line 3, if we set $\mathsf{c}[\cdot] = R$. Since the send operations at Line 8 are duals of the send operations at Line 3, the total volume of messages sent at Line 8 for the first mode is also equal to this number. By extending this reasoning to all other modes, we obtain that the cumulative (over all modes) volume of communication in one iteration of Algorithm 7 equals to the connectivity-1 cut-size metric. As the communication due to each mode take place in different stages, one might alternatively use a multi-objective hypergraph model to minimize the communication volume due to each mode (or equivalently, hyperedge type) independently.

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

We now discuss three methods for partitioning the described hypergraph.

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

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

Label propagation-like: Given that the standard partitioning approach is too costly in practice, we developed a fast hypergraph partitioning heuristic which has reasonable memory and computational costs. The method is based on the balanced label-propagation algorithm [93, 104], and includes some additional adaptations to handle hypergraphs [14, 37]. The heuristic starts with an initial assignment of vertices to parts, and then proceeds with multiple passes over the hypergraph. At each pass, the vertices are visited in an order, and are possibly moved to other parts in order to reduce the cutsize while respecting the balance constraints.

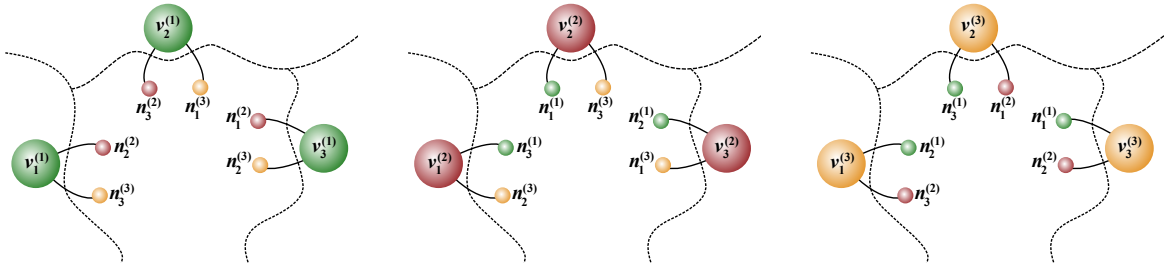
For the heuristic to be efficient on hypergraphs, some adaptations are needed. Each pass involves

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

Partitioning for the medium-grain parallelism

We propose a hypergraph model to capture the communication requirements of the medium-grain algorithm. It is an adaptation of the checkerboard partitioning designed for matrices [18,19] to tensors. The partitioning heuristic for the medium-grain parallelism proceeds in N steps for an N-dimensional tensor, and for simplicity, we discuss the three dimensional case in which we partition the tensor for a $P_1 \times P_2 \times P_3$ process topology where $P = P_1P_2P_3$. Each step partitions the tensor slices in the corresponding mode. In the first step, a hypergraph $H_1 = (V^{(1)}, E^{(1)})$ is built containing a vertex $v_i^{(1)}$ for each tensor slice $\boldsymbol{\mathcal{X}}(i, :, :)$ in the first mode, and hyperedges $n_j^{(2)}$ and $n_k^{(3)}$ for each index $j \in \mathbb{N}_{I_2}$ and $k \in \mathbb{N}_{I_3}$ in the second and the third modes. Next, each vertex $v_i^{(1)}$ is included in the hyperedge $n_j^{(2)}$ if $\boldsymbol{\mathcal{X}}(i, j, :)$ contains a nonzero; similarly, $v_i^{(1)}$ is included in the hyperedge $n_k^{(3)}$ if $\boldsymbol{\mathcal{X}}(i, :, k)$ is not empty. The resulting hypergraph is shown in Figure 3.3a for the previous sample tensor. The vertices have weights corresponding to the number of nonzeros in the associated slices. This hypergraph is then partitioned into P_1 parts. A similar hypergraph is built in the second step for the second mode, as shown in Figure 3.3b. In this hypergraph, each vertex $v_j^{(2)}$ has P_1 weights designating the number of nonzeros of the jth slice of the second mode of $\boldsymbol{\mathcal{X}}$ that are contained in the parts identified in the first step. After having the second hypergraph partitioned into P_2 parts, in the third step, the hypergraph associated with the third mode is built similarly, as shown in Figure 3.3c, and partitioned into P_3 parts. Here, each vertex $v_k^{(3)}$ has P_1P_2 weights representing the number of nonzeros of the kth slice of the third mode of $\boldsymbol{\mathcal{X}}$ that are contained in the Cartesian product of the parts identified in the first two steps. The resulting partitions are used to assign tensor nonzeros to the P processes. The underlying $P_1 \times P_2 \times P_3$ topology confines the communication in each mode to a 2D grid of processors, e.g., in the first mode, each processor communicates with P_2P_3 processors. This bounds the maximum number of messages sent per process by P_2P_3, but does not does not necessarily decrease the actual volume of communication.

Nevertheless, as P can be large, the number of constraints P_1 and P_1P_2 can similarly get large, in which case the state of the art partitioners are known to not perform well both in terms of partition quality and speed. For higher dimensional tensors, this situation only gets worse. That is why explicit communication reduction using hypergraph partitioning for the medium-grain algorithm is not feasible in practice. Hence, we use the partitioning heuristic by Smith and Karypis [96] to partition medium-grain hypergraphs for load balance, and to expect a communication reduction due to partition topology indirectly. We also determine the partition topology by choosing P_1, P_2, and P_3 proportional to the tensor dimensions I_1, I_2, and I_3.



(a) (b) (c)

Figure 3.3: Medium-grain hypergraph models for partitioning the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$ in all three dimensions. Vertex weights are not shown. In (a), we have a single constraint partitioning, whereas in (b) and (c), we employ multi-constraint partitioning with P_1 and $P_1 P_2$ constraints, respectively.

Mode partitioning

Once the nonzero partitioning is obtained for the given fine- or medium-grain parallelism, we proceed with partitioning the mode indices (or, equivalently, the rows of the factor matrices) using a similar heuristic common in similar work [50, 96]. For each matrix row i in dimension n, we identify the processes that have a data dependency to that row. These are exactly the processes which have at least one nonzero with index i in the nth dimension. Next, all row indices are sorted in increasing order of the number of dependent processes. Finally, each row is greedily assigned to the process having the minimum total communication volume among all processes dependent to that row.

3.3 Related work

There has been many recent advances in the efficient computations of tensor factorizations in general, and CP decomposition in particular. We briefly mention these here and refer the reader to the original sources for details. In [6], Bader and Kolda show how to efficiently carry out MTTKRP as well as other fundamental tensor operations on sparse tensors in MATLAB. GigaTensor [40] is a parallel implementation of CP-ALS using the Map-Reduce framework. DFacTo [22] is a C++ implementation with distributed memory parallelism using MPI, and it uses a particular formulation of MTTKRP using sparse matrix-vector multiplication. SPLATT [96] is an efficient parallelization of MTTKRP and CP-ALS both in shared [98] and distributed memory environments [96] using OpenMP and MPI, and is implemented in C. It uses a medium-grain distributed parallelism with a Cartesian partitioning of the tensor, and generalizes this technique to the tensor completion problem [97]. It is the fastest publicly available CP-ALS implementation in the existing literature, and their approach translates to performing $N(N-1)$ TTMVs in performing the MTTKRP in the main CP-ALS iteration. Karlsson et al. similarly discuss a parallel computation of the tensor completion problem using CP formulation [43] in which they replicate the entire factor matrix $\mathbf{U}^{(n)}$ among MPI processes unlike our approach, and report scalability results only up to 100 cores. For computing the CP decomposition of dense tensors, Phan et al. [82] propose a scheme that divides the tensor modes into two sets, precomputes the TTMVs for each mode set, and finally reuses these partial results to obtain the final MTTKRP result in each mode. This provides a factor of 2 improvement in the number of TTMVs over the traditional approach, and our dimension tree-based framework can be considered as the generalization of this ap-

proach that provides a factor of $N/\log N$ improvement. Finally, Chi and Kolda present an alternative Alternating Poisson Regression (CP-APR) algorithm for computing the CP decomposition of large scale sparse datasets in [21].

While this work was under evaluation, another paper appeared [69]. Li et al. use the same idea of storing intermediate tensors but use a different formulation based on tensor times tensor multiplication and a tensor times matrix through Hadamard products for shared memory systems. The overall approach is similar to that by Phan et al. [82], where the difference lies in the application of the method to sparse tensors and auto-tuning to better control memory use and gains in the operation counts.

Aside from CP decomposition, Baskaran et al. [9] provide a shared memory parallel implementation for the Tucker decomposition of sparse tensors. Kaya et al. [51] provide efficient shared and distributed memory parallelization of the Tucker decomposition for sparse tensors using OpenMP and MPI. Austin et al. [5] discuss a high performance distributed memory parallelization of dense Tucker factorization in the context of data compression. Finally, Perros et al. [81] investigate an efficient computation of hierarchical Tucker decomposition for sparse tensors.

3.4 Experiments

We first investigate the way CP-ALS implementations compare using a single thread to assess the algorithmic impact of using a BBDT in the same implementation. Then, we compare these implementations using multiple threads to evaluate their shared memory parallel performance. Finally, we investigate the medium- and the fine-grain distributed memory parallel algorithms.

3.4.1 Dataset and environment

We experimented with five real-world tensors, namely Netflix, NELL, Flickr, Delicious, and Amazon, which are described more in detail in Section 2.6 and whose sizes are given in Table 2.1. We also used synthetic tensors having 4, 8, 16, and 32 dimensions, which are described in the same section, to measure the effect of increasing dimensionality on performance.

We provide the dimension-tree based CP-ALS implementation in our tensor factorization library called HYPERTENSOR. It is a C++11 implementation providing shared and distributed memory parallelism through OpenMP and MPI libraries. We compared our code against SPLATT v1.1.1 [96], a C code with OpenMP and MPI parallelizations.

For shared memory experiments, we used **Grunch** environment described in Section 2.7 using 1 and 14 threads (single socket). All codes were compiled with gcc/g++-5.3.0 using OpenMP directives and compiler options -O3, -ffast-math, -funroll-loops, -ftree-vectorize, -fstrict-aliasing on this shared memory system. Distributed memory experiments were done on **Blue Gene/Q** using up to 256 nodes (4096 cores). Each core of **Blue Gene/Q** can handle an arithmetic and a memory operation simultaneously; therefore, we assigned 32 threads per node (2 threads per core) for better performance. On this system, all codes were compiled using Clang C++ compiler (version 3.5.2) with IBM MPI wrapper using the same optimization flags, and linked against IBM ESSL library for LAPACK and BLAS routines.

3.4.2 Shared memory experiments

We compare the shared memory performance of the dimension tree-based CP-ALS algorithm with the state of the art. We experimented with four methods called **ht-tree2**, **ht-tree3**, **ht-tree**, and **splatt**. The **ht-tree** method, implemented in HYPERTENSOR, uses a full BBDT to carry out TTMVs. The **ht-tree2** method is the same implementation as **ht-tree** except that it uses a 2-level flat dimension tree. In this tree, N leaf nodes are directly connected to the root, hence no intermediate results are generated. However, TTMVs are performed one mode at a time to benefit from the index compression to reduce the operation count. As a result, this method performs $N-1$ TTMVs for each mode in an iteration just as SPLATT; we thus expect comparable performance. **ht-tree3** uses a three level BBDT, whose second level has two nodes holding the partial TTMV results corresponding to the first and the second half of the set of dimensions. This is analogous to the approach by [82] for computing dense CP decompositions, but it also employs our data structure and shared memory parallelization for sparse tensors. Note that for 3- and 4-dimensional tensors, **ht-tree3** and **ht-tree** use identical trees, thus give the same results. These results for **ht-tree3** are indicated with an asterisk in the tables. Finally, **splatt** corresponds to the parallel CP-ALS implementation in SPLATT. We ran all algorithms for 20 iterations with the rank of approximation $R = 20$ (except for the sequential execution of 16- and 32-dimensional random tensors, which are run for 2 iterations due to their cost), and recorded the average time spent per CP-ALS iteration. Test instances in which a method gets out of memory are indicated with a dash symbol.

Sequential execution

In Table 3.1, we give the sequential per-iteration run time of all three methods. We report the run time in seconds for **splatt**, and the relative speedup with respect to **splatt** for the other three methods. We first note that **ht-tree2** runs slightly slower than **splatt** on three dimensional Amazon and NELL tensors (0.99x and 0.87x), and notably slower on Netflix tensor (0.60x). This is so because SPLATT has a specially tuned implementation for 3-dimensional tensors, whereas we use a single code for all dimensions. On all higher dimensional tensors, **ht-tree2** performs significantly better than **splatt**, up to 2.08x on Random8D, which shows the efficiency of our implementation for N-dimensional tensors even before using a BBDT. The gap between **splatt** and **ht-tree2** narrows using Random16D, as **ht-tree2** depletes the memory in one NUMA node, which is discussed more in Section 3.4.2, and starts accessing the distant memory in the other NUMA node. **ht-tree2** gets out of memory using Random32D, as it stores $O(N^2)$ index arrays.

We now measure the effect of dimension trees by comparing **ht-tree** with and **ht-tree2** in Table 3.1. These two methods use the same TTMV implementation, whereas **ht-tree** uses a full BBDT. On Delicious, Flickr, Netflix, and NELL, **ht-tree** obtains 1.78x, 1.61x, 1.63x, and 1.47x speedup over **ht-tree2** thanks to the BBDT. Likewise, on random tensors, we observed 1.43x, 1.91x, 3.01x speedup on tensors Random4D, Random8D, and Random16D, respectively, using **ht-tree**. This validates our performance expectation (Theorem 1) that as the dimensionality of the tensor increases, a BBDT results in significantly fewer TTMVs and better performance.

Comparing **ht-tree** with **splatt** similarly yields a speedup of 1.99x, 1.97x, 1.26x, 2.05x, and 3.95x on tensors Delicious, Flickr, NELL, Random4D, Random8D, and Random16D, respectively, which similarly meets our expectation of performance gain from Theorem 1. On Amazon, **ht-tree** was only 2% faster, whereas on Netflix, **splatt** was only 2% faster, which was the only instance in which **splatt**

Table 3.1: Sequential CP-ALS run time per iteration. Timings are in seconds for **splatt**, whereas we report the relative speedup with respect to **splatt** for other methods.

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

Table 3.2: Shared memory parallel CP-ALS run time per iteration (in seconds). Timings are in seconds for **splatt**, whereas we report the relative speedup with respect to **splatt** for other methods.

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

had a slight edge over **ht-tree**.

Finally, we note that the performance gap between **ht-tree** and **ht-tree3** widens significantly as the tensor gets higher dimensional. Using Random8D, Random16D, and Random32D, **ht-tree** is 1.47x, 1.95x, and 2.50x faster than **ht-tree3** as it incurs significantly fewer TTMVs. These results suggest that using a full BBDT is indeed the ideal method of choice.

Shared memory parallel execution

In Table 3.2, we give the run time results of all methods with shared memory parallelism using 14 threads. We first note that in HYPERTENSOR, using dimension trees consistently yields better execution times. Using **ht-tree**, we obtain 2.15x, 1.66x, 1.58x, 1.32x, and 1.10x speedup over **ht-tree2** on Delicious, Flickr, Netflix, NELL, and Amazon tensors, respectively. For random tensors, we get 1.62x, 2.69x, and 4.23x speedup on Random4D, Random8D, and Random16D. Comparing **ht-tree** with **splatt**, we observe a speedup of 2.00x, 1.81x, 1.46x, 1.47x, 2.18x, and 3.47x on tensors Delicious, Flickr, NELL, Random4D, Random8D, and Random16D, respectively. This demonstrates that the use of a BBDT in CP-ALS computations can be effectively parallelized in a shared memory

Table 3.3: Symbolic precomputation timings. We report the exact timing for **splatt** in seconds, and relative timing with respect to **splatt** for other methods, indicating the ratio at which **splatt** is faster than these methods this precomputation.

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

setting, on top of significantly reducing the amount of TTMV work. On three dimensional Amazon and Netflix tensors, **splatt** has a slight edge over **ht-tree**, which runs 5% and 14% slower, respectively. Another point to note is that **splatt** has somewhat better parallel speedup in general (over its own sequential run time) than **ht-tree2** and **ht-tree**. This is mostly due to the fact that TTMV is a memory-bound computation; hence, once the memory bandwidth is fully utilized, one cannot expect further speedup through multi-threading. When performing TTMVs, our implementation makes slightly more memory accesses due to extra pointer arrays involved in the dimension tree nodes, which saturates the bandwidth earlier, and in turn somewhat affects the parallel speedup. Nevertheless, using **ht-tree** we achieve up to 5.65x faster runs over **splatt** in a shared memory parallel execution.

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

Preprocessing cost

In Table 3.3, we provide symbolic TTV costs of our methods as well as the precomputation cost of **splatt** for setting up its data structures. All runtimes are for a sequential execution; neither SPLATT nor HYPERTENSOR parallelizes this step in their current version. We first note that **ht-tree** incurs significantly less cost than **ht-tree2** in all instances. This is expected, as **ht-tree2** has $O(N^2)$ index arrays to be sorted, whereas **ht-tree** has only $O(N \log N)$ of them. For the same reason, **ht-tree** gets notably faster than **ht-tree3** as the tensor dimensionality increases to 32. The cost is comparable between **splatt** and **ht-tree** for Delicious, Flickr, Netflix, NELL, and Amazon tensors, but **splatt** takes significantly less time as the dimensionality increases, up to 3.97x on Random16D, as it sorts only $O(N)$ arrays.

Comparing these timings with the iteration times in Table 3.1, we see that this precomputation is amortized in a few iterations, except for Netflix and NELL. In practice, CP-ALS is typically executed multiple times with different initial matrices and ranks of approximation using the same symbolic dimension tree construct, which should render this preprocessing cost less important even for these two tensors.

Table 3.4: Memory usage of different methods (in GBs).

	factors	**splatt**	**ht-tree2**	**ht-tree3**		**ht-tree**	
		index	index	index	buffer	index	buffer
Delicious	3	**7**	17.1	*	*	8.5	5.7
Flickr	4.5	**4.8**	11	*	*	6.5	4.3
Netflix	0.1	**3.4**	6.3	*	*	4.4	1.4
NELL	0.6	**2.5**	4.7	*	*	3.2	0.5
Amazon	1.4	**49.6**	91.2	*	*	65.4	65.4
Random4D	6	**9.1**	20.1	*	*	10.2	10.2
Random8D	11.9	**21**	89.1	22.7	15.3	22.7	30.5
Random16D	23.8	**44.9**	373.5	66.1	15.3	55.4	45.7
Random32D	47.7	**94.3**	OOM	226	15.3	123.8	61

Memory usage

We provide the memory consumption of all methods in Table 3.4. The first column corresponds to the amount of memory used to store factor matrices, which is common to all methods. We provide the memory usage for storing index arrays for all methods, and also give the buffer memory consumption for storing the value arrays of intermediate tensors for **ht-tree3** and **ht-tree**. We first note that **ht-tree2** uses the highest amount of memory to store index arrays as expected, up to 8.31 times more than **splatt** using Random16D. In all tensors, the amount of index memory used by **ht-tree** is only slightly higher than **splatt**, the worst cases being Flickr and Random32D tensors for which **ht-tree** consumes 1.35 and 1.31 times more memory in comparison. Even though **splatt** uses only $O(N)$ index arrays, we realized upon inspecting the implementation that it uses two different representations of a tensor for faster execution, effectively doubling the memory requirements. Aside from this, we note that the amount of buffer memory used to store the values of intermediate tensors is reasonable, Random8D being the only exception in which the buffer size exceeds the index size. Finally, **ht-tree3** uses more memory for index storage than **ht-tree** for high dimensional tensors, as it uses more index arrays. It uses less buffer, however, as it has only one level that stores intermediate tensors. All in all, we conclude upon considering these results that **ht-tree** provides remarkable performance improvements with a reasonable increase in the memory usage.

3.4.3 Distributed memory experiments

We compare the performance and the scalability of the fine- and the medium-grain parallel CP-ALS algorithms. In these experiments, we do not use SPLATT software to benchmark medium-grain parallelization for two reasons. First, we would like to compare the effect of load balance and communication cost in different algorithms using different partitionings, while isolating the effects of the efficiency of local CP-ALS computations. Since SPLATT's medium-grain implementation does not use BBDTs for local TTMVs, and is notably slower, comparing it against HYPERTENSOR's fine-grain implementation which has faster local TTMVs would not be fair, nor would correctly reveal the effect of different partitioning strategies. Second, we were not able to get SPLATT to work on our distributed system despite our full efforts. Therefore, we instead performed medium-grain partitioning of tensors following the description of SPLATT's heuristic [96], and ran HYPERTENSOR on

Table 3.5: Time spent per iteration (in seconds) for our distributed memory parallel CP-ALS using two threads per core with different partitions. R and C correspond to number of nodes (MPI ranks) and cores used in each instance, respectively.

R×C	Delicious				Flickr			
	med-gd	fine-rd	fine-lb	fine-hp	med-gd	fine-rd	fine-lb	fine-hp
8 × 1	45.078	69.350	46.741	45.256	29.870	-	25.674	26.021
8 × 16	3.192	9.717	3.603	2.487	2.394	-	2.011	1.512
16 × 16	2.589	7.156	2.297	1.454	1.603	9.476	1.091	0.862
32 × 16	1.612	4.995	1.458	0.873	1.120	6.365	0.694	0.479
64 × 16	1.315	2.907	0.899	0.539	1.013	3.627	0.353	0.286
128 × 16	0.822	1.746	0.533	0.351	0.652	1.911	0.222	0.179
256 × 16	0.603	1.115	0.318	**0.246**	0.554	1.093	0.173	**0.144**

R×C	Netflix				NELL			
	med-gd	fine-rd	fine-lb	fine-hp	med-gd	fine-rd	fine-lb	fine-hp
4 × 1	27.656	32.233	28.118	29.336	19.540	22.528	20.215	19.967
4 × 16	1.095	1.685	1.191	1.185	1.191	1.927	1.205	1.109
8 × 16	0.617	1.247	0.697	0.688	0.749	1.455	0.700	0.681
16 × 16	0.360	0.988	0.421	0.420	0.448	0.998	0.433	0.444
32 × 16	0.222	0.804	0.267	0.261	0.287	0.733	0.282	0.318
64 × 16	0.138	0.660	0.178	0.173	0.179	0.537	0.201	0.239
128 × 16	0.097	0.525	0.117	0.102	0.127	0.398	0.153	0.155
256 × 16	**0.086**	0.410	0.117	0.106	**0.099**	0.316	0.124	0.119

R×C	Amazon		
	med-gd	fine-rd	fine-lb
64 × 1	-	65.545	36.303
64 × 16	-	8.803	1.874
128 × 16	1.141	6.192	1.007
256 × 16	1.056	4.229	**0.570**

these partitions which incurs the same cost in terms of the communication volume and the number of messages as SPLATT, while using more efficient TTMV kernels. For local CP-ALS computations, we use the BBDT-based method **ht-tree** for shared memory parallelism, as it gives the best performance. This way, the experiments become more precise in terms of measuring the influence of medium- and fine-grain algorithms and associated partitionings on parallel scalability.

We investigate the performance in two tables. In Table 3.5, we give the run time results of the medium- and the fine-grain algorithms up to 256 MPI ranks using 4096 cores. Since we achieved the maximum scalability in most tensors with 256 MPI ranks and 4096 cores, the discussion is mostly confined to this case. Especially, in Table 3.6, we give the detailed load balance and communication cost metrics just for this case.

In Table 3.5, we compare the execution of Algorithm 7 with three partitioning methods. The **fine-hp** and **fine-lb** correspond, respectively, to the standard hypergraph partitioning and the label-propagation-like heuristic of Section 3.2.2. For **fine-hp**, we used PaToH [17] with the default settings. On Amazon tensor, we could not obtain results for **fine-hp** as the tensor was too big for PaToH to handle. For **fine-lb**, we ran the three alternatives, each for three passes, and chose the partitioning with the smallest cut. The **med-gd** method corresponds to the medium-grain partitioning heuristic [96]. The **fine-rd** method refers to the random partitioning of the fine-grain hypergraph, which is given as a

reference to illustrate the impact of a good partitioning. Due to memory constraints, we were not able to execute Algorithm 7 on a single node, as the original tensors are large. Therefore, for each tensor, we give the results starting from the minimum number of nodes needed, and for the same instance we also give the single threaded results. In passing from one core per node to 16 core per node, we see some speedup over 16 in Table 3.5; this is because we use two threads per core (see Section 3.4.1).

In order to be able to discuss the speedup results, we give the number of tensor nonzeros per part, computational load (the number of Hadamard products), communication volume, and the number of messages incurred by these three partitionings, using 256 MPI ranks in Table 3.6. For the four performance metrics, we give the maximum and the average value observed across all processes. We see in all instances except **fine-hp** on NELL that balancing the number of nonzeros per part gracefully translates into balancing the actual computational load. For each dataset, we use the run time of the single threaded execution of the fastest method as our baseline in computing the parallel speedup.

As seen in Table 3.5, using 256 MPI ranks on Delicious, **fine-hp** and **fine-lb** yield 2.5x and 1.9x speedup, respectively, over **med-gd**. We observe in Table 3.6 that this is due to better minimization of the total and the maximum communication volume. On Flickr, **fine-hp** is 3.9x faster than **med-gd** at 256 MPI ranks with 14.7x and 10.6x less total and maximum communication volume, while **fine-lb** shows a speedup of 3.2x over **med-gd** with 6.4x and 5.4x less total and maximum communication cost. In both tensors, **med-gd** results in about the half the communication volume of **fine-rd**. In overall, on Delicious **fine-hp** and **fine-lb** obtain 186x and 142x speedup using 4096 cores, whereas **med-gd** and **fine-rd** give 75x and 40x speedup for the same tensor. On Flickr, **fine-hp** and **fine-lb** similarly yield 178x and 148x speedup using 4096 cores, while **med-gd** and **fine-rd** could achieve 46x and 23x speedup. For the Delicious and Flickr tensors, while passing from 8 nodes (with 8×16 cores) to 256 nodes (with 256×16 cores), **med-gd** results in 5.29x and 4.32x speedup. The **fine-hp** and **fine-lb** result in 11.33x and 10.11x speedup for Delicious, and 11.62x and 10.50x speedup for Flickr in the same scenario. **fine-rd** is significantly slower than the other methods, incurring the highest communication as shown in Table 3.6.

On Netflix and NELL, **med-gd** yields 321x and 197x speedup using 4096 cores. **fine-hp** shows a comparable performance with 260x and 164x speedup, whereas **fine-lb** is slightly slower than **fine-hp** with 236x and 157x speedup. We first note that in passing from 4 nodes (with 4×16 cores) to 256 nodes (with 256×16 cores), **med-gd** results in 12.73x and 12.03x speedup for Netflix and NELL, respectively. The **fine-hp** and **fine-lb** partitioning result in 10.18x and 11.18x speedup for Netflix, and 9.72x and 9.32x speedup for NELL in the same scenario. **fine-rd** is similarly the slowest of all methods giving 67x and 62x speedup for these two tensors. Using Netflix and NELL, **med-gd** gets 23% and 20% faster than **fine-hp**, and 36% and 25% faster than **fine-lb**. In Table 3.6, we see that this is due to **med-gd** incurring smaller maximum communication volume and having less number of messages. We investigated this outcome and observed that when a tensor is long in one mode and short in all others, the communication due to the long mode dominates the overall cost, and the communication for the small modes remains negligible in comparison. On Netflix with $P = 256$ MPI ranks, using **med-gd** with $P = p \times q \times r$ topology, the worst case communication volume for the first mode is upper-bounded by $480K(qr - 1)$, as $I_1 = 480K$ indices are distributed to p process "slices" each with qr processes. Similarly, for the second and the third modes, the worst case communication volumes are $17K(pr - 1)$ and $2K(pq - 1)$. In such cases, choosing a large p and smaller q and r significantly reduces the worst-case communication cost in the first mode, while the cost in other modes stays low. The medium-grain heuristic achieves this. Specifically, on Netflix, the medium-grain heuristic chooses a grid size of $64 \times 4 \times 1$. This advantage is lost when there are at least two

Table 3.6: Load balance and communication statistics for 256-way partitioning.

Partitioning	Nnz		Comp. Load		Comm. Vol.		Num. Msg	
	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.
Delicious								
fine-hp	547K	547K	1947K	1807K	**199K**	**137K**	2039	2018
fine-lb	564K	547K	**1849K**	**1737K**	309K	265K	2040	2040
fine-rd	**550K**	547K	2747K	2737K	1083K	1080K	2040	2040
medium-gd	598K	547K	2353K	2214K	624K	571K	**1096**	**1096**
Flickr								
fine-hp	**441K**	441K	**1334K**	**1221K**	**54K**	**38K**	1827	1588
fine-lb	454K	441K	1335K	1257K	107K	87K	2040	2030
fine-rd	443K	441K	2318K	2308K	1042K	1038K	2040	2040
medium-gd	443K	441K	1826K	1806K	576K	558K	**1152**	**1152**
Netflix								
fine-hp	**392K**	392K	1439K	1211K	49K	**19K**	1380	1158
fine-lb	404K	392K	1252K	1208K	48K	39K	1530	1528
fine-rd	394K	392K	1681K	1674K	412K	411K	1530	1530
medium-gd	394K	393K	**1184K**	**1177K**	**26K**	24K	**642**	**642**
NELL								
fine-hp	**307K**	307K	1219K	**787K**	**46K**	**23K**	1513	1402
fine-lb	316K	307K	920K	888K	89K	84K	1522	1502
fine-rd	309K	307K	1211K	1205K	271K	269K	1530	1520
medium-gd	310K	307K	**871K**	855K	58K	51K	**583**	**570**
Amazon								
fine-lb	5104K	4955K	**16871K**	16241K	**211K**	203K	1503	1503
fine-rd	**4962K**	4955K	20964K	20940K	4656K	4651K	1530	1530
medium-gd	19984K	4955K	50023K	16255K	230K	**170K**	**550**	**514**

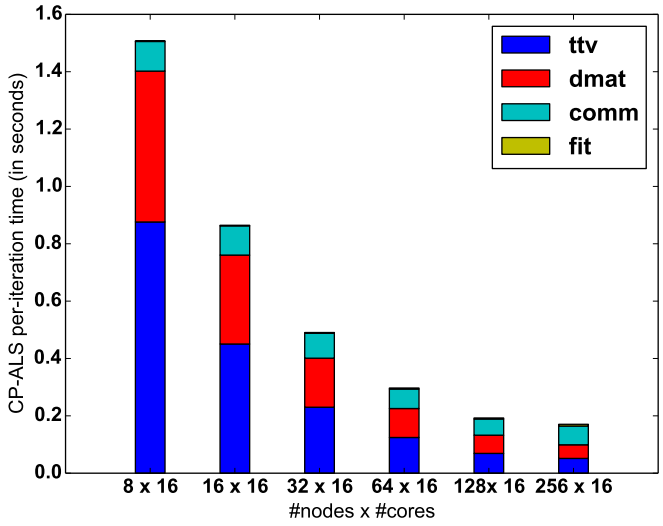


Figure 3.4: Running time dissection of a parallel CP-ALS iteration using **fine-hp** and hp-tree scheme. The legends "ttv", "dmat", "comm", and "fit" correspond to the time spent for TTMVs, dense matrix operations following the TTMVs, communication, and fit computation. The fit computation's time is not discernible in the plot.

long dimensions (see Delicious and Flickr), as using more processes in one long mode can increase the communication significantly in the other long modes.

On Amazon tensor, **med-gd** starts to lose scalability at 256 MPI ranks. In Table 3.6 we observe that this is due to the load imbalance. Amazon tensor has some relatively "dense" slices that make load balancing difficult for the medium-grain heuristic. This problem never arises in the fine-grain partitioning due to finer granularity of tasks; as a result, **fine-lb** runs 1.85x faster than **med-gd** using 256 MPI ranks. In this tensor, from 128 nodes to 256 nodes, **med-gd** displays a speedup of 1.08. With **fine-lb**, the parallel algorithm enjoys 1.86x speedup in passing from 64 nodes to 128 nodes, and 3.2x speedup in passing from 64 to 256 nodes.

In Figure 3.4, we present the dissection of the parallel run time for a CP-ALS iteration on Flickr tensor using 256 MPI ranks. We choose Flickr as representative, as it includes the highest proportion of dense matrix operations in comparison to all other tensors. Despite this fact and using a BBDT for faster TTMVs, the TTMV step still remains to be the dominant computational cost. In this figure, we first observe that the workload due to TTMV and dense matrix computations decrease with the increasing number of processes. Second, we expect in general that having more processes increases the total communication volume; yet we observe in the plot that the communication cost declines until 128 MPI ranks. This is because a good partitioning can reduce the communication volume per process (while increasing the total communication volume). At 256 MPI ranks, however, communication cost starts to increase and become the bottleneck. The fit computation takes a negligible amount of time hence is not discernible in the plot.

On Flickr tensor, the three variants of the **fine-lb** took 58.38, 89.66, and 65.04 seconds to partition the hypergraph, **med-gd** took 190 seconds, and **fine-hp** took 207 minutes. In all data instances, **fine-lb** gives good results while being a fast partitioning heuristic. **fine-hp** consistently provides better

partitions than **fine-lb** in all instances, yet the partitioning cost might render it impractical to use in real-world scenarios. **med-gd** heuristic is only effective when the tensor nonzeros are homogeneously distributed, and the tensor has only one large dimension. One might consider reducing the communication volume on a medium-grain topology using hypergraph partitioning, yet the high number of constraints prevents this approach from being amenable. Therefore, we believe that **fine-lb** serves well in most practical situations.

3.5 Conclusion

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

Chapter 4

Parallel Nonzero CP Decomposition of Sparse Tensors

Similar to the previous chapter, the focus of this chapter is on investigating an efficient parallel computation of CP-ALS for high dimensional big sparse tensors, which is particularly motivated by emerging big data applications [81]. The techniques introduced in the previous chapter provide substantial benefits for high dimensional tensors, reducing the cost of a CP-ALS iteration to $O(N \log N)$ TTMVs from $O(N^2)$. We investigate a new parallel algorithm for performing MTTKRPs, which further improves computational gains for high dimensional tensors, and its effective parallelization in shared memory.

Instead of the TTV-based formulation for MTTKRPs used in the previous chapter, this chapter employs the equivalent nonzero-based formulation for MTTKRPs provided in Algorithm 2, and discusses the cost analysis accordingly. Even though the MTTKRP implementation in DFACTO and SPLATT translates to Algorithm 2, they aim to reduce the cost of this algorithm with the help index overlaps after multiplication in a dimension. To illustrate, for a 3-dimensional tensor $\boldsymbol{\mathcal{X}}$ having nonzeros x_{i,j,k_1} and x_{i,j,k_2}, one can first compute $x_{i,j,k_1}\mathbf{U}^{(3)}(k_1,:) + x_{i,j,k_2}\mathbf{U}^{(3)}(k_2,:)$, then multiply this result with $\mathbf{U}^{(2)}(j,:)$ to obtain the final contribution to $\mathbf{M}^{(1)}(i,:)$. Even though this approach can substantially reduce the number of Hadamard multiplications, the worst-case complexity of this approach stays the same, i.e., $O(\text{nnz}(\boldsymbol{\mathcal{X}})NR)$. Throughout the chapter, we let HYPERTENSOR represent our dimension tree-based algorithm described in the previous chapter, in which MTTKRP is expressed as a series of R tensor-times-vector multiply operations whose worst-case amortized cost translates to $O(\text{nnz}(\boldsymbol{\mathcal{X}}) \log NR)$ for each tensor dimension, which provides significant performance gains as the tensor dimensionality increases in compare to SPLATT. The goal is chapter is to further reduce this cost down to $O(\text{nnz}(\boldsymbol{\mathcal{X}})R)$ while providing effective shared memory parallelizations. This is made possible due to the assumption of nonzero factor matrices in the algorithm.

We summarize the contributions in this chapter as following:

- We introduce a method for efficiently computing MTTKRP for high dimensional sparse tensors. This scheme asymptotically reduces the computational cost of MTTKRP as well as a common precomputation step performed in SPLATT and the dimension tree-based method discussed in Chapter 3.

- We provide an efficient parallelization of this computational scheme that runs up to 10.5 times

faster than SPLATT and 3.28 times faster than the dimension tree-based CP-ALS on a 28-core workstation.

The rest of the chapter is organized as follows. In Section 4.1, we describe our method for computing MTTKRP and CP-ALS, which imposes a nonzero constraint on factor matrices to reduce the computational costs. We compare the complexity of our approach with the state of the art, and discuss a carefully tuned parallelization of this approach for a shared memory NUMA architecture. Finally, we provide in Section 4.2 a comparison of sequential and parallel executions of our method with two state-of-the-art implementations.

The work in this chapter is published in [46].

4.1 Parallel CP decomposition using nonzero factors

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

4.1.1 Computing CP decomposition with nonzero factors

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

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

4.1.2 Parallelization

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

Algorithm 9 Parallel CP-ALS with nonzero factors.

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

```
1:  Initialize Z and W^(n) for all n ∈ ℕ_N.
2:  repeat
3:    for n = 1,...,N do
4:      parallel for p = 1...P do                          ▶ Compute M^(n)(I_p^(n),:)
5:        for i_n ∈ I_p^(n) do
6:          M^(n)(i_n,:) ← 0
7:          for z_{i_1,...,i_N} ∈ rl^(n)(i_n) do
8:            M^(n)(i_n,:) += z_{i_1,...,i_N} / U^(n)(i_n,:)
9:            z_{i_1,...,i_N} = z_{i_1,...,i_N} ⊘ U^(n)(i_n,:)
10:     H^(n) ← *_{i≠n} W^(i)                              ▶ Matrix Hadamard product
11:     U^(n) ← M^(n) H^(n)†                               ▶ Row-parallel GEMM
12:     λ ← NONZERO-COLUMN-NORMALIZE(U^(n))
13:     W^(n) ← U^(i)^T U^(i)                              ▶ Row-parallel SYRK
14:     parallel for p = 1...P do                          ▶ Update Z
15:       for i_n ∈ I_p^(n) do
16:         for z_{i_1,...,i_N} ∈ rl^(n)(i_n) do
17:           z_{i_1,...,i_N} = z_{i_1,...,i_N} * U^(n)(i_n,:)
18: until convergence or reaching maximum number of iterations
19: return [λ; U^(1),...,U^(N)]
```

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

$\mathbf{U}^{(n)}(i,:)$. Once $\mathbf{M}^{(n)}$ is formed, at Line 10 we compute $\mathbf{H}^{(n)}$ by performing the Hadamard product of $R \times R$ matrices $\mathbf{W}^{(k)}$ for $k \neq n$, whose cost is negligible as R is a small constant in practice. Next, at Line 11 we update the factor $\mathbf{U}^{(n)}$ in a parallel dense matrix multiplication step, in which each process p performs the multiplication $\mathbf{M}^{(n)}(\mathcal{I}_p^{(n)},:)\mathbf{H}^{(n)\dagger}$. Once $\mathbf{U}^{(n)}$ is computed, we swap its small entries with ϵ or $-\epsilon$ and normalize its columns in a combined step at Line 12 in which each process p works on the sub-matrix $\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)$. Using the updated $\mathbf{U}^{(n)}$, we first compute the new $\mathbf{W}^{(n)}$ in another parallel dense matrix multiplication step at line Line 13, where the process p similarly performs $\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)^T\mathbf{U}^{(n)}(\mathcal{I}_p^{(n)},:)$, then multiply the entries of $\mathbf{Z}$ with the corresponding rows of $\mathbf{U}^{(n)}$ using the same parallelization scheme as in Line 4. The initialization of matrices $\mathbf{Z}$ as well as $\mathbf{W}^{(n)}$ at Line 1 are done in parallel similar to the manner of updating these matrices in the iteration loop. At the end of each iteration, one has to check the convergence as well. This computation takes insignificant amount of time [52], hence we skip the details.

Reduction lists for a dimension n can be computed by making two passes over the tensor nonzero indices in nth dimension to form a very efficient compressed data structure consisting of $rl^{(n)}$, which correspond to reduction list pointers, and $rli^{(n)}$, which correspond to the elements in the reduction list, in $O(\text{nnz}(\boldsymbol{\mathcal{X}}))$ time. This yields $O(N\text{nnz}(\boldsymbol{\mathcal{X}}))$ cost for all dimensions, and we can process each dimension in parallel. In contrast, existing methods in the literature require sorting the tensor indices which takes $O(N\text{nnz}(\boldsymbol{\mathcal{X}})\log(\text{nnz}(\boldsymbol{\mathcal{X}})))$ time for SPLATT [95], and $O(N\log N\text{nnz}(\boldsymbol{\mathcal{X}})\log(\text{nnz}(\boldsymbol{\mathcal{X}})))$ for HYPERTENSOR [52]. We show this data structure for a small tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{5\times5\times5\times5}$ in Figure 4.1, and skip the computational details.

4.1.3 Load balancing

For a P-way parallel execution of Algorithm 9, one needs to partition the row indices $1,\ldots,I_n$ into P sets $\mathcal{I}_1^{(n)},\ldots,\mathcal{I}_P^{(n)}$ for each dimension n. There are two types of computational costs imposed on each process p by a such partition. First, the process p performs $O(\sum_{i\in\mathcal{I}_p^{(n)}}|rl^{(n)}(i)|)$ vector Hadamard operations at Lines 8, 9 and 17. Second, it performs the multiplication of matrices of size $|\mathcal{I}_p^{(n)}| \times R$ and $R \times R$ at Line 11, and of two matrices of size $|\mathcal{I}_p^{(n)}| \times R$ at Line 13. To balance the first cost pertaining to sparse tensor computations, one has to make sure that the associated cost $\sum_{i\in\mathcal{I}_p^{(n)}}|rl^{(n)}(i)|$ is partitioned equitably to processes. Regarding the second cost for dense matrix operations, each process should have equal number of rows, i.e., $|\mathcal{I}_p^{(n)}|$ should be balanced. Though these rows can be partitioned arbitrarily, in practice we desire to assign a contiguous set of rows to each thread to preserve the data locality. This not only helps improve the memory footprint of the MTTKRP step, but also increases the efficiency of BLAS routines used for dense matrix computations.

We define partitioning problem in this case as follows. For each row i_n, we have an associated pair $(|rl^{(n)}(i_n,:)|,1)$ of costs that corresponds to sparse tensor and dense matrix computations, respectively. We aim to partition this "chain" of rows into P contiguous parts so that both cost metrics are balanced across processes. The single-cost version of this problem corresponds to the chains-on-chains partitioning (CCP) problem in the literature for which many fast optimal algorithms and effective heuristics exist [83]. We employ CCP algorithms by using only the first cost metric $|rl^{(n)}(i,:)|$ for partitioning, as we observe that in practice, balancing this metric also establishes good row-balance.

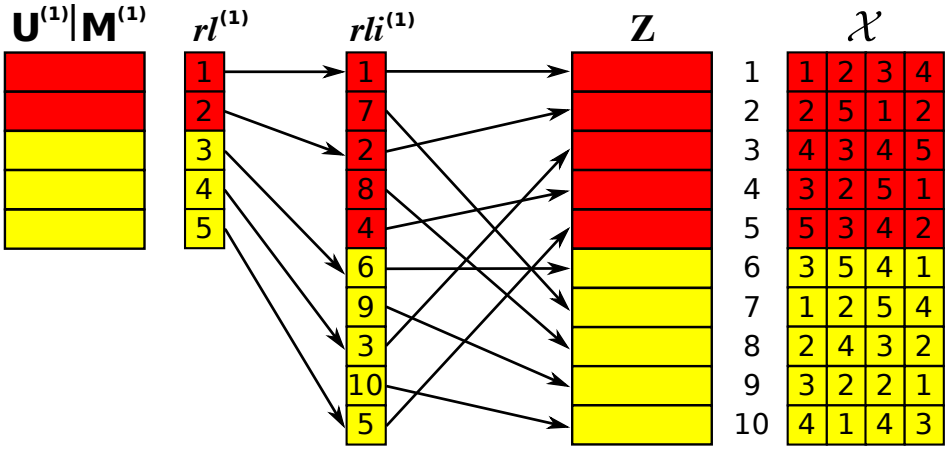


Figure 4.1: Performing MTTKRP for a 4-dimensional tensor $\mathcal{X} \in \mathbb{R}^{5\times5\times5\times5}$ in the first mode. Red (dark) and yellow (light) colors represent memory regions that are first touched by two different threads.

4.1.4 Optimizations for NUMA scalability

Performing MTTKRP for sparse tensors is an extremely memory bound operation as the tensor is very sparse in general, and the data accesses due to tensor nonzeros lack locality. Therefore, optimizing the memory footprint of the implementation plays a crucial role in obtaining high performance. Particularly on a NUMA architecture, one has to carefully allocate memory pages in NUMA nodes to be able to utilize the available memory bandwidth at maximum, and distribute the memory pages equitably across NUMA nodes. In most systems, this can be ensured by properly using memory first-touch policies after allocation, which in turn yields adequate memory page-to-socket bindings. In our implementation, after the allocation each thread performs a first-touch on the matrix rows as well as the rows of $rl^{(1)}$ and $rli^{(1)}$ that it owns. For the matrix $\mathbf{Z}$, each thread initializes a block of $\text{nnz}(\mathcal{X})/P$ vectors. This way, we not only maximize the NUMA bandwidth utilization, but also aim to reduce the inter-NUMA node memory requests as much as possible. This allocation scheme together with a balanced partitioning is shown in Figure 4.1 for two threads.

4.2 Experiments

We compared our algorithm with two state-of-the-art implementations, SPLATT and HYPERTENSOR, and ran them on the **Grunch** computing environment detailed in Section 2.7. We performed two types of experiments to measure the parallel scalability and the accuracy of our method. The first experiment compares the runtime of three methods using synthetically generated high dimensional sparse tensors. The second experiment uses a real-world tensor to compare the quality of approximation of the standard CP-ALS computation with our method. In all experiments, we use $\epsilon = 10^{-6}$ as the threshold parameter.

Table 4.1: Per-iteration CP-ALS runtime results (in seconds) for sequential, single-socket (14 threads) and dual-socket parallel executions of all methods.

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

4.2.1 Scalability

We compare the runtime of three methods using 4, 8, 16, and 32-dimensional randomly (uniform) generated tensors of size 10M at each dimension, and having 100M nonzero elements. We employ synthetic data instead of real-world tensors for two reasons. First, with random data we are able to control the dimensionality of the tensor while fixing other tensor parameters, e.g., dimension sizes, the number of nonzeros, and the distribution of nonzero indices; thereby, observe the performance of the algorithms with the increasing tensor dimensionality in a controlled manner. Second, there is a lack of available big high dimensional sparse tensors in the literature in parallel to the lack of efficient computational tools to handle such tensors. We run all three implementations using 1, 14 (single socket), and 28 cores/threads (two sockets) for 20 CP-ALS iterations using $R = 16$, and report the average time spent per iteration in Table 4.1 with labels **splatt**, **hypertensor**, and **cp-eps** corresponding to SPLATT, HYPERTENSOR, and our algorithm provided in Algorithm 9, respectively.

In Table 4.1, we observe that using ten-4D, **hypertensor** runs the fastest using 1 and 14 cores, yet **cp-eps** surpasses **hypertensor** using 28-cores owing to better NUMA optimizations described in Section 4.1.4. In all other instances, **cp-eps** stays the fastest among all three methods, and the performance gains increase steadily as the tensor dimensionality grows. Using 4-dimensional to 32-dimensional tensors, we observe that the speedup of **cp-eps** over **splatt** consistently increases from 1.39x to 10.47x using 28 threads, and from 1.34x to 16.67x using a single thread, which conforms with the algorithm complexities provided in Section 4.1.1. Similarly, the speedup of **cp-eps** over **hypertensor** varies from 1.56x to 3.29x using 28 threads, and from 0.75x to 2.40x using a single thread. Figure 4.2 demonstrates the speedup results of **cp-eps** and **hypertensor** over **splatt** for ten-32D.

In Table 4.2, we compare the time spent on setting up data structures for MTTKRP using 28 threads on all datasets. We observe that **cp-eps** performs the preprocessing step up to 15x faster than **splatt**, and up to 24x faster than **hypertensor**. This significant improvement is possible owing to the smaller asymptotic complexity described in Section 4.1.2, as well as the parallelization using **cp-eps**.

Table 4.2: Initial setup time (in seconds) for parallel CP-ALS.

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

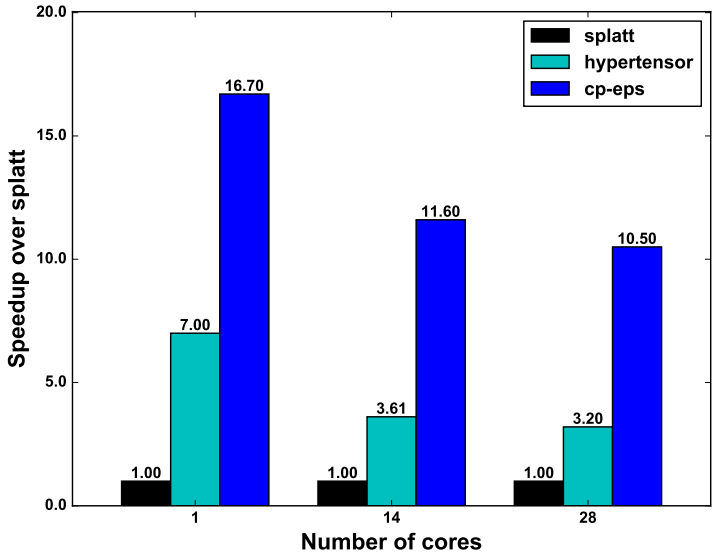


Figure 4.2: Speedup over **splatt** on ten32D.

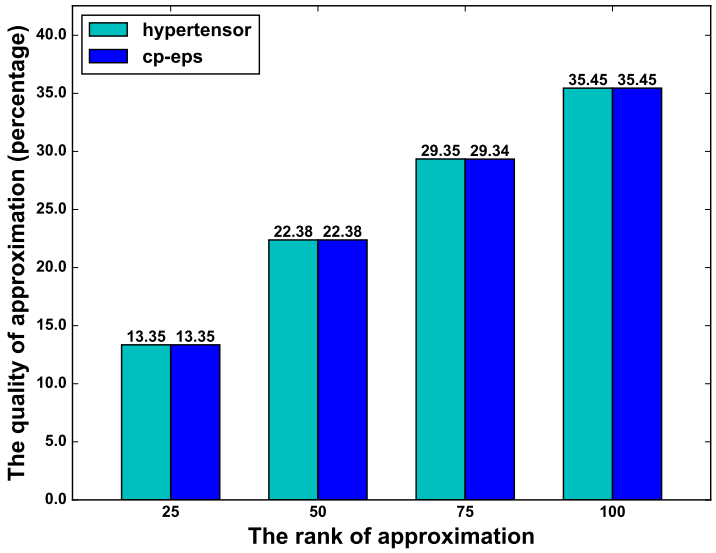


Figure 4.3: Accuracy comparison of **hypertensor** and **cp-eps** on NELL2-2. We show the geometric mean of approximation values of 100 CP-ALS executions with random initial factor matrices using both methods.

4.2.2 Accuracy

In computing Algorithm 9, we impose the constraint on factor matrices that they do not contain very small elements, which perturbs the decomposition slightly and can potentially affect the quality of approximation. To assess this, we compare the accuracy of this method with that of the original CP-ALS algorithm. We employ the 3-dimensional NELL2 tensor described in Section 2.6. We run **hypertensor** and **cp-eps** 100 times (each with a random initialization of factor matrices) with the rank of approximation $R \in \{25, 50, 75, 100\}$, and compute the geometric mean of the approximation quality in each case. In Figure 4.3 we detail all these results. We observe that both methods produce equally good approximations to the original tensor up to a small margin of error for $R = 75$ mostly due to randomization in factor matrix initialization. This shows that the nonzero constraint on factor matrices indeed has a negligible effect on the accuracy.

4.3 Conclusion

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

datasets amenable both in terms of computational and memory requirements for real world applications.

Chapter 5

Parallel Sparse Tucker Decomposition

This chapter concerns an efficient computation as well as shared and distributed memory parallelizations of HOOI algorithm for computing the Tucker decomposition of sparse tensors. Recall that the two main operations in an iteration of HOOI are the TTMc step, which involves the multiplication of the tensor with $N-1$ matrices, and the TRSVD step, which computes the truncated SVD of the matricization of the tensor obtained from the TTMc step. For an efficient algorithm using sparse tensors, both these steps should be performed in a way that effectively exploits the sparsity of the input tensor.

Towards this end, we present the following contributions in this chapter. We design efficient shared and distributed memory parallel algorithms for the TTMc operation on sparse tensors. To this end, we first introduce a particular nonzero-based reformulation of TTMc. Using this formulation, we then introduce a preprocessing step called symbolic TTMc to identify data dependencies and perform all index computations before the HOOI iterations for efficiency. Then, we provide a shared-memory parallel algorithm for the main iteration of HOOI which makes use of the symbolic TTMc step. Next, we introduce a coarse and a fine-grain task definition for TTMc and TRSVD steps within the HOOI algorithm, and propose a hybrid shared-distributed memory parallel algorithm based on the distribution of these tasks. We discuss the computational and the communication requirements of the algorithm for a given task distribution, and make use of the hypergraph models from our earlier work on CP-ALS [50] for reducing communication and achieving load balance during each HOOI iteration. Third, we stress how to efficiently perform the TRSVD step in a distributed memory setting, and make use of the PETSc [8] and SLEPc [87] libraries in this step. 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 step. Finally, we propose an efficient OpenMP-MPI hybrid parallel implementation of the HOOI algorithm in C++, and present scalability results on a high-end parallel system using up to 4096 cores on real world tensors. To the best of our knowledge, this is the first high performance parallel implementation of the HOOI algorithm for sparse tensors in shared/distributed memory environments using OpenMP/MPI.

The organization of this chapter is as follows. We provide a reformulation of the TTMc operation for sparse tensors in the next section. Then, in Section 5.2, we propose shared and distributed memory parallel HOOI algorithms based on this formulation, and discuss in detail the TTMc and the TRSVD steps. Next, we give a brief summary of recent related work in Section 5.3. Finally, we provide experimental results in Section 5.4, and conclude the chapter in Section 5.5. For the sake of simplicity

of the notation and the discussion, we occasionally discuss the case of a 3-dimensional tensor, even though our algorithms and implementations have no such restriction. We adequately generalize the discussion to N-dimensional tensors whenever necessary.

The work in this chapter is published in [51].

5.1 Performing multiple TTMs for sparse tensors

We recall that the TTM 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 $\mathcal{Y} \leftarrow \mathcal{X} \times_n \mathbf{U}$. and 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

$$y_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} x_{i_1,i_2,\ldots,i_N} u_{j i_n} \ . \tag{5.1}$$

A tensor can likewise be multiplied by a set of matrices along a given set S of modes. We use the notation TTMc$(\mathcal{X}, S, \{\mathbf{U}^{(n)} : n \in S\})$ to refer to the tensor n-dimensional product of $\mathcal{X}$ with matrices $\mathbf{U}^{(n)^T}$ for $n \in S$. We use TTMc(S) for clarity, as the tensor $\mathcal{X}$ and the matrices $\mathbf{U}^{(n)^T}$ should be clear from the context. The operation $\mathcal{Y} \leftarrow \mathcal{X} \times_{i \neq n} \mathbf{U}^{(i)^T}$ stands for the TTMc of a tensor in all modes except n, which we also denote as TTMc$(\{1 \ldots N\} \setminus \{n\})$.

The way to perform the TTMc operation at Line 4 of Algorithm 3 is especially important. These TTM's can be performed in any order [58] using various schemes [59] that formulate TTMc in terms of multiple tensor-times-vector (TTV) operations. As we are concerned with sparse tensors, we formulate TTMc in a way that specifies the computational task to be carried out for each nonzero $x_{i_1,\ldots,i_N} \in \mathcal{X}$, which in turn enables expressing parallelism with different task granularities. Let $\mathcal{Z} = \mathcal{X} \times_2 \mathbf{U}_2^T$ for $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and $\mathbf{U}^{(2)} \in \mathbb{R}^{I_2 \times R_2}$. By considering (5.1) for $\mathcal{Z}$, and performing the summation over nonzeros we obtain:

$$z_{i,t,k} = \sum_{x_{i,j,k} \in \mathcal{X}} x_{i,j,k} \mathbf{U}^{(2)}(j,t) \ . \tag{5.2}$$

We can vectorize this as

$$\mathcal{Z}(i,:,k) = \sum_{x_{i,j,k} \in \mathcal{X}} x_{i,j,k} \mathbf{U}^{(2)}(j,:) \ . \tag{5.3}$$

Similarly, $\mathcal{Y} = \mathcal{Z} \times_3 \mathbf{U}_3^T$ for $\mathbf{U}^{(3)} \in \mathbb{R}^{I_3 \times R_3}$ can be written as:

$$y_{i,m,t} \quad = \quad \sum_{z_{i,m,k} \in \mathcal{Z}} z_{i,m,k} \mathbf{U}^{(3)}(k,t) \ .$$

Rewriting $z_{i,m,k}$ as in (5.2) we get

$$y_{i,m,t} \quad = \quad \sum_{x_{i,j,k} \in \mathcal{Z}} x_{i,j,k} \mathbf{U}^{(2)}(j,m) \mathbf{U}^{(3)}(k,t),$$

and finally by applying the vectorization in (5.3) twice we obtain

$$\mathcal{Y}(i,:,:) \quad = \quad \sum_{x_{i,j,k} \in \mathcal{X}} x_{i,j,k} \mathbf{U}^{(2)}(j,:) \circ \mathbf{U}^{(3)}(k,:) \ .$$

Here, for each nonzero $x_{i,j,k} \in \mathcal{X}$, we perform the outer product $\mathbf{U}^{(2)}(j,:) \circ \mathbf{U}^{(3)}(k,:)$, scale it with $x_{i,j,k}$, then add the result to the $R_2 \times R_3$ dense matrix $\mathcal{Y}(i,:,:)$. Using the matricization of $\mathcal{Y}$ in the first mode, this results in the following formula where a Kronecker product replaces the outer product:

$$\mathbf{Y}_{(1)}(i,:) = \sum_{x_{i,j,k} \in \mathcal{X}} x_{i,j,k} \mathbf{U}^{(2)}(j,:) \otimes \mathbf{U}^{(3)}(k,:) \ . \tag{5.4}$$

For N-dimensional case, the formulation (5.4) trivially generalizes to

$$\mathbf{Y}_{(n)}(i_n,:) = \sum_{x_{i_1,\ldots,i_n,\ldots,i_N} \in \mathcal{X}} x_{i_1,i_2,\ldots,i_N} \otimes_{t \neq n} \mathbf{U}^{(t)}(i_t,:). \tag{5.5}$$

This formulation specifies the operations performed for each nonzero element of a tensor in TTMc. The resulting computation of TTMc is called nonzero-based, and is given in Algorithm 10.

Algorithm 10 TTMc operation $\mathcal{Y} = \mathcal{X} \times_{i \neq n} \mathbf{U}^{(i)^T}$

Input: $\mathcal{X}$: An N-dimensional tensor, $\mathcal{X} \in \mathbb{R}^{I_1,\ldots,I_N}$
$\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$: Matrices for TTMc, $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$
n: The mode for performing TTMc
Output: $\mathcal{Y} \leftarrow \mathcal{X} \times_{i \neq n} \mathbf{U}^{(i)^T}$
$\mathbf{Y}_{(n)} \leftarrow 0$
for all $x_{i_1,\ldots,i_N} \in \mathcal{X}$ **do**
$\mathbf{Y}_{(n)}(i_n,:) \leftarrow \mathbf{Y}_{(n)}(i_n,:) + x_{i_1,\ldots,i_N} \otimes_{t \neq n} \mathbf{U}^{(t)}(i_t,:)$

5.2 Parallel HOOI for sparse tensors

An efficient shared memory parallelization of the TTMc operation given in Algorithm 10 should avoid expensive lock mechanisms to resolve data dependencies. We perform a preprocessing step to organize the computations in a way that the subsequent numeric computations can be performed in parallel without any write conflicts. Following the TTMc step, computing the TRSVD of the matricized tensor $\mathbf{Y}_{(n)}$ requires special attention. Direct SVD methods that are employed to compute TRSVD in dense Tucker decomposition algorithms are not feasible for sparse Tucker decomposition due to computational and memory constraints. For this reason, we resort to iterative methods for TRSVD, which not only reduces the computational cost by exploiting the low rank of approximation, but also renders TRSVD computation feasible in terms of memory utilization.

For the distributed memory parallelism, we employ coarse- and fine-grain task definitions. A coarse-grain task corresponds to computing a particular row $\mathbf{Y}_{(n)}(i,:)$ of the TTMc result, as well as the corresponding row $\mathbf{U}^{(n)}(i,:)$ of the factor matrix using TRSVD. In this scenario, the owner of this task possesses all the tensor nonzeros $x_{i_1,\ldots,i_N}$ where $i_n = i$. Also, each such nonzero implies data dependencies to the tasks corresponding to rows $\mathbf{U}^{(1)}(i_1,:), \ldots, \mathbf{U}^{(N)}(i_N,:)$ to be able to perform the computation of $\mathbf{Y}_{(n)}(i,:)$ using Algorithm 10. Fine-grain task definition relaxes this constraint by allowing nonzeros to be distributed freely. It associates each nonzero $x_{i_1,\ldots,i_N}$ with a task which is responsible to compute $x_{i_1,\ldots,i_N} \otimes_{t \neq n} \mathbf{U}^{(t)}(i_t,:)$ and generate a partial result for $\mathbf{Y}_{(n)}(i_n,:)$ of size $\prod_{t \neq n} R_t$. This size is exponential in the ranks of approximation; therefore, merging the partial results can get very expensive in terms of communication, hence should be avoided. For this reason, we propose a method to effectively handle this communication within the TRSVD step.

5.2.1 Shared memory parallelism

Parallel TTMc

As shown in Algorithm 10, each nonzero $x_{i_1,\ldots,i_N}$ contributes an outer product to $\mathbf{Y}_{(n)}(i_n,:)$ while performing TTMc in the first mode. For shared memory parallelism, this poses a write conflict whenever two threads simultaneously process nonzeros whose first index are i. To resolve this, we make a pass over the data to compute an update list $ul_n(i_n)$ that holds the list of nonzeros $x_{i_1,\ldots,i_N}$ that contribute to $\mathcal{Y}_{(n)}(i_n,:)$. In the actual implementation, we only store the index t of the nonzero $x(t) = x_{i_1,\ldots,i_N}$ to avoid duplicating the nonzero within $ul_n(i_n)$. This way, we untangle the write conflicts for each row of $\mathbf{Y}_{(n)}$ and avoid using lock mechanisms. We also store the set $\mathcal{J}_n$ of all indices $i \in I_n$ such that $ul_n(i) \neq \emptyset$. We repeat this computation in all dimensions, and name this step as *symbolic TTMc*, as it resolves all the index computations and dependencies once and for all outside the main loop of HOOI (shown at Lines 1–2 of Algorithm 11). This symbolic data can be reused many times for faster *numeric TTMc*s within the main loop of HOOI. Finally, symbolic TTMc of each dimension can be performed independently; hence, we perform this computation in parallel for each dimension.

After the symbolic TTMc, each row i of $\mathbf{Y}_{(n)}$ can be updated independently in parallel by using $ul_n(i)$, which composes the parallel numeric TTMc step at Lines 5–8 of Algorithm 11. In our implementation, we use OpenMP parallel loop with dynamic scheduling to distribute the tasks to threads.

Algorithm 11 Shared memory parallel HOOI

Input: $\mathcal{X}$: An N-dimensional tensor
$\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}$: Initial factor matrices
$R_1,\ldots,R_N$: Ranks of approximation
Output: $[\![\mathcal{G};\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$: Tucker approximation of $\mathcal{X}$

1: **parfor** $n = 1$ **to** N **do**
2: $\quad\{ul_n, \mathcal{J}_n\} \leftarrow$ SymbolicTTMc$(\mathcal{X}, \{1,\ldots,N\} \setminus \{n\})$
3: **repeat**
4: $\quad$**for** $n = 1$ **to** N **do**
5: $\quad\quad$**parfor** $i \in \mathcal{J}_n$ **do** ▶ TTMc for mode n
6: $\quad\quad\quad\mathbf{Y}_{(n)}(i,:) \leftarrow 0$
7: $\quad\quad\quad$**for all** $x_{i_1,\ldots,i_N} \in ul_n(i)$ **do**
8: $\quad\quad\quad\quad\mathbf{Y}_{(n)}(i,:)$ += $x_{i_1,\ldots,i_N}[\otimes_{t\neq n}\mathbf{U}^{(t)}(i_t,:)]$
9: $\quad\quad\mathbf{U}^{(n)} \leftarrow$ TRSVD$(\mathbf{Y}_{(n)}, R_n)$
10: $\quad\mathcal{G} \leftarrow \mathcal{Y} \times_N \mathbf{U}^{(N)}$
11: **until** convergence or maximum number of iterations
12: **return** $[\![\mathcal{G};\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}]\!]$

Dimension tree-based parallel TTMc

Note that each TTMc call involves the multiplication of $\mathcal{X}$ with the set $\{\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}\} \setminus \mathbf{U}^{(n)}$ of matrices to eventually update $\mathbf{U}^{(n)}$ at the end of the subiteration for mode n. Similar to the techniques used in Chapter 3, this opens the possibility to store and reuse partial TTM results across TTMc calls in different dimensions as follows. For a 4-dimensional tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times I_3\times I_4}$, in the first two ALS subiterations, one needs to compute $\mathcal{X} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$ and $\mathcal{X} \times_1 \mathbf{U}^{(1)} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$

and subsequently update the corresponding matrices $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$, respectively. Note that in doing so, $\mathbf{U}^{(3)}$ and $\mathbf{U}^{(4)}$ remains unchanged, which brings up the possibility of computing the intermediate tensor $\mathcal{Z} = \mathcal{X} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$, then reusing this partial result in the first and second subiterations as $\mathcal{Z} \times_2 \mathbf{U}^{(2)}$ and $\mathcal{Z} \times_1 \mathbf{U}^{(1)}$, respectively. This computation stays valid as one can perform TTM in a set of distinct modes in any order. This way, however, the cost of TTMs reduces significantly.

As we did in Chapter 3, we use BBDTs to minimize the number of TTMs and the number of intermediate tensors held in the dimension tree. Note also that setting the ranks of approximation to 1 equates a TTM to a TTV, in which case the nonzero structure of tree tensors are obtained by the symbolic TTV step described in Section 3.1.2. Increasing the ranks of approximation only increments the size of the value array per tensor element proportional to these ranks according to (5.5), while the sparsity structure in the modes in which TTM is not performed stays still. Therefore, we use the same symbolic TTV step described in Section 3.1.2 to construct the nonzero set of the tree tensors. Once the tree is constructed, we employ Algorithm 12, which is analogous to Algorithm 6, to carry out TTM operations for a node of a dimension tree constructed using a sparse tensor. The only major difference here is the outer product replacing the Hadamard product at Line 9, which is due to (5.5). We can then invoke SMP-DTREE-TTM($\mathcal{L}_n$) to compute TTMc for a dimension n.

Algorithm 12 SMP-DTREE-TTM

Input: t: A dimension tree node/tensor
Output: Numerical values $\mathcal{V}_t$ are computed.

1: **if** EXISTS($\mathcal{V}_t$) **then**
2: **return** ▶ Numerical values $\mathcal{V}_t$ are already computed.
3: SMP-DTREE-TTV($P(t)$) ▶ Compute the parent's values $\mathcal{V}_{P(t)}$ first.
4: **parallel for all** $(i_1, \ldots, i_N) \in \mathcal{I}_t$ **do** ▶ Compute each $\mathcal{V}_t(i_1, \ldots, i_N)$ in parallel.
5: $\mathcal{V}_t(i_1, \ldots, i_N) \leftarrow \text{zeros}(1, R)$ ▶ Initialize with a zero vector of size $1 \times R$.
6: **for all** $(j_1, \ldots, j_N) \in \mathcal{R}(i_1, \ldots, i_N)$ **do** ▶ Perform updates using elements in $\mathcal{R}$.
7: $r \leftarrow \mathcal{V}_{P(t)}(j_1, \ldots, j_N)$
8: **for all** $d \in \delta(t)$ **do** ▶ Multiply the vector r with corresponding matrix rows.
9: $r \leftarrow r \otimes \mathbf{U}^{(d)}(j_d, :)$
10: $\mathcal{V}_t(i_1, \ldots, i_N) \leftarrow \mathcal{V}_t(i_1, \ldots, i_N) + r$ ▶ Do the update due to element $(j_1, \ldots, j_N)$.

Parallel truncated SVD

Following the TTMc in a mode n, HOOI requires finding the leading R_n singular vectors of the matricized tensor $\mathbf{Y}_{(n)}$ to update the matrix $\mathbf{U}^{(n)}$. Here, $\mathbf{Y}_{(n)}$ is of size $I_n \times \prod_{t \neq n} R_t$. In Algorithm 11, we directly compute this matricized tensor, and thereby avoid the cost of matricization. In a recent work on parallel Tucker decomposition of dense tensors [5], leading singular vectors of $\mathbf{Y}_{(n)}$ are extracted by computing the eigenvalues of the Gram matrix $\mathbf{Y}_{(n)}{\mathbf{Y}_{(n)}}^T$ of size $\mathbb{R}^{I_n \times I_n}$, where I_n is typically in the order of thousands. However, I_n can easily be in the order of millions for sparse tensors, rendering this method impractical for our purpose. Also, direct methods for eigenvalue and singular value problems typically compute all eigen/singular values at once, whereas in HOOI we need only R_n left leading singular value/vector pairs out of $\min(I_n, \prod_{i \neq n} R_i)$. For these reasons, we resort to using matrix-free iterative methods to compute the TRSVD of $\mathbf{Y}_{(n)}$ [87]. This way, we avoid forming the Gram matrix, and compute only the required singular value/vector pairs. In a shared memory context, this TRSVD can be parallelized by using optimized BLAS2 gemv kernel for the matrix-vector (MxV) and matrix transpose-vector (MTxV) multiplications, which dominate the computational cost of the

TRSVD due to the matrix being dense. As we demonstrate in the next section, this approach also enables us to reduce the communication requirements in the distributed memory setting. One can also employ randomized linear algebra [34] and cross-approximation techniques [29] to further accelerate this step, yet the proposed parallelization and the communication techniques stay pertinent.

After all factor matrices are updated, the core tensor is formed at Line 10 to check the convergence. Here, since the preceding subiteration of HOOI computes $\mathbf{Y}_{(N)}$, which already holds the tensor $\mathcal{Y} = \mathcal{X} \times_{i \neq N} \mathbf{U}^{(i)^T}$ in the matricized form, we can multiply $\mathcal{Y}$ with $\mathbf{U}^{(N)}$ in mode N to obtain $\mathcal{G}$. Both $\mathcal{Y}$ and $\mathcal{G}$ are dense tensors while $\mathcal{G} \in \mathbb{R}^{R_1 \times \cdots \times R_N}$ being significantly smaller than $\mathcal{Y} \in \mathbb{R}^{R_1 \times \cdots \times R_{N-1} \times I_N}$. The parallel computation of dense $\mathcal{G}$ can be performed efficiently using BLAS3, and in practice its cost should be negligible compared to the cost of sparse irregular operations and TRSVDs carried out in the preceding N subiterations of HOOI. We skip the details of the parallelization of Line 10, and refer the reader to a recent work by Li et al. [68].

5.2.2 Distributed memory parallelism

Coarse-grain parallel HOOI

Recall that there are two main operations for each mode n in an iteration of HOOI: a TTMc step to obtain a matricized tensor $\mathbf{Y}_{(n)}$, and a TRSVD step to obtain the matrix $\mathbf{U}^{(n)}$ from $\mathbf{Y}_{(n)}$. In the coarse-grain task decomposition, we define computing each row $\mathbf{U}^{(n)}(i,:)$ as an atomic task in the TTMc step, and hold the owner of this task responsible for computing $\mathbf{Y}_{(n)}(i,:)$ as well in the TRSVD step. We denote this task by $t_i^{(n)}$, and make the owner of $t_i^{(n)}$ own the corresponding data elements $\mathbf{U}^{(n)}(i,:)$ and $\mathbf{Y}_{(n)}(i,:)$. For each mode n, we partition these tasks. As a result, the process p_k owns the index set $\mathcal{I}_k^{(n)}$ of tasks, i.e., $t_i^{(n)}$ is owned by p_k for each $i \in \mathcal{I}_k^{(n)}$.

For a 3-dimensional $\mathcal{X}$, $\mathbf{Y}_{(1)}(i,:)$ receives a contribution $x_{i,j,k}\left(\mathbf{U}^{(2)}(j,:) \otimes \mathbf{U}^{(3)}(k,:)\right)$ for each nonzero $x_{i,j,k} \in \mathcal{X}$ in Algorithm 10. Therefore, $t_i^{(1)}$ needs all nonzeros in the tensor slice $\mathcal{X}(i,:,:)$ as well as the corresponding rows $\mathbf{U}^{(2)}(j,:)$ and $\mathbf{U}^{(3)}(k,:)$ to perform the Kronecker product. To compute $\mathbf{Y}_{(1)}(i,:)$, $t_i^{(1)}$ involves $|\mathcal{X}(i,:,:)|$ Kronecker products due to TTMc. During this computation, for each nonzero $x_{i,j,k}$, $t_i^{(1)}$ needs the data owned by $t_j^{(2)}$ and $t_k^{(3)}$ to perform the Kronecker product. This specifies the data to be exchanged in the communication step.

Algorithm 13 gives the distributed memory parallel HOOI executed by the process p_k. Initially, we assume a partition of task indices $\mathcal{I}_k^{(n)}$ for each mode n, as well as the set of nonzeros $\mathcal{X}_k$ that are needed to perform the local computations associated with these tasks. At Lines 1–6, p_k performs the symbolic TTMc with its local tensor $\mathcal{X}_k$. Next, at Lines 9–12 the local TTMc operation $\mathbf{Y}_{(n)} \leftarrow \mathcal{X}_k \times_{i \neq n} \mathbf{U}^{(i)}$ is performed. Note that at Line 4, we set $K_n \leftarrow \mathcal{I}_k^{(n)}$ to make sure that the coarse-grain algorithm only computes TTMc results for the owned set of rows $\mathcal{I}_k^{(n)}$. Also, p_k does not need to store the whole matrix $\mathbf{U}^{(n)}$; instead, it stores the set of owned rows $\mathbf{U}^{(n)}(\mathcal{I}_k^{(n)},:)$, and the rows i_n that are accessed in the local TTMc computations due to $x_{i_1,\ldots,i_n,\ldots,i_N} \in \mathcal{X}^k$.

After the local TTMc step, we obtain the row-wise distributed matrix $\mathbf{Y}_{(n)}$ where p_k owns the set $\mathcal{I}_k^{(n)}$ of rows. For the subsequent TRSVD step, we need to perform the MxV and MTxV multiplications $y \leftarrow \mathbf{Y}_{(n)}x$ and $x^T \leftarrow y^T\mathbf{Y}_{(n)}$. We use a block partitioning for the vector x. On the other hand,

Algorithm 13 Distributed memory parallel HOOI executed at process p_k

Input: $\mathcal{X}_k$: Partition of $\mathcal{X}$ owned by the process p_k
type: 'coarse-grain' or 'fine-grain'
$\mathcal{I}_k^{(1)}, \ldots, \mathcal{I}_k^{(N)}$: Set of task indices owned by p_k in each mode
$\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$: Initial factor matrices
$R_1, \ldots, R_N$: Ranks of approximation
Output: $[\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$: Tucker approximation of $\mathcal{X}$

parfor $n = 1$ **to** N **do**
  $\{ul_n, \mathcal{J}_n\} \leftarrow$ SymbolicTTMc($\mathcal{X}_k, \{1, \ldots, N\} \setminus \{n\}$)
  if type = 'coarse-grain' **then**
    $K_n \leftarrow \mathcal{I}_k^{(n)}$
  else
    $K_n \leftarrow \mathcal{J}_n$
repeat
  for $n = 1$ **to** N **do**
    parfor all $i \in K_n$ **do** ▶ TTMc for mode n
      $\mathbf{Y}_{(n)}(i,:) \leftarrow 0$
      for all $x_{i_1,\ldots,i_N} \in ul_n(i)$ **do**
        $\mathbf{Y}_{(n)}(i,:) \mathrel{+}= x_{i_1,\ldots,i_N} [\otimes_{t \neq n} \mathbf{U}^{(t)}(i_t,:)]$
    $\mathbf{U}^{(n)} \leftarrow$ TRSVD($\mathbf{Y}_{(n)}, R_n$)
    Send/receive the updated rows of $\mathbf{U}^{(n)}$
  $\mathcal{G}_k \leftarrow \mathcal{Y} \times_N \mathbf{U}^{(N)}$
  $\mathcal{G} \leftarrow$ AllReduce($\mathcal{G}_k$)
until convergence or maximum number of iterations
return $[\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

we partition the vector y according to $\mathcal{I}_k^{(n)}$. This way, after gathering all entries of x at all processes, the MxV operation $y \leftarrow \mathbf{Y}_{(n)}x$ can be performed locally without any communication on y entries. Also, after the TRSVD solver converges, the computed left singular vectors have the same partition as y. As a result, p_k ends up having all rows $\mathbf{U}^{(n)}(\mathcal{I}_k^{(n)},:)$ in place, avoiding any post-communication. For $x^T \leftarrow y^T\mathbf{Y}_{(n)}$, we compute the local result $y^T\mathbf{Y}_{(n)}$, then perform an all-to-all reduction to sum up the final results in the owner processes.

In our recent work [50], we used a similar coarse-grain task definition in the context of the parallel CP-ALS algorithm, and proposed a hypergraph model for representing the computational and communication requirements of the parallel algorithm. Here, we adopt the same hypergraph model to reduce the total communication volume and to balance the computational load during the HOOI iterations. In this model, we represent tasks with vertices and their interdependence using hyperedges. For each task $t_i^{(n)}$, we add a vertex and a hyperedge to the hypergraph. For $N = 3$, we set the computational weights $|\mathcal{X}(i,:,:)|$, $|\mathcal{X}(:,j,:)|$, and $|\mathcal{X}(:,:,k)|$ to the tasks $t_i^{(1)}$, $t_j^{(2)}$, and $t_k^{(3)}$, respectively. This becomes an exact measure for the computational cost of TTMc, as for each nonzero we perform an outer product with $N-1$ vectors. One can potentially associate a secondary computational cost of 1 with each task to balance the number of rows owned per process for the MxV and MTxV multiplication steps of the TRSVD step; yet in practice we observed that not adding this constraint still provides balanced number of rows. For each nonzero $x_{i,j,k}$ that imposes an update on tasks $t_i^{(1)}$, $t_j^{(2)}$, and $t_k^{(3)}$, we connect the vertices of each such task with the corresponding 3 hyperedges to model their data dependency. The standard partitioning problem of this hypergraph corresponds to reducing the total communication volume while establishing the load balance in the TTMc step. This could also ensure

load balance in the TRSVD step if the processes have almost equal number of tasks (we investigate this in our experiments). Finally, as mentioned above, only the MTxV operation in the TRSVD step requires communication, which is regular and has a cost independent from the task distribution.

Fine-grain parallel HOOI

The coarse-grain approach has two main limitations. First, the number of tasks for a dimension n is limited by I_n. In case the tensor is very small in one of the dimensions, this poses a granularity problem by not having enough tasks for parallelism, which hinders scalability. Also, coarse-grain tasks tend to be heavily interdependent with their data. As a result, there is typically little room in finding a good partition for parallelization. To address both limitations, we propose a fine-grain variant of the parallel HOOI which enables the partial computation of the rows of $\mathbf{Y}_n$. The fine-grain approach in Algorithm 13 differs from the coarse-grain one at Line 6, which enables computing partial results for the rows that are not owned by p_k.

Similar to the coarse-grain algorithm, we define the task $t_i^{(n)}$ to denote the ownership of $\mathbf{U}^{(n)}(i,:)$. In the fine-grain case, the owner of $t_i^{(n)}$ does not necessarily perform all the computations associated with $\mathbf{Y}_{(n)}(i,:)$ nor $\mathbf{U}^{(n)}(i,:)$. For each nonzero $x_{i_1,\ldots,i_N} \in \boldsymbol{\mathcal{X}}$, we define an associated task $z_{i_1,\ldots,i_N}$, and let the process p_k having the set of nonzeros $\boldsymbol{\mathcal{X}}_k$ also own the corresponding z-type tasks. For each mode n, the owner of $z_{i_1,\ldots,i_N}$ is responsible for performing the operation $x_{i_1,\ldots,i_N}[\otimes_{t\neq n}\mathbf{U}^{(t)}(i_t,:)]$ and generating a *partial* result for $\mathbf{Y}_{(n)}(i_n,:)$. One may consider merging these partial results in a way that the owner of the task $t_i^{(n)}$ gets the final result $\mathbf{Y}_{(n)}(i_n,:)$. This way, one can proceed with the TRSVD computation of the row-wise distributed matrix $\mathbf{Y}_{(n)}$ just as in the coarse-grain case. However, the problem in this scenario is that each partial result $\mathbf{Y}_{(n)}(i,:)$ to be communicated is of size $\prod_{i\neq n} R_i$, which is exponential in the ranks of approximation, and can easily get very large. In contrast, each message for communicating the rows of $\mathbf{U}^{(n)}$ at Line 14 of Algorithm 13 is of size R_n.

We make use of the following observation to asymptotically reduce the communication cost due to partial results of TTMc. In a P-way parallel execution, performing the local TTMc with a fine-grain task distribution produces the matrix $\mathbf{Y}_{(n)}$ in the sum-distributed form $\mathbf{Y}_{(n)} = \mathbf{Y}_{(n)}^1 + \cdots + \mathbf{Y}_{(n)}^P$ where $\mathbf{Y}_{(n)}^k$ is the partial TTMc result generated by the process p_k. One should avoid assembling $\mathbf{Y}_{(n)}$, otherwise a high communication overhead incurs. Fortunately, we only need to provide MxV and MTxV operations associated with the matrix $\mathbf{Y}_{(n)}$ in the subsequent TRSVD step. We can perform these multiplications without assembling $\mathbf{Y}_{(n)}$ as follows. For the MxV operation, we perform $y \leftarrow \mathbf{Y}_{(n)}^k x$ at each process p_k and generate a partial result on y. Then, we execute a point to point communication on the entries of y so that the owner of the task $t_i^{(n)}$ sums all the partial results to obtain the final value of y_i. In this way, instead of communicating a partial result $\mathbf{Y}_{(n)}^k(i,:)$ of size $\prod_{i\neq n} R_i$, we communicate a single vector entry y_i for each i requiring communication in each iteration of the TRSVD solver. The number of TRSVD iterations is typically in the order of R_n or less, so it makes this communication cost conformal with the cost at Line 14 of Algorithm 13. We can easily perform the MTxV operation by computing $x^T \leftarrow y^T\mathbf{Y}_{(n)}^k$ at each process p_k, and then by performing an all-to-all communication on x^T just as in the coarse-grain algorithm.

We model the tasks of the fine-grain algorithm and their dependencies using the same hypergraph from our previous study [50], where we model tasks with vertices and the dependencies among tasks with hyperedges. With this model, the communication cost at Lines 13–14 of Algorithm 13 is equal to

the cutsize of a partition of the corresponding hypergraph and can be effectively reduced by existing hypergraph partitioning tools. We refer the reader to [50] for the detailed analysis. We note also that by not combining the partial results of $\mathbf{Y}_{(n)}$, we increase the total computational load of matrix-vector multiplications in the TRSVD solver, as we end up having more rows than $|I_n|$ to multiply in total. Fortunately, this increase is also equal to the cutsize of the hypergraph, and is significantly reduced with a good partition. Therefore, minimizing the cutsize is beneficial for reducing both the communication cost of the parallel HOOI and the redundant computation in its TRSVD step. Finally, 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 [45, 84]. We do not explicitly address this problem and hope that assigning equal amount of n-dimensional indices will lead to load balance.

5.3 Related work

We give a brief overview of the recent progress on efficient tensor decomposition (CP and Tucker) algorithms. These can be categorized into four classes: (i) toolboxes for Matlab and similar environments [3, 6, 7, 59, 79]; (ii) implementations for shared memory systems [9, 10, 68, 98]; (iii) implementations based on MapReduce paradigm [39, 40]; (iv) implementations for distributed memory systems [5, 22, 50, 94]. 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 of a single machine. 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.

To 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 [39] is a MapReduce based HOOI implementation. Li et al. [68] 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. This work does not discuss other components of an HOOI implementation. Austin et al. [5] propose a distributed memory parallel implementation of HOOI for dense tensors. The challenges that are faced are very different from those faced in the sparse case, essentially due to all communications involving all processes, and memory accesses being regular in the dense case.

5.4 Experiments

We conducted our experiments on two parallel computing platforms. The first platform is the **Blue Gene/Q** described in Section 2.7. On this cluster, we ran our experiments up to 256 nodes (4096 cores) where we achieved the maximum scalability. Each core of PowerPC A2 can handle one arithmetic and memory operation simultaneously; therefore we assigned 32 threads per node (2 threads per core) to benefit from this. All codes we used in our benchmarks were compiled using the Clang C++ compiler (version 3.6.0) with IBM MPI wrapper using -O3 option for compiler optimizations, and

linked against IBM ESSL library for LAPACK and BLAS routines. Our code depends on PETSc and SLEPc (version 3.6.2) libraries for the distributed truncated SVD computations. The second one is the the **Ada** cluster of IDRIS, 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 using 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. Finally, we conducted our experiments using four tensors, namely Netflix, NELL, Flickr, and Delicious, described in Section 2.6.

Our parallel algorithms are independent from the partitioning method. Therefore, we use two partitioning methods to test their performance. The first partitioning method assigns the tasks uniformly at random to processes (for coarse-grain tasks we use a blocked variant), and the second one uses hypergraph partitioning tool PaToH [17]. The first method is fast, promises load balance, but it does not pay attention to the communication overhead. The second method achieves load balance, reduces communication, but is time consuming. Speedups using these two partitioning methods show the worst case behavior and the potential of the parallel algorithms, if one is willing to pay the preprocessing cost. In general, a good parallel algorithm should deliver good performance with the second partitioning method; but it should also enjoy acceptable speed up with the first partitioning method.

We used PaToH (version 3.2) with default options to partition the hypergraphs. 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 costly. Yet in most applications, the tensors from the real-world data are built incrementally and analyzed repetitively. In this scenario, 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 [54]. In these cases, the time spent in partitioning can be amortized across multiple runs.

To the best of our knowledge, HaTen2 [39] is the only parallel implementation of HOOI (see Section 5.3). All reported parallel runtimes of HaTen2 [39] are larger than that of the sequential execution of MET algorithm [7,59], and the sequential runtime of our method is less than that of MET. For example, on a random tensor of size $10\text{K} \times 10\text{K} \times 10\text{K}$ with 1M nonzeros, Tucker decomposition with five HOOI iterations took 87.2 seconds in MET and 11.3 seconds in our method (on a single core), including all preprocessing. This difference is expected as neither HaTen2 (which uses MapReduce) nor MET (which is a Matlab tool) are made for high performance; thus, we do not report further comparisons.

We set the ranks of approximation $R_1 = R_2 = R_3 = 10$ for the 3-dimensional tensors, and $R_1 = R_2 = R_3 = R_4 = 5$ for the 4-dimensional tensors, which is a viable choice in data analysis applications [103, 110]. We then run the parallel HOOI for 5 iterations, and report the average time spent per HOOI iteration. The symbolic TTMc is expected take much less time than the HOOI iterations. For instance, in 256-way parallel execution of Algorithm 13 using the fine-grain hypergraph partitioning for 5 iterations, symbolic TTMc took 14%, 12%, 19%, and 5% of the total execution time for Delicious, Flickr, Netflix, and NELL tensors, respectively. In general, HOOI is expected to run for more iterations; so this cost is expected to become less important. In addition, finding a

Table 5.1: Time spent per iteration (in seconds) for our distributed memory parallel HOOI using two threads per core with different partitions. Missing results are due to insufficient memory.

#nodes×#cores	Delicious				Flickr			
	fine-hp	fine-rd	coarse-hp	coarse-bl	fine-hp	fine-rd	coarse-hp	coarse-bl
8 × 16	164.9	-	235.3	400.5	206.2	-	287.5	308.5
16 × 16	85.2	162.0	197.5	302.4	115.6	221.8	210.5	230.1
32 × 16	47.6	96.2	155.6	206.5	64.6	124.5	166.3	190.1
64 × 16	27.2	57.8	98.9	159.6	36.8	69.9	124.1	129.0
128 × 16	18.2	34.7	80.8	96.4	22.6	42.9	87.9	102.3
256 × 16	12.2	22.1	65.1	77.1	20.0	29.2	73.8	86.3

#nodes×#cores	NELL				Netflix			
	fine-hp	fine-rd	coarse-hp	coarse-bl	fine-hp	fine-rd	coarse-hp	coarse-bl
1 × 16	222.1	222.1	240.1	240.1	-	-	-	-
2 × 16	151.6	137.6	198.5	164.4	-	-	-	-
4 × 16	87.7	75.9	180.6	131.4	33.7	39.2	46.0	42.8
8 × 16	67.8	46.9	172.5	109.7	18.6	26.1	30.6	33.4
16 × 16	54.9	28.3	112.4	94.1	10.3	18.3	32.2	27.8
32 × 16	43.9	17.2	73.8	68.2	5.7	13.9	26.2	26.7
64 × 16	35.4	11.9	67.1	54.5	3.9	10.9	26.2	21.7
128 × 16	26.7	8.4	50.3	48.5	2.9	8.7	19.8	18.7
256 × 16	14.8	7.7	48.1	44.9	3.8	8.3	14.7	16.1

good Tucker approximation of a tensor typically involves executing HOOI algorithm with various ranks [54]; symbolic TTMc can be computed once and used for all these executions.

5.4.1 Distributed memory results

In Table 5.1, we give the strong scalability results for our distributed memory parallel HOOI algorithm. We report the average time spent per iteration in Algorithm 13. For each tensor in our dataset, we report two results for fine-grain and coarse-grain parallel algorithms (with two different partitioning methods). Those with the suffix "-hp" uses PaToH's partitionings; "fine-rd" refers to a random partitioning, whereas and "coarse-bl" corresponds to a contiguous block partitioning of tasks. We evaluate the algorithms up to 256 compute nodes where each node executes Algorithm 13 using 32 threads. Since the amount of memory per node of Blue Gene/Q is only 16GBs, some runs were not feasible. These missing runs are shown with "-" in Table 5.1. In two tensors, using the fine-rd partition was feasible only after 16 nodes due to higher memory requirements for storing partial results (which is proportional to the communication cost).

We first observe in Table 5.1 that in all test cases our algorithm graciously scale up to 256 MPI ranks (or 4096 cores), except with the fine-hp partition on Netflix tensor, where we observe a slowdown at 256 nodes. On Delicious tensor, our parallel algorithm achieves 13.5x speedup using 256 nodes over the run on 8 nodes with the fine-hp partition. With the fine-rd partition, the algorithm also scales to 256 nodes, although it runs almost two times slower than fine-hp. This is expected, as fine-rd targets load balance but incurs more communication overhead. Similarly on Flickr, our parallel algorithm achieves 10.3x speedup using 256 nodes over its execution on 8 nodes with the fine-hp partition, and runs roughly twice as fast as the configuration with the fine-rd partition. Unfortunately,

with Delicious, Flickr, and Netflix datasets, we were not able to get the sequential and shared memory parallel timings on a single node due to data not fitting into the memory; hence we cannot provide overall speedup results over the sequential execution. For those results, we refer the reader to a technical report [49], where we present speedups up to 742x on a different architecture with more memory. On NELL tensor, we managed to get runs on a single node. Using the fine-rd partition using 256 nodes (4096 cores), we obtained 280x speedup over the sequential execution. This translates into 29x speedup over the execution on a single node. Yet, unlike other three tensors, on this data the fine-hp partition lead to slower execution than the fine-rd partition. We analyzed the underlying reason for this result on 256 MPI ranks and saw that the *maximum* communication volume per process for with the fine-hp partition in the dominant dimension was 543K in contrast to 366K in the fine-rd partition. Here, this entailed a large overhead that could not be compensated by the reduction in the total communication volume (20M vs 94M).

In Table 5.2, we give the computation and communication requirements of one HOOI iteration on the Flickr tensor with all partitionings for 256 MPI ranks (4096 cores). In this table, $\mathcal{W}_{TTMc}$ and $\mathcal{W}_{TRSVD}$ correspond to the computational load of the TTMc and TRSVD steps of Algorithm 13 and Comm. vol. corresponds to the volume of send/receive communication incurred by different partitioning methods. We give both the average and the maximum values across all processes for all three metrics.

We observe in the $\mathcal{W}_{TTMc}$ columns of Table 5.2 that TTMc work per MPI-rank is always well balanced with the fine-grain partitions. This is owing to the finer granularity of tasks which allows perfect balance. On the other hand, with the coarse-grain formulation, the TTMc tasks are not well balanced, as some tasks might be significantly more costly than others. Particularly in the TTMc computation of the 4th dimension, we observe some computational imbalance of 436% and 471% using the coarse-hp and coarse-block partitions.

We realize in the $\mathcal{W}_{TRSVD}$ columns of Table 5.2 that the average TRSVD work $\mathcal{W}_{TRSVD}$ given by the fine-hp partition is only slightly higher than that given by the coarse-grain partitions, which introduce no overhead to the TRSVD computation. Particularly, $\mathcal{W}_{TRSVD}$ is dominant in the third dimension, and the fine-hp partition results in the same total/average work (110K) as in the coarse-grain partitions. Using the random partition (fine-rd), however, the average $\mathcal{W}_{TRSVD}$ is drastically increased to 435K in the same dimension. Moreover, even though none of the partitioning methods explicitly try to establish load balance for $\mathcal{W}_{TRSVD}$, we observe that the obtained load balance is generally acceptable. In the computationally dominant third mode, the fine-hp partition leads to 100% load imbalance, whereas the fine-rd, coarse-hp, and coarse-block partitions lead to 25%, 79%, and 18%.

The last two columns of Table 5.2 show the maximum and the average communication volume per process. This involves the send and receive volumes at Lines 13 and 14 of Algorithm 13. Using the fine-hp and fine-rd partitions, the average communication volumes are 11K and 1735K, respectively. Recall that the average communication volume is proportional to the cutsize of the corresponding hypergraph partition, and the cutsize equals to the total redundancy in MxV and MTxV operations. That is why we observe higher $\mathcal{W}_{TRSVD}$ value using the fine-rd partition. The maximum communication volume per process also decreases to 166K with fine-hp partition, in comparison to 1744K using fine-rd partition. Our final observation is that fine-hp partitions are more effective in reducing the communication volume than the coarse-hp partitions.

In Table 5.3, we provide the relative timings of TTMc, TRSVD, and the computation of the core

Table 5.2: Statistics for the computation and communication requirements with different partitionings of Flickr in one HOOI iteration for 4096-way parallel run with 256 MPI ranks.

Mode	$\mathcal{W}_{TTMc}$		$\mathcal{W}_{TRSVD}$		Comm. vol.	
	Max	Avg	Max	Avg	Max	Avg
fine-hp						
1	441K	441K	590	507	2218	2029
2	441K	441K	8778	5656	24K	17K
3	441K	441K	221K	110K	166K	11K
4	441K	441K	32K	19K	77K	53K
fine-rd						
1	443K	441K	668	648	5884	2597
2	443K	441K	98K	96K	409K	385K
3	443K	441K	545K	435K	1744K	1735K
4	443K	441K	110K	100K	432K	413K
coarse-hp						
1	718K	441K	22	3	4910	1213
2	810K	441K	2700	248	118K	66K
3	798K	441K	197K	110K	3187K	810K
4	2368K	441K	13K	6250	170K	102K
coarse-block						
1	958K	441K	252	3	18K	907
2	756K	441K	5401	248	126K	80K
3	441K	441K	130K	110K	3324K	1250K
4	2518K	441K	60K	6250	296K	138K

Table 5.3: Relative timings of TTMc, TRSVD, and core tensor computation steps within HOOI using 256-way fine-hp partition (in percentage).

Step	Delicious	Flickr	NELL	Netflix
TTMc	75.6	64.6	71.2	27.7
TRSVD+comm	19.2	32.6	24.8	71.6
core+comm	5.2	2.8	4.0	0.7

Table 5.4: Time spent per iteration (in seconds) for shared memory parallel HOOI on node(s) with a 16-core processor. The number of MPI ranks used for each tensor is shown in the parentheses.

#threads	Delicious (8)	Flickr (8)	NELL (1)	Netflix (4)
1	1182.7	1055.8	2173.6	660.1
2	634.5	583.2	1146.3	330.8
4	361.1	354.5	616.4	167.6
8	227.5	241.6	354.1	87.3
16	173.2	201.0	252.7	48.7
32	164.9	206.2	222.7	33.7

tensor within an iteration of HOOI using the 256-way fine-hp partition. The TRSVD timings include the time spent in the communication of the vector entries in the MxV and MTxV operations, as the communication takes place within the PETSc calls. We see in Table 5.1 that on Netflix tensor with the fine-hp partition, we lose scalability at 256 nodes. Table 5.3 shows that for this instance TRSVD step begins to dominate the timings, and the communication cost starts to prevent further scalability. Note that in this case, the increase in the communication cost and imbalance also affects the computational cost of the TRSVD step. This is because of the fact that for each communicated vector entry there is an associated unit of redundant work in the MxV and MTxV computations. We also realize in these results that the cost of forming the core tensor $\mathcal{G}$ with a TTM followed by an ALLREDUCE communication is negligible, as we expected. Finally, we inform that in all instances, TRSVD (as provided by SLEPc) converged in less than 5 iterations.

5.4.2 Shared memory results

We evaluate the shared memory scalability of the distributed memory HOOI algorithm by varying the number of threads from 1 to 32, and using the minimum number of nodes possible. We needed 8, 8, 4, and 1 nodes to be able to execute the code on Delicious, Flickr, Netflix, and NELL, respectively, using the fine-hp partitions.

TTMc is a memory latency-bound operation; for each nonzero $x_{i,j,k}$, the access to $\mathbf{U}^{(1)}(i,:)$, $\mathbf{U}^{(2)}(j,:)$, and $\mathbf{U}^{(3)}(k,:)$ likely results in a cache miss, due to the irregular pattern of nonzeros. Multi-threading is an effective way of hiding this latency; therefore it offers a great opportunity of acceleration through parallelism. However, the MxV and MTxV operations in the TRSVD step are memory bandwidth-bound due to the matrices being dense. Once the memory bandwidth is saturated, one may not expect a notable speedup with multi-threading (except in a NUMA architecture where each socket has an independent memory bandwidth; yet in this case the parallelism within a socket has the same issue). The TTMc operation count is proportional to the number of nonzeros, whereas

Table 5.5: Comparison of the sequential and 2048-way parallel run times (in seconds) using fine-hp partitioning, with (dtree) and without (flat) dimension trees

TTMc scheme/#procs	Netflix	NELL	Delicious	Flickr
flat/1	166	176	669	772
dtree/1	123	129	560	678
flat/2048	0.44	1.3	2.09	1.04
dtree/2048	0.39	1.2	0.97	0.90

the TRSVD cost is proportional to the number of rows of the matrix (or equivalently, the size of a dimension of the tensor).

As seen in Table 5.4, we manage to improve the runtime using 32 threads for all tensors except Flickr. Using 32 threads, the speedups we obtain for Delicious and Flickr tensors are 7.2x and 5.1x, whereas on NELL and Netflix we get 9.8x and 20x, respectively. Delicious and Flickr tensors have very large third dimension of size 17M and 28M, whereas the largest dimensions of NELL and Netflix are of size 3.2M and 480K. Therefore, NELL and particularly Netflix have more latency-bound computations and provide more room for speedup, which explains the better speedup results in comparison to Delicious and Flickr tensors. Another interesting point is that on the Netflix tensor, using 16 threads we achieve 13.8x speedup over the single threaded execution. Increasing to 32 threads results in a superlinear speedup of 20x on 16 cores. We believe that this is mostly due to each core being able to execute two threads (one for memory and one for compute operations) simultaneously, which is particularly advantageous for latency-bound sparse irregular operations.

5.4.3 The effect of dimension trees

We now compare the sequential and parallel execution timings of HOOI with and without dimension trees to evaluate its effect. These timings are provided in Table 5.5 on **Ada** cluster for sequential as well as parallel executions using 256 ranks and 8 threads. Here, "flat" refers to the standard nonzero-based TTMc computation given in Algorithm 10, whereas "dtree" represents the dimensional tree-based computational scheme.

First, we note that in all instances, using a dimension tree yields faster execution. In the sequential execution, dimension tree provides 35%, 36%, 19%, and 14% speedup over the traditional scheme on Netflix, NELL, Delicious, and Flickr tensors, respectively, wheres in parallel runs it results in 13%, 8%, 215%, and 16% faster executions. This clearly shows that storing and reusing partial TTMc results indeed reduces the computational cost notably both in sequential and parallel runs.

5.5 Conclusion

We discussed an efficient parallelization of the alternating least squares-based Tucker decomposition algorithm (HOOI, also called Tucker-ALS) for sparse tensors in shared and distributed memory systems. We introduced a nonzero-based TTMc formulation, and proposed a shared-memory parallel HOOI with a preceding symbolic computation step that uses this formulation. We proposed a coarse and a fine-grain parallel algorithm with their corresponding task definitions, and investigated the issues of load balance and communication cost reduction on different components of the parallel

algorithms. Gathering all these together, we achieved scalability up to 4096 cores using 256 MPI ranks on real world tensors with an efficient hybrid OpenMP-MPI implementation within our high performance parallel sparse tensor library, HyperTensor.

We finally note that the TTMc operation is used in other algorithms [26] for Tucker decomposition; therefore, proposed methods of parallelism can be used by those algorithms.

Chapter 6

Parallel Sparse NMF

Though the main focus of this thesis is mainly on tensor decompositions, this chapter investigates an efficient computation of a widely used tool in data analysis and machine learning applications, namely non-negative sparse matrix factorization, whose parallelization involves striking similarities with parallel sparse tensor decompositions presented in the preceding chapters. This opens up to possibility of properly adopting these techniques for obtaining a fast distributed sparse non-negative matrix factorization algorithm, which is the ultimate goal of the work presented in this chapter.

Non-negative matrix factorization (NMF) is the problem of finding two low rank factors $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ for a given input matrix $\mathbf{A} \in \mathbb{R}_+^{m \times n}$ such that $\mathbf{A} \approx \mathbf{WH}$. Here, $\mathbb{R}_+^{m \times n}$ denotes the set of $m \times n$ matrices with non-negative real values. Formally, the NMF problem [90] can be defined as

$$\min_{\mathbf{W} \geqslant 0, \mathbf{H} \geqslant 0} \|\mathbf{A} - \mathbf{WH}\|_F + \phi(\mathbf{W}) + \psi(\mathbf{H}), \tag{6.1}$$

where $\|\mathbf{X}\|_F = (\sum_{ij} x_{ij}^2)^{1/2}$ is the Frobenius norm. The functions $\phi(\mathbf{W})$ and $\psi(\mathbf{H})$ are called regularization functions that prevent the model from overfitting the data, and these functions are chosen depending on the characteristics of the data. In our case, we are considering $\phi(\mathbf{W}) = \alpha\|\mathbf{W}\|_F^2$, called ℓ_2 regularizer, and $\psi(\mathbf{H}) = \beta \sum_{i=1}^{n} \|\mathbf{h}_i\|_1^2$, called ℓ_1 regularizer, for addressing the inherent sparsity in the input matrix. The lack of ℓ_1 regularization on the matrix $\mathbf{H}$ can result in numerical instability.

NMF is widely used in data mining and machine learning as a dimensionality reduction and factor analysis method. It is a natural fit for many real world problems as the non-negativity is inherent in many representations of real-world data, in which case the resulting low rank factors are expected to have a natural interpretation. The applications of NMF range from text mining [80], computer vision [38], and bioinformatics [55] to blind source separation [24], unsupervised clustering [61, 62], and many others. In most real-world applications, m and n can be on the order of millions or more while k being much smaller in the order of tens to thousands.

We would like to highlight that "non-negative" matrix factorization is not the matrix factorization in collaborative filtering for recommender systems, for which many implementations exist. The collaborative filtering problem is different than NMF – the focus of this chapter– because it interprets the "*zero*" entries of the input matrix as missing data, while the NMF problem is defined for a completely known input matrix and does not handle missing values. This leads to different optimization problems, algorithms, and computational steps. Specifically, computing NMF involves three main types of operations; multiplication of $\mathbf{A}$ with a factor matrix $\mathbf{W}$ or $\mathbf{H}$, computing the Gram matrices

$\mathbf{W}^T\mathbf{W}$ and $\mathbf{H}\mathbf{H}^T$, and solving a non-linear least squares (NNLS) problem using these two resulting dense matrices to update factor matrices, which can be performed directly (all columns at once) or column-by-column within the ALS framework [2]. Also, while ALS framework is popularly used for NMF, direct techniques updating all factors at once are known to be more robust particularly in ill-conditioned cases albeit requiring higher per-iteration computational cost [25]. Nevertheless, the approaches share similarities from a computational perspective, hence the contributions in this chapter remain pertinent in both cases.

To the best of our knowledge, the only high performance software available for parallelizing this computation is MPIFAUN [42] which operates on both sparse and dense input matrices. For parallelism, it employs a fixed 2D uniform partitioning on $\mathbf{A}$, and partitions factor matrices $\mathbf{W}$ and $\mathbf{H}$ conformally with this 2D partition. Each process works on its local matrix block of $\mathbf{A}$, and communicates the rows and columns of $\mathbf{W}$ and $\mathbf{H}$ corresponding to its process row and column in the 2D topology. This communication is *optimal* in the dense case in the sense that no process receives a data element not used in its local computations. In the sparse case, however, the actual communication requirements of processes depend on the sparsity of $\mathbf{A}$ and its partitioning, and is in general significantly less than the dense case. Therefore, employing the same collective communication scheme in the sparse case brings about a major redundancy in the communication cost. Also, MPIFAUN similarly uses only fixed 2D uniform partitioning on sparse matrices; this creates significant imbalance both in computation and communication as typically the data is not homogenously distributed in $\mathbf{A}$. In overall, the lack of an efficient sparse communication scheme that allows flexibility in partitioning hinders the scalability of this approach.

We summarize the contributions in this chapter as follows:

- We propose an efficient parallel NMF algorithm called DIST-SPNMF for sparse matrices that employs a point-to-point communication scheme to leverage the sparsity of the input matrix, and eliminate any redundant communication existing in MPIFAUN. The algorithm is flexible to work with any partition of the input matrix $\mathbf{A}$ as well as factor matrices $\mathbf{W}$ and $\mathbf{H}$.

- We employ a 2D cartesion process topology in a way that eliminates the communication cost of parallel NMF in one dimension while bounding the maximum number of messages sent per process in our MPI implementation.

- We introduce effective partitioning strategies to further reduce the communication cost and enhance the parallel scalability of our algorithm.

- We introduce regularization to NMF computations to prevent potential numerical instabilities using large matrices.

- We compare our implementation with MPIFAUN, and report scalability results up to 32678 processors on two different parallel computing platforms using two real-world datasets.

Table 6.1 summarizes the notation we use throughout this chapter. We use subscripts to refer to sub-blocks of matrices. For example, $\mathbf{A}_{ij}$ refers to the sub-block (i, j) of $\mathbf{A}$ in a 2D partition. We use m and n to denote the numbers of rows and columns of $\mathbf{A}$, respectively, and assume w.l.g that $m \geqslant n$ throughout.

$\mathbf{A}$	Input matrix
$\mathbf{W}$	Left low rank factor
$\mathbf{H}$	Right low rank factor
m	Number of rows of $\mathbf{A}$
n	Number of columns of $\mathbf{A}$
k	Low rank parameter
P	Number of parallel processes
P_r	Number of rows in the processor grid
P_c	Number of columns in the processor grid
$\mathbf{A}_p$	Submatrix of $\mathbf{A}$ owned by process p
$\mathcal{I}_p$,$\mathcal{J}_p$	Set of rows/columns of $\mathbf{W}$/$\mathbf{H}$ owned by process p
$\mathcal{F}_p$,$\mathcal{G}_p$	Set of unique row and column indices of $\mathbf{A}_p$
$\mathbf{W}(\mathcal{I}_p, :)$	Owned rows of $\mathbf{W}$ by process p
$\mathbf{H}(:, \mathcal{J}_p)$	Owned columns of $\mathbf{H}$ by process p

Table 6.1: Notation.

6.1 Sparse non-negative matrix factorization

The Alternating-Updating NMF (AU-NMF) algorithms are those that alternate between updating one of $\mathbf{W}$ and $\mathbf{H}$ using the given input matrix $\mathbf{A}$ and other 'fixed' factor - $\mathbf{H}$ for updating $\mathbf{W}$, and $\mathbf{W}$ for updating $\mathbf{H}$. This update is performed using the Gram matrix associated with the fixed factor matrix ($\mathbf{H}\mathbf{H}^T$ for $\mathbf{W}$ and $\mathbf{W}^T\mathbf{W}$ for $\mathbf{H}$), and the product of the input matrix $\mathbf{A}$ with the fixed factor matrix ($\mathbf{A}\mathbf{H}^T$ or $\mathbf{W}^T\mathbf{A}$). We show this framework in Algorithm 14.

After computing the Gram matrix and the multiplication of $\mathbf{A}$ with the fixed factor matrix, the specifics of the updates at Lines 3 and 4 depend on the NMF algorithm, and we refer to the computation associated with these lines as the Local Update Computations (**LUC**).

Algorithm 14 AU-NMF: Alternating-Updating NMF algorithm

Input: $\mathbf{A}$: An $m \times n$ matrix
k: The rank of approximation
Output: $[\mathbf{W}, \mathbf{H}]$: Factor matrices
1: Initialize $\mathbf{H}$ with a non-negative matrix in $\mathbb{R}_+^{n\times k}$.
2: **while** stopping criteria not met **do**
3: Update $\mathbf{W}$ using $\mathbf{H}\mathbf{H}^T$ and $\mathbf{A}\mathbf{H}^T$
4: Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$

We note that AU-NMF is very similar to a two-block, block coordinate descent (BCD) framework as explained by Bertsekas [12]. The BCD framework expresses solving optimization variables in complex non-linear optimization problem as one block at a time, while keeping the others fixed. In NMF, the two blocks are the unknown factors $\mathbf{W}$ and $\mathbf{H}$, and we *solve* the following subproblems,

which have unique solutions for a full rank $\mathbf{H}$ and $\mathbf{W}$:

$$\begin{aligned}\mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \tilde{\mathbf{W}}\mathbf{H}\right\|_F + \phi(\tilde{\mathbf{W}}) + \psi(\mathbf{H}),\\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A} - \mathbf{W}\tilde{\mathbf{H}}\right\|_F + \phi(\mathbf{W}) + \psi(\tilde{\mathbf{H}}).\end{aligned} \tag{6.2}$$

Since each subproblem involves non-negative least squares, this two-block BCD method is also called the Alternating Non-negative Least Squares (ANLS) method [56]. Block Principal Pivoting (**ABPP**) is one algorithm that solves these NLS subproblems. In the context of the AU-NMF algorithm, an ANLS method *maximally* reduces the overall NMF objective function value by finding the optimal solution for given $\mathbf{H}$ and $\mathbf{W}$ in Lines 3 and 4, respectively.

From time to time, these updates do not necessarily solve each of the subproblems (6.2) to optimality but simply improve the overall objective function (6.3), such as in Multiplicative Update (**MU**) [90] and Hierarchical Alternating Least Squares (**HALS**) [24] methods.

The convergence properties of these different NMF algorithms are discussed in detail by Kim, He and Park [56]. While we focus only on the **MU** algorithms in this paper, we highlight that our algorithm is not restricted to this, and is seamlessly extensible to other NMF algorithms as well, including **HALS**, **ABPP**, Alternating Direction Method of Multipliers (**ADMM**) [102], and Nesterov-based methods [33].

6.1.1 Multiplicative update

As we are considering ℓ_2 regularization on the matrix $\mathbf{W}$ and ℓ_1 regularization on the matrix $\mathbf{H}$ to address the sparsity of the input matrix, the NMF problem becomes

$$\min_{\mathbf{W}\geqslant 0,\mathbf{H}\geqslant 0} \|\mathbf{A} - \mathbf{W}\mathbf{H}\|_F + \alpha(\|\mathbf{W}\|)_F^2 + \beta\sum_{i=1}^{n}\|\mathbf{h}_i\|_1^2. \tag{6.3}$$

The values α and β are fixed for the experiments. In the case of **MU** [90], individual entries of $\mathbf{W}$ and $\mathbf{H}$ are updated with all other entries fixed. In this case, the update rules are

$$\begin{aligned} w_{ij} &\leftarrow w_{ij}\frac{(\mathbf{A}\mathbf{H}^T)_{ij}}{(\mathbf{W}(\mathbf{H}\mathbf{H}^T + 2\beta\mathbf{1}_k))_{ij}}, \text{ and}\\ h_{ij} &\leftarrow h_{ij}\frac{(\mathbf{W}^T\mathbf{A})_{ij}}{((\mathbf{W}^T\mathbf{W} + 2\alpha\mathbf{I}_k)\mathbf{H})_{ij}}.\end{aligned} \tag{6.4}$$

where $\mathbf{1}_k$ is a matrix of $k \times k$ with all one's and $\mathbf{I}_k$ is an identity matrix of size $k \times k$.

After computing the Gram matrices $\mathbf{H}\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{W}$ and adding the appropriate regularizers and the products $\mathbf{A}\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{A}$, the extra cost of computing $\mathbf{W}(\mathbf{H}\mathbf{H}^T + 2\beta\mathbf{1}_k)$ and $(\mathbf{W}^T\mathbf{W} + 2\alpha\mathbf{I}_k)$ is $F(m,n,k) = 2(m+n)k^2$ flops to perform updates for all entries of $\mathbf{W}$ and $\mathbf{H}$, as the other elementwise operations affect only lower-order terms. The details about using AU-NMF in Algorithm 14 for other algorithms **HALS** and **ABPP** are explained in [41, 42]. This update in (6.4) can be easily parallelized using dense matrix kernels.

It is important to observe that the update function of $\mathbf{W}$ is element-wise normalization of the sparse matrix-dense matrix multiplication $\mathbf{A}\mathbf{H}^T$ with the denominator. Given that all the processes

own the $k \times k$ gram of the factor matrix $\mathbf{H}$, computing the entire denominator $(\mathbf{W}(\mathbf{H}\mathbf{H}^T + 2\beta\mathbf{1}_k))$ does not require any communication, hence can be done locally. One can argue that the same holds for updating $\mathbf{H}$ as well. Therefore, for row and column index sets $\mathcal{I}_p$ and $\mathcal{J}_p$, one can update these rows of the factor matrices as follows:

$$\begin{aligned} \mathbf{W}(\mathcal{I}_p, :) &\leftarrow \mathbf{W}(\mathcal{I}_p, :) \circledast (\tilde{\mathbf{W}}(\mathcal{I}_p, :)) \oslash (\mathbf{W}(\mathcal{I}_p, :)(\mathbf{G}_H + 2\beta\mathbf{1}_k)), \text{ and} \\ \mathbf{H}(\mathcal{J}_p, :) &\leftarrow \mathbf{H}(\mathcal{J}_p, :) \circledast (\tilde{\mathbf{H}}(\mathcal{J}_p, :) \oslash (\mathbf{G}_W + 2\alpha\mathbf{I}_k)\mathbf{H}(\mathcal{J}_p, :)). \end{aligned} \tag{6.5}$$

where $\circledast$ and $\oslash$ correspond to element-wise multiplication and division of matrices or vectors. This scheme provides row-wise parallelism in NNLS computation. **HALS** and **ABPP** can similarly be expressed in this row-parallel form.

6.2 Parallel sparse NMF

Here, we first introduce our parallel NMF algorithm that operates on a partition of the matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. For a given partition, we describe how parallel computations and communications take place within the algorithm, and illustrate computational and communication costs associated with a partition. In doing so, we elaborate the advantages of our parallelization scheme compared to the state of the art. We then discuss efficient partitioning strategies to better establish computational load balance and reduce communication.

6.2.1 Distributed parallel sparse NMF algorithm

Parallelizing NMF involves partitioning data and computational tasks to processes, and communicating results wherever necessary, just as any parallel algorithm. Specifically for NMF, one needs to partition the sparse matrix $\mathbf{A}$ as well as the factor matrices $\mathbf{W}$ and $\mathbf{H}$, where the former partitioning distributes the computational load of sparse matrix-dense matrix multiplications $\mathbf{A}\mathbf{H}$ and $\mathbf{W}^T\mathbf{A}$, whereas the latter divides the workload of NNLS computations to processes. We provide the execution of our parallel algorithm for computing a rank-k NMF of a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ in Algorithm 15, which is executed by each process p for $1 \leq p \leq P$. The algorithm starts with an arbitrary partition of the input matrix and the factor matrices; process p owns the submatrices $\mathbf{W}(\mathcal{I}_p, :)$ and $\mathbf{H}(:, \mathcal{J}_p)$ as well as the nonzero elements of the sparse matrix $\mathbf{A}_p$ where $\mathbf{A} = \bigcup_{i=1}^{P} \mathbf{A}_i$, i.e., $\mathbf{A}_1, \ldots, \mathbf{A}_p$ partitions the nonzeros of $\mathbf{A}$. The sets $\mathcal{F}_p$ and $\mathcal{G}_p$ denote the "footprints" of the process p on the rows and columns of matrices $\mathbf{W}$ and $\mathbf{H}$, respectively, in the course of the algorithm. That is, we have $i \in \mathcal{F}_p$ or $j \in \mathcal{G}_p$ if only if $i \in \mathcal{I}_p$ or $j \in \mathcal{J}_p$ (row/column is owned), or there is a nonzero element $a_{i,j} \in \mathbf{A}_p$ (row/column is used in local computations). At each iteration, the process p is responsible for gathering the new value of submatrices $\mathbf{W}(\mathcal{I}_p, :)$ and $\mathbf{H}(:, \mathcal{J}_p)$, and sending these values to processes in need.

In an iteration of Algorithm 15, each process p possesses three types of computational tasks as well as associated pre- and post-communication steps. The first task involves performing sparse matrix-dense matrix multiplications $\mathbf{A}_p\mathbf{H}(:, \mathcal{G}_p)$ and $\mathbf{W}(\mathcal{F}_p, :)^T\mathbf{A}_p$, whose results are stored in distributed matrices $\tilde{\mathbf{W}}$ and $\tilde{\mathbf{H}}$, which follow the same row-/column-wise data distribution as $\mathbf{W}$ and $\mathbf{H}$. Note that carrying out these multiplications must be preceded by a communication step where each process

Algorithm 15 DIST-SPNMF: Distributed memory Sparse NMF algorithm

Input: $\mathbf{A}_p$: An $m \times n$ sparse matrix
$\mathcal{I}_p$,$\mathcal{J}_p$: Set of rows/columns of $\mathbf{W}$ and $\mathbf{H}$ owned by process p
$\mathcal{F}_p$,$\mathcal{G}_p$: Footprints of process p on $\mathbf{W}$ and $\mathbf{H}$
$\mathbf{W}(\mathcal{I}_p,:)$,$\mathbf{H}(:,\mathcal{J}_p)$: Owned rows and columns of initial $\mathbf{W}$ and $\mathbf{H}$
k: The NMF rank
Output: Process p gets the final value of $\mathbf{W}(\mathcal{I}_p,:)$ and $\mathbf{H}(:,\mathcal{J}_p)$

1: **repeat**
2: COMM-EXPAND($\mathbf{H}(:,\mathcal{G}_p)$)
3: $\tilde{\mathbf{W}}(\mathcal{F}_p,:) \leftarrow \mathbf{A}_p\mathbf{H}(:,\mathcal{G}_p)$
4: COMM-FOLD($\tilde{\mathbf{W}}(\mathcal{F}_p,:)$)
5: $\mathbf{G}_H \leftarrow$ ALL-REDUCE($\mathbf{H}(:,\mathcal{J}_p)\mathbf{H}(:,\mathcal{J}_p)^T$)
6: $\mathbf{W}(\mathcal{I}_p,:) \leftarrow$ NNLS($\mathbf{G}_H, \tilde{\mathbf{W}}(\mathcal{I}_p,:)$)
7: COMM-EXPAND($\mathbf{W}(\mathcal{F}_p,:)$)
8: $\tilde{\mathbf{H}}(:,\mathcal{G}_p) \leftarrow \mathbf{W}(\mathcal{F}_p,:)^T\mathbf{A}_p$
9: COMM-FOLD($\tilde{\mathbf{H}}(:,\mathcal{G}_p)$)
10: $\mathbf{G}_W \leftarrow$ ALL-REDUCE($\mathbf{W}(\mathcal{I}_p,:)^T\mathbf{W}(\mathcal{I}_p,:)$)
11: $\mathbf{H}(\mathcal{J}_p,:) \leftarrow$ NNLS($\mathbf{G}_W, \tilde{\mathbf{H}}(\mathcal{J}_p,:)$)
12: **until** convergence or maximum number of iterations

p gets the submatrices $\mathbf{H}(:,\mathcal{G}_p \setminus \mathcal{J}_p)$ and $\mathbf{W}(\mathcal{F}_p \setminus \mathcal{I}_p,:)$ that are accessed by entries of $\mathbf{A}_p$, and this step is performed at Lines 2 and 7. These multiplications performed by each process p generate partial results for the set $\mathcal{F}_p$ and $\mathcal{G}_p$ of the rows of $\tilde{\mathbf{W}}$ and the columns of $\tilde{\mathbf{H}}$, respectively, which is highlighted at Lines 3 and 8. Indeed, partial results for the submatrices $\tilde{\mathbf{W}}(\mathcal{F}_p \setminus \mathcal{I}_p,:)$ and $\tilde{\mathbf{H}}(:,\mathcal{G}_p \setminus \mathcal{J}_p)$ correspond to rows and columns owned by other processes; hence, they need to be communicated. The results for $\tilde{\mathbf{W}}(\mathcal{I}_p,:)$ and $\tilde{\mathbf{H}}(:,\mathcal{J}_p)$, however, should be kept locally, and all partial results for these matrix rows and columns generated by other processes should similarly be received and accumulated in order to obtain the final value for these owned portions. The second task is to compute the Gram matrices $\mathbf{G}_H = \mathbf{H}\mathbf{H}^T$ and $\mathbf{G}_W = \mathbf{W}^T\mathbf{W}$ of size $k \times k$, and to make these matrices available to all processes, which is performed at Lines 5 and 10. This is done in a row-parallel dense matrix multiplication step, in which the process p computes $\mathbf{H}(:,\mathcal{J}_p)\mathbf{H}^T(:,\mathcal{J}_p)$ and $\mathbf{W}^T(\mathcal{I}_p,:)\mathbf{W}(\mathcal{I}_p,:)$, followed by an ALL-REDUCE communication of these partial multiplications. The third task pertains to updating the factor matrices $\mathbf{W}$ and $\mathbf{H}$ using matrices $\tilde{\mathbf{W}}$ and $\mathbf{G}_H$, or $\tilde{\mathbf{H}}$ and $\mathbf{G}_W$, which takes place at Lines 6 and 11. This corresponds to Lines 3 and 4 of Algorithm 14, and can be computed locally at each process p by executing NNLS algorithm on dense matrices $\tilde{\mathbf{W}}(\mathcal{I}_p,:)$ and $\mathbf{G}_H$, or $\tilde{\mathbf{H}}(:,\mathcal{J}_p)$ and $\mathbf{G}_W$, to obtain new $\mathbf{W}(\mathcal{I}_p,:)$ or $\mathbf{H}(\mathcal{I}_p,:)$, respectively, as described in Section 6.1.1.

The first type of communication in Algorithm 15 pertains to an ALL-REDUCE of a dense matrix of fixed size $k \times k$ at Lines 5 and 10, and the cost of this step is typically negligible in compare to the rest. The other two communication types involve (i) transferring the partial row results of $\tilde{\mathbf{W}}$ and $\tilde{\mathbf{H}}$ to their owner processes at Lines 4 and 9 to accumulate at the owners, (ii) sending the updated rows of $\mathbf{W}$ and $\mathbf{H}$ to processes in need at Lines 2 and 7. We respectively call these steps *fold* and *expand* communications, following the convention used by the sparse matrix community. The way these two communications are carried out plays a vital role in obtaining parallel scalability as they dominate the communication cost of the algorithm. Here, we employ an efficient point-to-point communication framework to perform fold and expand communications. In this scheme, after determining a partition

of the matrix, each process p determines the sets $\mathcal{F}_p$ and $\mathcal{G}_p$ using the row and column indices in its local matrix $\mathbf{A}_p$ and its sets $\mathcal{I}_p$ and $\mathcal{J}_p$ of owned row indices, then identifies the owners of these rows in order to request these rows from the corresponding processes in the expand step. Next, these requests are exchanged in a communication step so that each process p determines the unique row indices that it needs to send to or receive from each other process in expand and fold steps. Once this communication structure is formed, it is reused throughout all iterations in Algorithm 15. This is possible thanks to the data partition staying fixed during the algorithm, hence data dependencies do not change.

The existing work in the literature for parallel sparse NMF [41, 42] amounts to employing a collective communication strategy for both expand and fold steps of Algorithm 15 for dense as well as sparse $\mathbf{A}$. In this strategy, the rows and columns indices $1, \ldots, m$ and $1, \ldots, n$ are divided into P_r and P_c ($P = P_r P_c$) sets $I_1, \ldots, I_{P_r}$ and $J_1, \ldots, J_{P_c}$ of equal size and having contiguous indices. This effectively defines a 2D uniform checkerboard partition of $\mathbf{A}$ with matrix blocks $\mathbf{A}_{(r,c)} = \mathbf{A}(I_r, J_c)$ for all $1 \leq r \leq P_r$ and $1 \leq c \leq P_c$, as well as a row partition of $\mathbf{W}$ and $\mathbf{H}$. Here, process p owns the matrix subblock $\mathbf{A}_{(r,c)}$, where $r = \lfloor P/P_c \rfloor + 1$ and $c = (P \mod P_c) + 1$, as well as m/P and n/P rows of $\mathbf{W}(I_r, :)$ and $\mathbf{H}(J_c, :)$. The communication is performed within each process row and column using ALL-GATHER and REDUCE-SCATTER routines, in which the process p receives all matrix rows $\mathbf{W}(I_r, :)$ and $\mathbf{H}(I_c, :)$ belonging to processes in the same row and column of the process grid. Despite being favorable due to small number of exchanged messages in collective routines, in this strategy processes receive many rows that they do not need in their local sparse matrix dense matrix multiplication, and this redundancy dramatically increases the communication volume, thus preventing scalability. To illustrate, in Figure 6.1 we provide a sparse matrix partitioned using a 5×5 checkerboard scheme where each process owns a subblock of the matrix, and show only the nonzero elements belonging to three columns with indices j_1, j_2, and j_3. Here, in performing the expand communication of Line 2, the collective communication strategy sends these column vectors to all processes in the same column, whereas point-to-point communication identifies the exact set of processes in need beforehand, indicated with the same color, and communicates the data only with these processes. In this example, the collective communication scheme incurs 4, 3, and 2 units of redundant communication for columns j_3, j_2, and j_1, respectively, and this redundancy only worsens with the increasing number of processes.

6.2.2 Partitioning

Algorithm 15 requires a partitioning of the nonzeros of $\mathbf{A}$ as well as the rows and columns of $\mathbf{W}$ and $\mathbf{H}$, and these three partitions completely determine its computational and communication costs. Here, we compare different partitioning strategies, and argue how they relate to these two performance metrics.

Our algorithm is fine-grained; meaning that it can work any fine-grain, 1D, or 2D/checkerboard partition of the sparse matrix $\mathbf{A}$. Here, however, we exclusively focus on $P_r \times P_c$ (where $P = P_r P_c$) checkerboard partitionings, and always set P_c to the number of cores available per compute node in the cluster. This helps us reduce the communication cost in two ways. The first one corresponds to that we do not employ shared-memory parallelism in our implementation, hence assign one MPI rank per core. In this scenario, by assigning each group of P_c processes owning the same row block of $\mathbf{A}$ to the same node, and similarly assigning the corresponding row blocks of $\mathbf{W}$ and $\tilde{\mathbf{W}}$ to these processes, we effectively restrain the communication to stay within the same node in the row dimen-

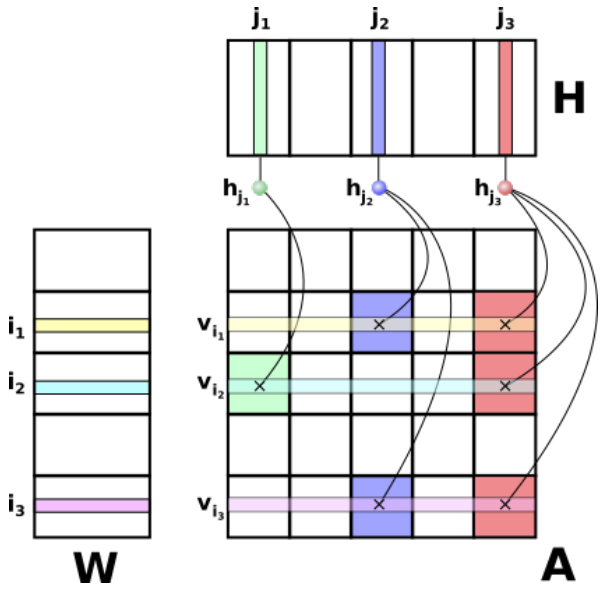


Figure 6.1: A 5x5 checkerboard partition of a sparse matrix.

sion of the process grid (which pertains to communicating matrices $\mathbf{W}$ and $\tilde{\mathbf{W}}$), whose cost become negligible (as it is performed by memory copies). This enables us to focus solely on reducing the communication due to columns of $\mathbf{H}$ and $\tilde{\mathbf{H}}$. Second, this topology bounds the maximum number of messages sent per process by $P_r - 1$ in the column dimension, thereby significantly reduces the communication latency by a factor of P_c in compare to a fine-grain or a 1D partition, which incur up to $P - 1$ messages per process.

Partitioning sparse input matrix $\mathbf{A}$

Hypergraph partitioning (P2PHP) We aim to partition the matrix $\mathbf{A}$ into P_r row slices first, and P_c column slices next, to obtain an $P_r \times P_c$ checkerboard partition. To this end, we adopt hypergraph partitioning techniques used in parallel sparse matrix literature [19], and modify them appropriately to match the requirements of sparse NMF computations.

The typical checkerboard hypergraph partitioning approach used for parallel sparse matrix-vector multiplication (SpMV) proceeds in two steps as in what follows. First, a *column-net* hypergraph model is formed; for each row i and column j of $\mathbf{A}$, a vertex v_i and a hyperedge h_j are added to the hypergraph, and for each nonzero $(i, j) \in \mathbf{A}$ v_i is connected to h_j. Note that in the sparse NMF case, this connection also models the dependency to $\mathbf{H}(:, j)$ in computing $\tilde{\mathbf{W}}(i, :)$ at Line 3. Next, the weight $|\mathbf{A}(i, :)|$ is assigned to each vertex v_i, which represents the computational load for the SpMV. In partitioning this hypergraph into P_r parts, minimizing the cutsize of the hypergraph corresponds to minimizing the total communication volume due to column vector entries, while establishing the balance for vertex weights corresponds to balancing the computational load of SpMV. After obtaining the row partition this way, a second *multi-constraint* hypergraph partitioning step is performed on the *row-net* hypergraph model; for each column j and row i of $\mathbf{A}$, a vertex v_j and a hyperedge h_i are added to the hypergraph, and for each nonzero $(i, j) \in \mathbf{A}$, v_j is connected to h_i. Similarly, this connection models the dependency to $\mathbf{W}(i,)$ in computing $\tilde{\mathbf{H}}(:, j)$ at Line 8. This time, each vertex v_j

is assigned P_r weights, each corresponding to the number of nonzeros corresponding to different row partitions in the jth column of $\mathbf{A}$. Partitioning this hypergraph into P_c parts finally defines a $P_r \times P_c$ checkerboard topology. Minimizing the cutsize of this hypergraph corresponds to minimizing the total row communication volume, whereas balancing vertex weights corresponds to balancing the number of nonzeros of $\mathbf{A}$ assigned to each part (SpMV load).

In the context of parallel sparse NMF, we modify both row and column partitioning steps to better grasp its computational and communication requirements. Note that in the first step, the traditional scheme balances the number of nonzeros per row slice, which in turn balances the workload for multiplying $\mathbf{A}$ with dense matrices $\mathbf{W}$ and $\tilde{\mathbf{H}}$. However, this can still yield a significant imbalance in the number of rows assigned to each row slice. In the SpMV case, this incurs an imbalance in the number of vector entries assigned to each row slice, which does not cause a significant problem as the cost of vector operations is typically negligible in an iterative solver. In the NMF case, however, this causes a computational imbalance in the NNLS step involving dense matrices $\mathbf{W}$ and $\tilde{\mathbf{W}}$, which is a major computational step with a non-negligible cost, particularly when the rank of approximation gets higher. To achieve load balance in the number of rows per row slice, one can similarly employ multi-constraint partitioning in this step as well, by assigning a second fixed weight of 1 to each vertex, then balancing both costs in the partitioning. This formulation is expected to achieve balance in both SpMM and NNLS steps. In practice, however, multi-constraint hypergraph partitioning is known to not work too good in practice; therefore, we aim to achieve the same effect by assigning a single weight $(\text{nnz}(\mathbf{A})/m + \text{nnz}(\mathbf{A}(i,:))$ to each vertex v_i. This weighting is preferred to find a tradeoff between balancing the number of vertices per row slice, which corresponds to the cost of dense matrix operations, and balancing the number of nonzeros per part, which corresponds to the cost of SpMM. We establish these two load balance goals by assigning a fixed cost to each vertex in the first summand, and increasing the cost of the vertices corresponding to denser rows in the second summand. In Figure 6.1 we show this hypergraph construction for a 3×3 subset of the matrix $\mathbf{A}$, in which the rows corresponding to each vertex are highlighted, and the vertices are connected to related hyperedges. The vertex weights are omitted in the figure for simplicity.

In the second partitioning step, the traditional scheme balances the number of nonzeros per part, which similarly balances the workload for multiplying $\mathbf{A}$ with dense matrices $\mathbf{W}$ and $\tilde{\mathbf{H}}$, while minimizing the total communication volume due to vector rows. This approach, however, is not ideal for the sparse NMF for multiple reasons. First, while balancing the sparse matrix multiplication load, it similarly disregards the NNLS work proportional to the number of columns assigned per column slice. Second, it employs multi-constraint partitioning with P_r constraints which can be quite high, in which case multi-constraint partitioners are known to yield poor load balance even for the SpMM step. Finally, it tries to minimize the total communication volume due to rows, whose cost is expected to be negligible as it is restrained to stay within a compute node. The important communication objective here is *balancing* the column communication volume per column slice, which is neglected in this approach.

In overall, in the second partitioning phase we can discuss $2P_r + 1$ weights corresponding to the computation and communication induced by each column: one fixed weight of value 1 to balance the number of columns per column slice for NNLS; P_r weights for the number of nonzeros in the column correspond to each row slice; P_r weights for the volume communication incurred to each row slice by the column. Our objective here is to partition these columns in a way that all $2P_r + 1$ weights are independently balanced. This is essentially a knapsack problem with multiple weights, which is NP-hard. One can employ a specific heuristic for this partitioning; however, the computatational and

memory requirements might prevent this approach from being practical for a high n and P_r value. We instead partition the columns randomly, which is expected to give acceptably good balance for all these weights in practice. Indeed, this approach disregards the row communication volume; yet this cost is expected to be unimportant.

Randomized checkerboard partitioning(P2PRP) This scheme corresponds to partitioning both the rows and columns of $\mathbf{A}$ into R segments randomly. It is expected to provide good load balance both in sparse and dense matrix operations, but it overlooks the communication cost.

Uniform checkerboard partitioning(FAUN) This partitioning variant forms an $R \times C$ partition of $\mathbf{A}$ by putting a contiguous set of m/R and n/C rows and columns in each slice. $\mathbf{W}$ and $\mathbf{H}$ are partitioned conformally with this topology; each process is assigned a contiguous set of m/P and n/P rows and columns, respectively, belonging to the corresponding row and column blocks of $\mathbf{H}$. This is the only partitioning scheme employed by MPIFAUN [41, 42], hence we only use it for this implementation. We also use a randomized variant (**FAUNRP**) of this scheme in which the rows and columns of $\mathbf{A}$ are permuted randomly before executing MPIFAUN to better balance its nonzeros.

Partitioning factors W and H

Once $\mathbf{A}$ is partitioned with an $R \times C$ checkerboard topology, one has to partition the rows and columns of factor matrices conformally with this topology to form the sets $\mathcal{I}_p$ and $\mathcal{J}_p$ in Algorithm 15. In doing so, we are interested in assigning rows and columns to processes in the corresponding process row or column equitably, as this balances NNLS work as well as the memory cost for storing a part of $\mathbf{W}$ and $\mathbf{H}$. For this purpose, we specify imbalance parameters α_r and α_c that correspond to the maximum imbalance we allow in this partitioning; i.e., $|\mathcal{I}_p| \leq \alpha_r m/P$ and $|\mathcal{J}_p| \leq \alpha_c n/P$ for each process p. We normally set these imbalance parameters to 1.1. However, when $m >> n$, we increase α_c proportionally in order to give more flexibility in column partitioning which results in smaller communication volume.

Next, for each row and column of $\mathbf{W}$ and $\mathbf{H}$, we create a list of processes that have a dependency to that row or column, which corresponds to processes owning the matrix blocks of the same color in Figure 6.1. Then, we are to assign each row/column to, preferably, one of these dependent parts. Ideally, we would like to avoid a communication imbalance due to a significantly higher send or receive volume in either of fold and expand communications. We argue that randomly assigning a row or column to one of its dependent parts establishes this communication balance for the following reason. Consider any part dependent to column j, and let h_j be the hyperedge connecting the parts for this column. In the random assignment, this part owns column j with a probability of $1/\lambda(h_j)$, which in turn incurs a receive/send volume of $\lambda(h_j) - 1$ in the fold/expand communication. With $(\lambda(h_j)-1)/\lambda(h_j)$ probability, the part does not own the column j, which yields a send/receive volume of 1 in the fold/expand communication. In this case, both the expected send and receive volumes of fold and expand communications due to column j are $(\lambda(h_j)-1)/\lambda(h_j)$; therefore, this assignment is expected to balance the overall send and receive volumes of all parts in both communication steps. To satisfy the balance constraints, however, we do this random assignment only to one of the dependent parts satisfying the imbalance constraint. If all processes in the dependence list are overloaded, we randomly assign it to a process that is not overloaded in the same process row or column. Note that the

latter assignment increases the communication volume due to that row or column by 1; hence, larger imbalance parameters yield smaller communication volume by reducing this type of assignment in general.

6.3 Related work

In the data mining and machine learning literature, there is an overlap between low rank approximations and matrix factorizations due to the nature of applications. Despite its name, non-negative matrix "factorization" is in fact a low rank approximation. Recently, there has been a growing interest in collaborative filtering based recommender systems. One of the popular techniques for collaborative filtering is matrix factorization, often with nonnegativity constraints, and its implementation is widely available in many off-the-shelf distributed machine learning libraries such as GraphLab [72], MLLib [75], and many others [89, 112]. However, we would like to emphasize that collaborative filtering using matrix factorization is a different problem than NMF: In the case of collaborative filtering, nonzeros in the matrix are considered to be observed ratings, and zeros are treated as missing entries, while in the case of NMF, there is no missing entries and all zeros are considered as observed entries.

There are several recent distributed NMF algorithms in the literature [27, 70, 71, 111]. Liu et al. propose running Multiplicative Update (MU) for KL divergence, squared loss, and "exponential" loss functions [71]. Matrix multiplication, element-wise multiplication, and element-wise division are the building blocks of the MU algorithm. The authors discuss performing these matrix operations efficiently on Hadoop for sparse matrices. Using similar approaches, Liao et al. implement an open source Hadoop-based MU algorithm and study its scalability on large-scale biological data sets [70]. Also, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [111]. Similarly, Faloutsos et al. propose a distributed, scalable method for decomposing matrices, tensors, and coupled data sets through stochastic gradient descent on a variety of objective functions [27]. The authors also provide an implementation that can enforce non-negative constraints on the factor matrices. All of these works use Hadoop framework to implement their algorithms, hence are not very efficient.

Spark [113] is a popular big-data processing infrastructure that is generally more efficient for iterative algorithms such as NMF than Hadoop, as it maintains data in memory and avoids file system I/O. Even with a Spark implementation of previously proposed Hadoop-based NMF algorithms, the performance still suffers from expensive communication of input matrix entries, and Spark does not have innate mechanisms to overcome this shortcoming. Spark has collaborative filtering libraries such as MLlib [75] which use matrix factorization and can impose non-negativity constraints.

In parallel with the Hadoop and Spark implementations, there has been a growing interest in the HPC community towards efficiently computing these algorithms with tuned high performance implementations. Kannan, Ballard and Park [41, 42] proposed MPIFAUN framework to implement various NMF algorithms such as multiplicative update (MU), Hierarchical Alternating Least Squares (HALS), and Alternating Non-negative Least Squares using Block Principal Pivoting (ANLS-BPP). We choose this work as a baseline, as it is the only available high performance implementation of NMF, and it performs significantly faster than Hadoop and Spark-based approaches. To elaborate this, Gittens et.al. [28] recently benchmarked the implementations of different matrix factorization algorithms (such as NMF and Principal Component Analysis (PCA)) in Spark and in MPI/C. They

claim that native MPI implementations on HPC platforms outperform Spark implementation by a speedup factor of 44x. Similar observations have been made by Sukumar, Kannan, Matheson and Lim [100, 101] on supercomputers at Oak Ridge Leadership Computing Facility. Finally, there are implementations of the MU algorithm in a distributed memory setting using X10 [32], and on a GPU [74].

6.4 Experiments

In this section, we compare our algorithm DIST-SPNMF against MPIFAUN on two big sparse matrices formed from real world datasets. We analyze and compare the computation and the communication timings of these algorithms on a smaller cluster, then test the scalability limits of our method on a larger supercomputing environment.

6.4.1 Experimental setup

Datasets

We use two matrices formed from Flickr.com and Delicious.com that involve images tagged with different labels by users. These are essentially matrices obtained from Delicious and Flickr tensors described in Section 2.6 by discarding their first and second dimensions. Flickr and Delicious matrices are of size 28Mx1.6M and 17Mx2.5M, and have 112M and 72M nonzero elements, respectively. The current implementation of MPIFAUN can only operate when R and C can divide m and n, hence we trimmed the matrices slightly.

Parallel platform

We conducted our experiments on two different parallel computing platforms. The first platform is the "Rhea" cluster at the Oak Ridge Leadership Computing Facility (OLCF), which is a commodity-type Linux cluster with a total of 512 nodes and a 4X FDR Infiniband interconnect. Each node contains dual-socket 8-core Intel Sandy Bridge-EP processors operating at 2GHz clock frequency, and 128 GB of memory. Each socket has a shared 20MB L3 cache, and each core has a private 256K L2 cache. There, we ran our experiments up to 3072 cores, which is the maximum allowed in the cluster. The second platform is **Blue Gene/Q** described in Section 2.7. We ran both algorithms using 16 MPI ranks per node, and set $P_c = 16$ in all partitionings.

Our code for local matrix operations is developed using the matrix library Armadillo [88]. We use BLAS and LAPACK for dense matrix operations by linking Armadillo with Intel MKL, OpenBLAS [109], or any other BLAS and LAPACK implementation. Both codes are compiled using the default GNU C++ Compiler (g++ (GCC) 5.3.0) and MPI library (Open MPI 1.8.4) on RHEA, and Clang compiler (3.5.0) with IBM MPI library on Blue Gene/Q.

Algorithms

In our experiments, we considered the following algorithms and partitionings:

- **FAUN**: MPIFAUN algorithm [41, 42] with uniform natural partitioning (**FAUN**) where each process holds an input matrix of size $m/P_r \times n/P_c$.

- **FAUNRP**: The partitioning strategy in **FAUN** could result in a significant computational load imbalance with a skewed nonzero distribution of $\mathbf{A}$. We alleviate this by randomly permuting the rows and columns of the matrix before executing MPIFAUN, and call this scheme **FAUNRP**.

- **P2PHP**: DIST-SPNMF (Algorithm 15) with hypergraph checkerboard partitioning explained in Section 6.2.2.

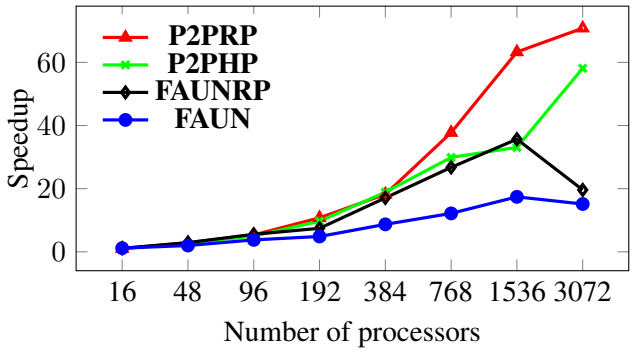
- **P2PRP**: DIST-SPNMF (Algorithm 15) with randomized checkerboard partioning explained in Section 6.2.2.

6.4.2 Strong scaling

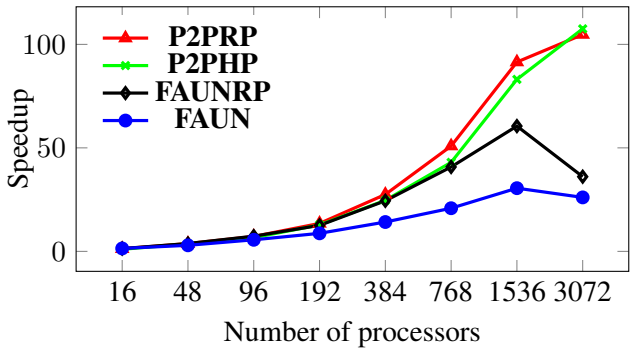
In Figures 6.2a and 6.2b, we show the speedup results of all four instances on the Rhea cluster using up to 3072 MPI ranks/cores on Flickr data. The speedup values are with respect to the slowest running time among all four instances using 16 cores (single node). We observe in Figures 6.2a and 6.2b that all algorithms scale up to 1536 cores, yet MPIFAUN instances achieve this with significantly lower parallel efficiency. This mostly is due to higher communication costs involved in the collective communication scheme for both instances. We also realize that **FAUNRP** significantly improves the runtime with respect to **FAUN**, meaning that **FAUN** indeed causes load imbalance in partitioning the nonzeros of $\mathbf{A}$. At 3072 processes, both **FAUNRP** and **FAUN** lose scalability and slow down, whereas **P2PHP** and **P2PRP** scale to 3072 processors. One point to note is that for $k = 16$, **P2PHP** is considerably slower than **P2PRP**, whereas for $k = 48$, **P2PHP** starts to get faster than **P2PRP**. The former result is due to that **P2PHP** typically causes more imbalance in factor matrix rows and the input matrix nonzeros, whereas in the latter result, the size of each communication unit is tripled; therefore, the reduction in the communication cost using **P2PHP** begins to compensate for the performance loss due to load imbalance.

In Figures 6.3a and 6.3b, we provide the same speedup results for Delicious matrix. We observe a similar trend in the comparisons of different methods, except that **FAUNRP** and **FAUN** scale even worse in this case. Our algorithm also loses scalability after 1536 processes, and similarly to the previous case **P2PHP** starts slower than **P2PRP** due to load imbalance, and catches up for $k = 48$. These two test cases clearly show that employing a point-to-point communication with effective nonzero and matrix row partitionings is essential for obtaining high performance in the NMF algorithm.

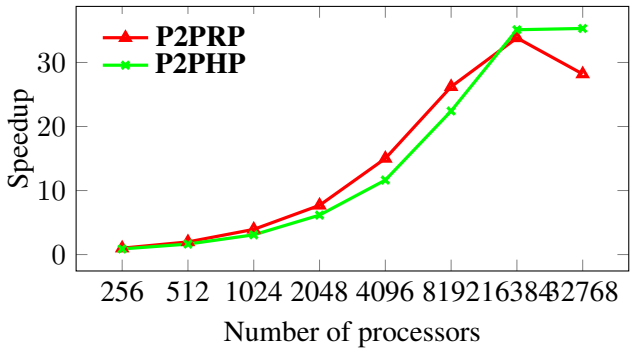
To better test the scalability limits of our algorithms, we ran them on an IBM Blue Gene/Q supercomputer up to 32768 processors using the same two matrices. The results of these two experiments are provided in Figures 6.2c and 6.3c. Our algorithm graciously scales up to 16384 cores in all four instances, and **P2PHP** manages to slightly improve the runtime using 32768 cores on Flickr, while other three instances witness a slowdown using 32768 cores. Again, **P2PHP** is slower than **P2PRP** using lower number of processors as the communication cost is negligible in these instances, and **P2PHP** introduces worse load balance than **P2PRP**. However, using 32768 processors **P2PHP** manages to outrun **P2PRP** by incurring less communication.



(a) Flickr matrix with $k = 16$ on Rhea.

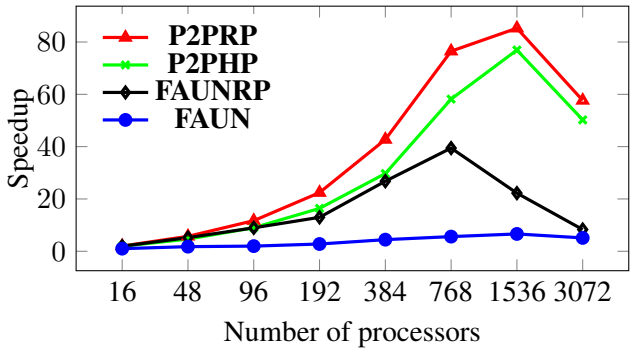


(b) Flickr matrix with $k = 48$ on Rhea.

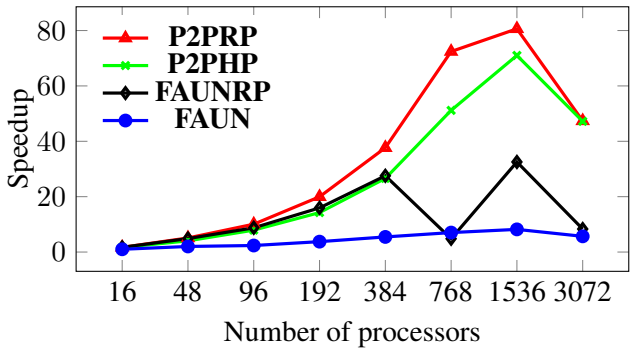


(c) Flickr matrix with $k = 48$ on Blue Gene/Q.

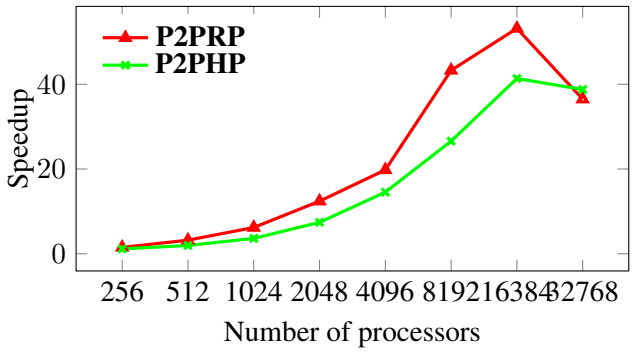
Figure 6.2: Strong scaling on Flickr matrix.



(a) Delicious matrix with $k = 16$ on Rhea.



(b) Delicious matrix with $k = 48$ on Rhea.



(c) Delicious matrix with $k = 48$ on Blue Gene/Q.

Figure 6.3: Strong scaling on Delicious matrix.

6.4.3 Runtime dissection per iteration

In this section, we provide the time spent on each individual operation type and communication within an NMF iteration. We report the averages over 30 iterations on Rhea, and 10 iterations on Blue Gene/Q for each run. As provided in Algorithm 15, there are three types computations and two types of communications within an NMF iteration, and we present timings for these steps with the following labels:

- **Gram**: Computing local contribution to Gram matrix, and performing an ALL-REDUCE to gather the final result.
- **MM**: Computing SpMM using $\mathbf{A}_p$ and one of the factor matrices.
- **LUC** : Local NNLS computation to obtain the new factor matrix.
- **Comm**: Total expand and fold communication time in the case of **P2PRP** and **P2PHP**, and the total time spent on ALL-GATHER and REDUCE-SCATTER steps for **FAUN** and **FAUNRP**.

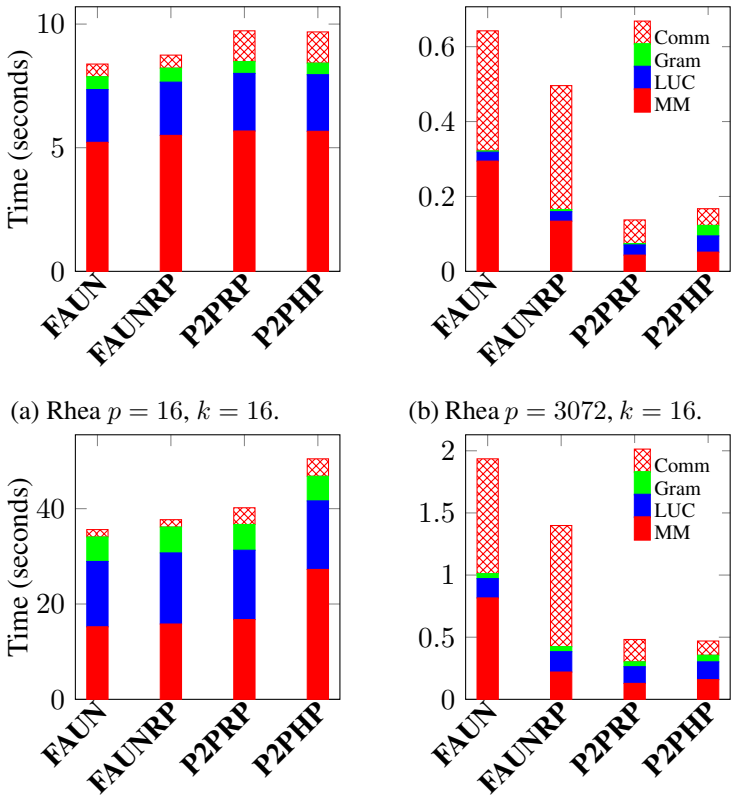
In our results, we do not distinguish the costs of these tasks for $\mathbf{W}$ and $\mathbf{H}$ separately; we instead report their sum.

We report the runtime dissection for Flickr and Delicious matrices in Figure 6.4, Figure 6.5 for Rhea and Figure 6.6 for Blue Gene/Q. For each cluster and data set, we show the timings for the smallest and the largest number of processors used. Our objective in this experiment is to better analyze the speedup results and by comparing the computational and communication costs of different communication schemes and partitionings.

Flickr on Rhea: We observe in Figure 6.4 that in one node configuration with $p = 16$, the **FAUN** and **FAUNRP** performs similar to **P2PRP** and **P2PHP** in terms of computation, and the communication time takes a small portion of the execution in all instances. As the number of processes increases to 3072, the communication time of **P2PRP** and **P2PHP** stays reasonably low, whereas in the case of **FAUN** and **FAUNRP**, we clearly observe that the communication cost dominates the execution time using both low rank values $k = 16$ and $k = 48$. Randomization offers load balance to **FAUNRP** which gives it a slight edge over **FAUN**, yet both instances suffer from the high communication cost associated with the collective communication strategy, which explains the drop in the scalability results.

Delicious on Rhea: In Figure 6.5, we observe that **P2PRP** and **P2PHP** perform better than **FAUN** even in single node configuration. Figure 6.5 shows that **FAUN** takes twice more time than **P2PRP** and **P2PHP** in the sparse matrix multiplication step. This highlights the skewed distribution of the matrix nonzeros, which is alleviated to a certain extend by randomly permuting the matrix. Similar to Flickr data, using 3072 processors, **P2PRP** and **P2PHP** perform significantly better than **FAUN** and **FAUNRP**, whose iteration times are dominated by communication cost.

Flickr and Delicious on Blue Gene/Q: In Figure 6.6, we give the timings for computation and communication steps using our methods with two different partitionings of matrices on Blue Gene/Q. We observe that using 512 processors, communication cost is negligible, and **P2PRP** beats **P2PHP**



(a) Rhea $p = 16$, $k = 16$.

(b) Rhea $p = 3072$, $k = 16$.

(c) Rhea $p = 16$, $k = 48$.

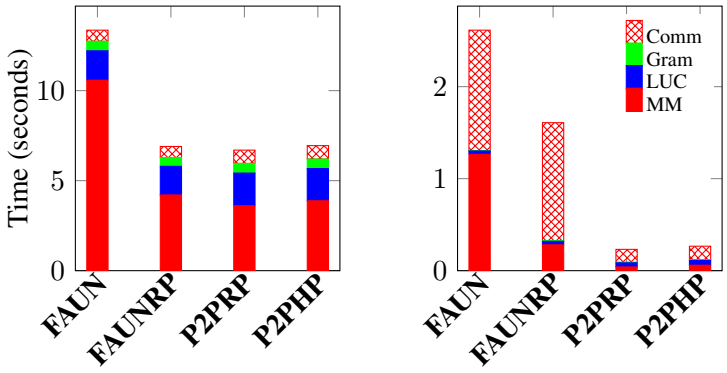
(d) Rhea $p = 3072$, $k = 48$.

Figure 6.4: Flickr matrix runtime dissection for $k = 16$ and $k = 48$ on Rhea.

thanks to better load balance. Using 16384 processors, however, **P2PHP** gets faster than **P2PRP** on Flickr matrix due to significant reduction in the communication volume. On Delicious matrix, **P2PHP** similarly better reduces the communication, yet this is outweighed by the load imbalance in matrix multiplications.

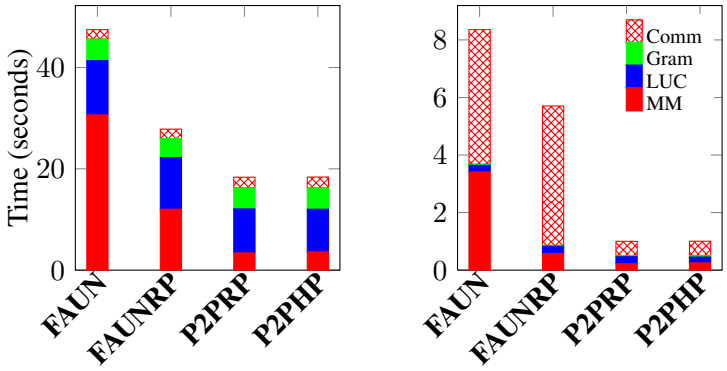
6.5 Conclusion

In this chapter, we proposed a novel algorithm parallel non-negative matrix factorization of sparse matrices in distributed memory environments. To the best of our knowledge, this is the first high performance implementation that takes the sparsity of the input matrix into consideration in communication to reduce the communication cost, and employs effective partitionings to further enhance parallel scalability. We introduce effective partitioning strategies for both the sparse input matrix $\mathbf{A}$ as well as factor matrices $\mathbf{W}$ and $\mathbf{H}$ that helps us achieve good computational load balance while minimizing the communication costs. With all the algorithmic contributions and an efficient parallel implementation, our method outperforms the state of the art by a significant margin, and gracefully scales up to 32768 cores on an IBM Blue Gene/Q supercomputer. Our immediate next steps for extending our work involve adding shared memory parallelism to obtain further speedup.



(a) Rhea $p = 16$, $k = 16$.

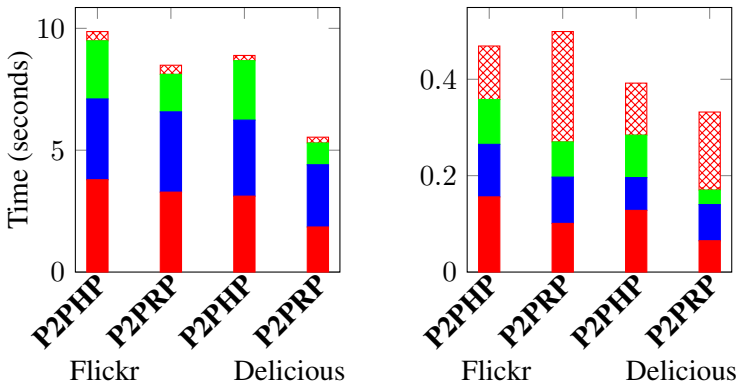
(b) Rhea $p = 3072$, $k = 16$.



(c) Rhea $p = 16$, $k = 48$.

(d) Rhea $p = 3072$, $k = 48$.

Figure 6.5: Delicious matrix runtime dissection for $k = 16$ and $k = 48$ on Rhea.



(a) BG/Q $p = 512$.

(b) BG/Q $p = 16384$.

Figure 6.6: Runtime dissection of Flickr and Delicious for $k = 48$ on Blue Gene/Q. The left two bars are for the Flickr matrix, and the right two bars are for the Delicious matrix.

Part II

Dense Tensor Decompositions

Chapter 7

On Computing Dense Tensor Decompositions Using Dimension Trees

As already mentioned in the introduction, tensors have been increasingly used in many application domains in the recent past, and in particular, methods for dense tensors have proven to be among the most powerful tools in many signal processing applications [77, 78, 91]. Among these applications are analyzing speech signals for source separation [77], finding the localization of the signal source from radar signals [78], and other communication applications [91]. The computational core of CP and Tucker decomposition algorithms involve special operations called the matricized tensor-times Khatri-Rao product (MTTKRP), and tensor-times-matrix multiplication (TTM). Performing these operations on high dimensional big tensor datasets gets computationally expensive in these applications; hence, efficiently carrying out these steps is indispensable to be able to address the computational needs of such applications. Our aim in this chapter is to investigate a fast computation of MTTKRP and TTM steps in CP-ALS and HOOI algorithms for dense tensors using the dimension tree-based computational scheme we introduced in Chapter 3 to achieve this performance goal.

The contributions in this chapter are as follows. We investigate an efficient computation of MTTKRP and TTM operations performed in a repetitive manner, as performed by CP-ALS and HOOI algorithms and others variants for computing CP and Tucker decompositions. To achieve this, we introduce a novel computational scheme based on the dimension tree data structure we introduced in Section 2.4, and significantly reduce the computational costs of these operations by reusing common partial results with the help of the tree structure. We discuss optimal dimension tree structures minimizing the computational cost in both these algorithms, and show that finding optimal dimension trees is NP-hard in both cases. Finally, we introduce an optimal greedy algorithm that finds the cheapest ordering for performing a sequence of TTM operations, which is found at the core of HOOI algorithm.

The organization of the chapter is as follows. In the next section, we show how dimension trees can be used for computing TTM and MTTKRP steps in HOOI and CP-ALS algorithms. In Section 7.1, we provide the dimension tree-based HOOI algorithm followed by all our theoretical findings regarding an optimal tree structure. Similarly, in Section 7.2, we give the CP-ALS algorithm using dimension trees followed by related theorems for optimality.

The work in this chapter is currently under review [48].

7.1 Computing dense Tucker decomposition using dimension trees

In computing Tucker decomposition using Algorithm 3, performing successive TTMs to compute the resulting tensor $\mathcal{Y}$ constitutes the most expensive step in each ALS subiteration. This step involves the multiplication of $\mathcal{X}$ with the set $\{\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}\} \setminus \mathbf{U}^{(n)}$ of matrices to eventually update $\mathbf{U}^{(n)}$ at the end of the subiteration for mode n. Here, we propose an efficient algorithm to perform this step faster with the following observation, which is similar to that discussed in Chapter 3. For a 4-dimensional tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$, in the first two ALS subiterations, one needs to compute $\mathcal{X} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$ and $\mathcal{X} \times_1 \mathbf{U}^{(1)} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$ and subsequently update the corresponding matrices $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$, respectively. Note that in doing so, $\mathbf{U}^{(3)}$ and $\mathbf{U}^{(4)}$ remains unchanged, which brings about the possibility of computing the intermediate tensor $\mathcal{Z} = \mathcal{X} \times_3 \mathbf{U}^{(3)} \times_4 \mathbf{U}^{(4)}$, then reusing this partial result in the first and second subiterations as $\mathcal{Z} \times_2 \mathbf{U}^{(2)}$ and $\mathcal{Z} \times_1 \mathbf{U}^{(1)}$, respectively. This computation stays valid as one can perform TTM in a set of distinct modes in any order. This way, however, the cost of TTMs reduces significantly.

For an N-dimensional tensor, we are interested in identifying and reusing such common partial results as much as possible in a systematical way. This can be achieved by using a dimension tree $\mathcal{T}$ as follows. With each node $t \in \mathcal{T}$, we associate a tensor $T(t)$ corresponding to the multiplication $\mathcal{X} \times_{m_1} \mathbf{U}^{(m_1)} \times_{m_2} \cdots \times_{m_{|\mu' t|}} \mathbf{U}^{(m_{|\mu' t|})}$ where $\mu'(t) = \{m_1, \ldots, |\mu' t|\}$. Note that this implies $T(\textsc{Root}(\mathcal{T})) = \mathcal{X}$, and $T(\mathcal{L}_n) = \mathcal{X} \times_{-n} \mathbf{U}^{(n)}$ for all $n \in \mathbb{N}_N$. With this tree structure, the tensor of any non-root tree node t can be computed from that of its parent as $T(t) = T(P(t)) \times_{d_1} \mathbf{U}^{(d_1)} \times_{d_2} \cdots \times_{d_k} \mathbf{U}^{(d_k)}$ where $\mu(P(t)) \setminus \mu(t) = \{d_1, \ldots, d_k\}$. Indeed, if $T(P(t))$ does not exist, it needs to be similarly computed from its parent's tensor. We provide the algorithm for computing the tensor of any tree node t in Algorithm 16.

Algorithm 16 DTREE-TTM: Performing TTM on a dimension tree

Input: t: A dimension tree node
Output: $T(t)$: The tensor of t
1: **if** EXISTS($T(t)$) **then**
2: **return** $T(t)$
3: $T(t) \leftarrow$ DTREE-TTM($P(t)$)
4: **for** $d \in \mu(t) \setminus \mu(P(t))$ **do**
5: $T(t) \leftarrow T(t) \times_d \mathbf{U}^{(d)}$
6: **return** $T(t)$

One indeterminacy we have in this algorithm is the order of TTMs performed at Line 5. The order of these multiplications can have a significant impact on the overall computational cost. For minimizing this cost, we use an optimal greedy ordering method that minimizes the cost of a series of TTMs in distinct modes, which is demonstrated in the following theorem.

Theorem 4 (Optimal TTM order)**.** *Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be a tensor and $D = \{d_1, \ldots, d_{|D|}\}, D \subseteq \mathbb{N}_N$ represent the set of dimensions in which $\mathcal{X}$ is to be multiplied with a set $\{\mathbf{U}_1, \ldots, \mathbf{U}_{|D|}\}$ of matrices of corresponding sizes, i.e., $\mathbf{U}_i \in \mathbb{R}^{I_{d_i} \times R_i}$. Let $\beta_i = R_i/(1 - R_i/I_{d_i})$, and suppose w.l.g that $\beta_1 \leq \beta_2 \leq \cdots \leq \beta_{|D|}$. Then, the total cost of TTM is minimized with the multiplication order $\mathcal{X} \times_{d_1} \mathbf{U}_1 \times_{d_2} \mathbf{U}_2 \times_{d_3} \cdots \times_{d_{|D|}} \mathbf{U}_{|D|}$.*

Proof. Let $\alpha_n = R_n/I_n$ for $n \in \mathbb{N}_N$ and $\mathcal{T} = \prod_{n \in \mathbb{N}_N} I_n$ be the number of elements in $\mathcal{X}$. Then, the cost of the multiplication $\mathcal{X} \times_1 \mathbf{U}^{(1)} \times_2 \cdots \times_N \mathbf{U}^{(N)}$ becomes $\mathcal{T}(R_1 + \alpha_1 R_2 + \alpha_1 \alpha_2 R_3 + \cdots +$

$\alpha_1 \dots \alpha_{N-1} R_N)$ according to the formulation in (2.2).

Consider the permutation σ of $\mathbb{N}_N$ that minimizes the cost of the multiplication $\boldsymbol{\mathcal{X}} \times_{\sigma(1)} \mathbf{U}^{(\sigma(1))} \times_{\sigma(2)} \cdots \times_{\sigma(N)} \mathbf{U}^{(\sigma(N))}$. This cost is $C_\sigma = \mathcal{T}(R_{\sigma(1)} + \alpha_{\sigma(1)} R_{\sigma(2)} + \alpha_{\sigma(1)}\alpha_{\sigma(2} R_{\sigma(3)} + \cdots + \alpha_{\sigma(1)} \dots \alpha_{\sigma(N-1)} R_{\sigma(N)})$. Assume by contradiction that σ is not the identity permutation, and let i be the first index such that $\beta_{\sigma(i)} > \beta_{\sigma(i+1)}$. We rewrite the cost C_σ as $C_\sigma = C_1 + C_2 + C_3$ where

- $C_1 = \sum_{j=1}^{i-1} (\Pi_{k=1}^{j-1} \alpha_{\sigma(k)}) R_{\sigma(j)}$ corresponds to the first $i-1$ terms;
- $C_2 = (\Pi_{k=1}^{i-1})(R_{\sigma(i)} + \alpha_{\sigma(i)} R_{\sigma(i+1)})$ corresponds to terms i and $i+1$;
- $C_3 = \sum_{j=i+1}^{n} (\Pi_{k=1}^{j-1} \alpha_{\sigma(k)}) R_{\sigma(j)}$ corresponds to the last $n-i-1$ terms.

Now let us permute dimensions $\sigma(i)$ and $\sigma(i+1)$. Formally, this amounts to replacing σ by $\sigma^* = \tau_{i,i+1} \circ \sigma$, where $\tau_{i,i+1}$ is the transposition that exchanges elements in position i and $i+1$. The costs C_1 and C_3 are not modified, so that $C_{\sigma^*} = C_1 + C_2^* + C_3$, with

$$C_2^* = (\Pi_{k=1}^{i-1})(R_{\sigma(i+1)} + \alpha_{\sigma(i+1)} R_{\sigma(i)}).$$

But

$$\begin{aligned} C_2^* < C_2 \quad &\Leftrightarrow \quad R_{\sigma(i+1)} + \alpha_{\sigma(i+1)} R_{\sigma(i)} < R_{\sigma(i)} + \alpha_{\sigma(i)} R_{\sigma(i+1)} \\ &\Leftrightarrow \quad R_{\sigma(i+1)}(1 - \alpha_{\sigma(i)}) < R_{\sigma(i)}(1 - \alpha_{\sigma(i+1)}) \\ &\Leftrightarrow \quad \beta_{\sigma(i+1)} < \beta_{\sigma(i)}. \end{aligned}$$

Hence, permuting dimensions $\sigma(i)$ and $\sigma(i+1)$ does decrease the optimal cost, the desired contradiction. This concludes the proof. □

In Algorithm 17, we provide the HOOI algorithm that performs TTMs using a dimension tree $\mathcal{T}$ and the TTM routine in Algorithm 16. In the ALS subiteration for mode n, it starts at Line 6 with destroying all tensors in the tree that involves a multiplication with $\mathbf{U}^{(n)}$, as $\mathbf{U}^{(n)}$ will subsequently be updated in that subiteration, invalidating these tensors. Note that every tree node t that does not lie in the path from $\mathcal{L}_n$ to ROOT($\mathcal{T}$) has $n \in \mu(t)'$, hence their tensors are destroyed. Next, DTREE-TTM is performed for the leaf node $\mathcal{L}_n$ at Line 7 to compute $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_{-n} \mathbf{U}^{(n)}$. Algorithm 16 only computes tensors of nodes in the path from $\mathcal{L}_n$ to ROOT($\mathcal{T}$), and all other tensors not lying on this path are already destroyed at Line 6; therefore, at any instant of Algorithm 17, only the tensors of nodes lying on a path from a leaf to the root are stored. Following the computation of TTM for $\mathcal{L}_n$, $\mathbf{U}^{(n)}$ is similarly updated by performing a truncated SVD on $\mathbf{Y}_{(n)}$ at Line 8. After all N ALS subiterations, the core tensor $\boldsymbol{\mathcal{G}}$ is formed using $\mathbf{U}^{(n)}$ and the last $\boldsymbol{\mathcal{Y}}$ corresponding to $\boldsymbol{\mathcal{X}} \times_{-n} \mathbf{U}^{(n)}$ at Line 9.

Since $\mathcal{L}_1, \dots, \mathcal{L}_N$ corresponds to a post-order traversal of the tree, the tensor of each tree node is computed when DTREE-TTM is called for its first leaf descendant node, stays valid and reused throughout the subiterations of all its leaf descendants, and is destroyed in the subiteration following its last leaf descendant. This implies the tensor of every tree node is computed and destroyed exactly once per HOOI iteration.

An important point in Algorithm 17 is that $\mathcal{T}$ can be of any shape (balanced, unbalanced, k-ary), and can employ arbitrary partitions of dimensions at each level. Indeed, each tree configuration yields a different computational cost for TTMs; hence, we are interested in finding a tree topology that minimizes this cost. Intuitively, we argue that using a binary tree is a good choice because of the following observation. For a non-leaf node $t \in \mathcal{T}$ with children $t_1, \dots, t_k$ ($k \geq 2$), $\mu(t)$ is partitioned into k disjoint sets, namely $\mu(t_1), \dots, \mu(t_k)$. Performing DTREE-TTM on each child t_l for $l \in \mathbb{N}_k$ requires TTM in dimensions $\mu'(t_l) = \mu(t) \setminus \mu(t_l)$. Overall, calling DTREE-TTM on all

Algorithm 17 DTREE-HOOI: Dimension tree-based HOOI algorithm

Input: $\mathcal{X}$: An N-dimensional tensor
$R_1, \ldots, R_N$: The rank of the decomposition for each mode
$\mathcal{T}$: A dimension tree for N dimensions
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**
    for all $t \in \mathcal{T}$ **do**
      if $n \in \mu'(t)$ **then**
        DESTROY$(T(t))$ ▶ Destroy all tensors that are multiplied by $\mathbf{U}^{(n)}$.
    $\mathcal{Y} \leftarrow$ DTREE-TTM$(\mathcal{L}_n)$ ▶ Perform TTM for the leaf node.
    $\mathbf{U}^{(n)} \leftarrow R_n$ leading left singular vectors of $\mathbf{Y}_{(n)}$
  $\mathcal{G} \leftarrow \mathcal{Y} \times_N \mathbf{U}^{(N)^T}$ ▶ Form the core tensor.
until convergence or the maximum number of iterations
return $[\![\mathcal{G}; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$

k children requires $k-1$ TTMs for each dimension in $\mu(t)$, which is minimized for $k = 2$ using a binary tree. Nevertheless, in some extreme cases, using a binary tree may not be optimal in terms of actual computational cost, despite avoiding extra TTMs, and we provide such a counter-example using a 6-dimensional tensor in Section 7.1.1. In most practical scenarios, however, binary dimension tree (BDT) is expected to provide optimal or close to optimal results; hence, we conduct the analysis using BDT in the rest of the discussion in this section.

We now investigate the computational complexity of finding an optimal BDT that minimizes the TTM cost in DTREE-HOOI. We will show that finding an optimal "balanced" binary dimension tree (BBDT) in which the dimensions $\mu(t)$ of each tree node t are equitably partitioned to its children t_1 and t_2, i.e., $|\mu(t_1)|, |\mu(t_2)| \geq \lfloor |\mu(t)|/2 \rfloor$, is NP-hard, and conjecture that the problem using arbitrary dimension trees still remains NP-hard. In doing so, we consider the PRODUCT-PARTITION problem, which is NP-hard [76]: Given a set $S = \{s_1, \ldots, s_K\}$ of positive integers, find a subset S' of S such that $\max(\prod_{i \in S'} i, \prod_{i \in S \setminus S'} i)$ is minimized. We use a variant of this problem called BALANCED-PRODUCT-PARTITION where $|S| = 2K$ is a multiset for $K > 0$, and $|S'| = K$ is another multiset containing exactly a half of the elements of S. This problem still remains NP-hard as one can trivially solve any instance of PRODUCT-PARTITION using an instance of BALANCED-PRODUCT-PARTITION with the following polynomial-time reduction. For the given set S of size K for PRODUCT-PARTITION, we create a multiset Q of size $2K$ having all elements of S, and K 1s. Let Q' be the optimal solution for BALANCED-PRODUCT-PARTITION of Q, and $S' = Q' \cap S$ be the set of elements of the optimal solution Q' belonging to S. Since the multiset $Q' \setminus S$ can only contain 1s, we get $\prod_{i \in S'} i = \prod_{i \in Q'} i$ as well as $\prod_{i \in S \setminus S'} i = \prod_{i \in Q \setminus Q'}$. Therefore, $\max(\prod_{i \in Q'} i, \prod_{i \in Q \setminus Q'} i)$ is minimized if and only if $\max(\prod_{i \in S'} i, \prod_{i \in S \setminus S'} i)$ is minimized, which makes S' an optimal solution for PRODUCT-PARTITION of S. Note that the objective of BALANCED-PRODUCT-PARTITION is also equivalent to minimizing the sum $\prod_{i \in Q'} i + \prod_{i \in Q \setminus Q'} i$, and the same for PRODUCT-PARTITION.

Theorem 5 (Optimal BBDT for Tucker decomposition)**.** *Finding a BBDT that minimizes the cost of TTMs in computing Tucker decomposition is NP-hard.*

Proof. We perform a reduction from BALANCED-PRODUCT-PARTITION using a set $S = \{s_1, \ldots, s_{2N}\}$. The intuition of the proof is as follows. We aim to find an optimal BBDT for computing the Tucker

decomposition of a $2N+2$-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_{2N+2}}$ using ranks of approximation $R_1, \ldots, R_{2N+2}$. The reduction associates the first $2N$ dimensions of the tensor with the corresponding elements of S, and uses two more auxiliary dimensions. Forming a BBDT for this tensor yields $N+1$ dimensions in both sets $\mu(t_1)$ and $\mu(t_2)$ of the children t_1 and t_2 of the root. The goal is to have exactly N dimensions corresponding to elements of S in both $\mu(t_1)$ and $\mu(t_2)$, and to show that these two subsets of S provide an optimal solution for BALANCED-PRODUCT-PARTITION for an optimal BBDT. We achieve this in two steps. First, we analyze the case where both $\mu(t_1)$ and $\mu(t_2)$ have N dimensions associated with S, and provide lower and upper bounds for the TTM cost of a such BBDT. We show in this case that the cost of the BBDT is minimized when the multiplications of the elements of two associated subsets of S (with $\mu(t_1)$ and $\mu(t_2)$) are balanced (i.e., the sum of two multiplications is minimized), which is effectively the objective of BALANCED-PRODUCT-PARTITION. Next, we argue that if a child of the root has $N-1$ (or equivalently, $N+1$) dimensions associated with S, then the BBDT cannot be optimal, exceeding the provided upper bound in the previous case. As both children of the root must have $N+1$ dimensions in total, there would be no other partitioning possibilities regarding these $2N$ dimensions associated with S, and the two subsets of S of size N associated with $\mu(t_1)$ and $\mu(t_2)$ in the former case would provide an optimal solution for BALANCED-PRODUCT-PARTITION of S.

We set $R_i = s_i$ and $I_i = 3\prod_{s\in S} s$ for $i \in \mathbb{N}_{2N}$. We use $R_{2N+1} = R_{2N+2} = K_1(I_1)^N = K_1(3\prod_{s\in S} s_i)^N$ and $I_{2N+1} = I_{2N+2} = K_2 R_{2N+1}$ where the coefficients $K_1 \geq 1$ and $K_2 \geq 1$ are left to be determined appropriately in the course of the proof. This yields $\alpha_i = s_i/(3\prod_{s\in S} s) \leq 1/3$ for $i \in \mathbb{N}_{2N}$ which also implies $\beta_i = R_i/(1-\alpha_i) \leq 3s_i/2$. For dimensions $2N+1$ and $2N+2$, we get $\alpha_{2N+1} = \alpha_{2N+2} = K_2^{-1}$, and

$$\begin{aligned}\beta_{2N+1} = \beta_{2N+2} &= R_{2N+1}/(1-\alpha_{2N+1}) \\ &> R_{2N+1} = K_1(3\prod_{s\in S} s)^N \\ &> 3s_{max}/2 \geq \beta_i, i \in N_{2N},\end{aligned}$$

where $s_{max} = \max_{s\in S} s$. Hence, these two dimensions are to be multiplied after dimensions $1 \ldots 2N$ in an optimal solution according to Theorem 4.

We start by investigating the cost of a BBDT where dimensions $2N+1$ and $2N+2$ reside in $\mu(t_1)$ and $\mu(t_2)$, respectively. Without loss of generality, suppose that $\mu(t_1) = \{1, \ldots, N, 2N+1\}$ and $\mu(t_2) = \{N+1, \ldots, 2N, 2N+2\}$, and that $\beta_1 \leq \cdots \leq \beta_N$ and $\beta_{N+1} \leq \cdots \leq \beta_{2N}$ (which can otherwise be obtained by a proper permutation of dimensions). Therefore, we perform TTMs with $\boldsymbol{\mathcal{X}}$ in the sequence of dimensions $1, \ldots, N, 2N+1$ and $N+1, \ldots, 2N, 2N+2$ for nodes t_2 and t_1, respectively, as this corresponds to the optimal dimension ordering stated in Theorem 4. Let $\mathcal{C}$ represent the TTM cost of an optimal BBDT with the given $\mu(t_1)$ and $\mu(t_2)$, and $\mathcal{Z} = \prod_{i\in\mathbb{N}_{2N+2}} I_i$ be the number of elements in $\boldsymbol{\mathcal{X}}$. Then, we can express the TTM cost as

$$\begin{aligned}\mathcal{C} = \mathcal{Z}(&s_1 + \alpha_1 s_2 + \cdots + \alpha_1 \ldots \alpha_{N-1} s_N + \alpha_1 \ldots \alpha_N K_1(3\prod_{s\in S} s)^N + \\ &s_{N+1} + \alpha_{N+1} s_{N+2} + \cdots + \alpha_{N+1} \ldots \alpha_{2N-1} s_{2N} + \alpha_{N+1} \ldots \alpha_{2N} K_1(3\prod_{s\in S} s)^N) + \\ &C(1, \ldots, N, 2N+1) + C(N+1, \ldots, 2N, 2N+2)),\end{aligned} \tag{7.1}$$

where the first two summands correspond to the TTM cost due to nodes t_2 and t_1, whereas $C(1,\ldots,N,2N+1)$ and $C(N+1,\ldots,2N,2N+2)$ denote the total TTM cost of all remaining nodes in the subtrees rooted at t_1 and t_2, respectively, in an optimal BBDT. We rewrite (7.1) as follows

$$\begin{aligned}\mathcal{C} = \mathcal{Z}K_1[K_1^{-1}(&s_1 + \alpha_1 s_2 + \cdots + \alpha_1 \ldots \alpha_{N-1} s_N + \\ &s_{N+1} + \alpha_{N+1} s_{N+2} + \cdots + \alpha_{N+1} \ldots \alpha_{2N-1} s_{2N}) + \\ &s_1 \ldots s_N + s_{N+1} \ldots s_{2N}] + \\ &C(1,\ldots,N,2N+1) + C(N+1,\ldots,2N,2N+2),\end{aligned} \tag{7.2}$$

□

and obtain a trivial lower bound

$$\mathcal{C} > \mathcal{Z}K_1(s_1 \ldots s_N + s_{N+1} \ldots s_{2N}), \tag{7.3}$$

as the rest of the summands are positive. Next, we aim find an upper bound for $\mathcal{C}$ by bounding the costs $C(1,\ldots,N,2N+1)$ and $C(N+1,\ldots,2N,2N+2)$ as following. $C(1,\ldots,N,2N+1)$ corresponds to the TTM cost using a BBDT having $N+1$ leaves (and at most $2N$ nodes in total excluding its root t_1). The number of elements in the tensor at the root of this sub-tree is $\mathcal{Z}_1 = \mathcal{Z}\alpha_{N+1}\ldots\alpha_{2N}K_2^{-1}$ as the tensor is obtained from the multiplication of $\boldsymbol{\mathcal{X}}$ in all modes in $\mu(t_2)$. Each node can require at most one TTM per each dimension in $\mu(t_1)$, and the cost of the TTM in a dimension d cannot exceed $\mathcal{Z}_1 R_d$ since the tensor cannot grow larger after multiplications ($R_d \leq I_d$). Therefore, we obtain the following upper bound on the total TTM cost $C(1,\ldots,N,2N+1)$:

$$\begin{aligned}C(1,\ldots,N,2N+1) &\leq 2N\mathcal{Z}_1\Big(\sum_{i=1}^{N} R_i + R_{2N+1}\Big) \\ &< \mathcal{Z}\alpha_{N+1}\ldots\alpha_{2N}K_2^{-1}2N\Big(\sum_{i=1}^{2N+2} R_i\Big).\end{aligned}$$

Finally, setting $K_2 = 2N(\sum_{i=1}^{2N+2} R_i)$ yields

$$C(1,\ldots,N,2N+1) < \mathcal{Z}\alpha_{N+1}\ldots\alpha_{2N}.$$

We similarly obtain the upper bound $\mathcal{Z}\alpha_1\ldots\alpha_N$ for $C(N+1,\ldots,2N,2N+2)$. Using these upper bounds in (7.2) gives

$$\begin{aligned}\mathcal{C} < \mathcal{Z}K_1\big(&K_1^{-1}(s_1 + \alpha_1 s_2 + \cdots + \alpha_1 \ldots \alpha_{N-1} s_N + \\ &s_{N+1} + \alpha_{N+1} s_{N+2} + \cdots + \alpha_{N+1} \ldots \alpha_{2N-1} s_{2N} + \\ &\alpha_{N+1} \ldots \alpha_{2N} + \alpha_1 \ldots \alpha_N) + \\ &s_1 \ldots s_N + s_{N+1} \ldots s_{2N}\big).\end{aligned}$$

Since $\alpha_i \leq 1/3$ for $i \in \mathbb{N}_{2N}$, setting $K_1 = (4s_{max} + 2)$ yields the final upper bound

$$\mathcal{C} < \mathcal{Z}K_1(s_1 \dots s_N + s_{N+1} \dots s_{2N} + 1). \tag{7.4}$$

Next, we analyze the case where dimensions $2N+1$ and $2N+2$ reside in the same child of the root, namely t_1. Suppose w.l.g that $\mu(t_1) = \{1, \dots, N-1, 2N+1, 2N+2\}$ and $\mu(t_2) = \{N, \dots, 2N\}$. Then, the TTM cost for t_2 becomes

$$\begin{aligned}
\mathcal{C}_{t_2} &= \mathcal{Z}(s_1 + \alpha_1 s_2 + \dots + \alpha_1 \dots \alpha_{N-2} s_{N-1} + \\
&\quad \alpha_1 \dots \alpha_{N-1} K_1 (3 \prod_{s \in S} s)^N + \alpha_1 \dots \alpha_{N-1} K_2^{-1} K_1 (3 \prod_{s \in S} s)^N) \\
&> Z \alpha_1 \dots \alpha_{N-1} K_1 (3 \prod_{s \in S} s)^N \\
&= \mathcal{Z} K_1 (s_1 \dots s_{N-1} 3 \prod_{s \in S} s) \\
&\geq \mathcal{Z} K_1 (s_1 \dots s_N + s_{N+1} \dots s_{2N} + 1),
\end{aligned}$$

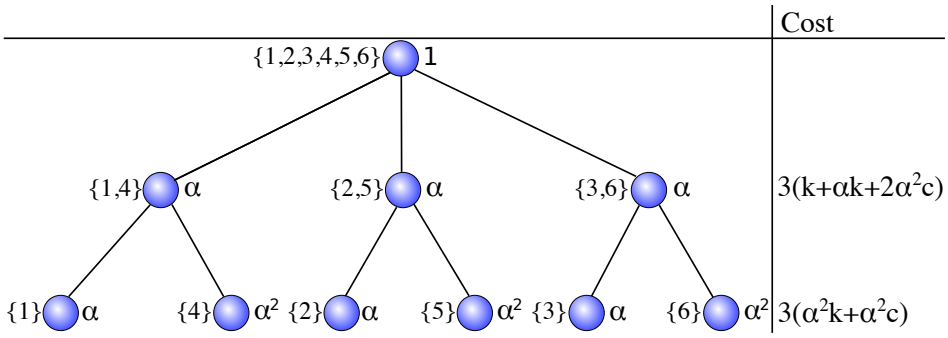
which already exceeds the upper bound in (7.4). Therefore, we conclude that dimensions $2N+1$ and $2N+2$ cannot reside in the same child of the root in an optimal solution. As a result, we obtain exactly N dimensions associated with S in $\mu(t_1)$ and $\mu(t_2)$ in an optimal solution.

Using this result and the bounds in (7.3) and (7.4) we can finally perform a reduction from the decision version of BALANCED-PRODUCT-PARTITION: Given a set S of $2N$ positive integers, is there a $S' \subset S, |S'| = N$ such that $\prod_{s \in S'} s + \prod_{s \in S \setminus S'} s \leq \mathcal{C}$ for some $\mathcal{C} \geq 1$? We claim that such S' exists if and only if there exists a BBDT constructed in the aforementioned manner whose cost is smaller than $\mathcal{Z}K_1(\mathcal{C}+1)$. If a such S' exists, then (7.4) suggests that one can construct a corresponding BBDT whose cost is less than $\mathcal{Z}K_1(\mathcal{C}+1)$. On the contrary, if there exists no such S', then any S' yields $\prod_{s \in S'} s + \prod_{s \in S \setminus S'} s \geq \mathcal{C}+1$, in which case the cost of the associated BBDT exceeds $\mathcal{Z}K_1(\mathcal{C}+1)$ due to (7.3), which concludes the proof.

7.1.1 Counter-example: Tensor having no optimal BDT

Here, we provide a counter-example using a 6-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \dots \times I_6}$ to show that using a BDT is not necessarily optimal to compute Tucker decomposition. The first three and the last three dimensions of $\boldsymbol{\mathcal{X}}$ have identical sizes and ranks of approximation. Specifically, we let $I_1 = I_2 = I_3 = k$ and $R_1 = R_2 = R_3 = R < k$ in the first three dimensions, and $I_4 = I_5 = I_6 = R_4 = R_5 = R_6 = c$ in the last three dimensions. We call the first three and the last tree dimensions *type-1* and *type-2 dimensions*, respectively. Note that $\alpha = \alpha_1 = \alpha_2 = \alpha_3 < \alpha_4 = \alpha_5 = \alpha_6 = 1$, and similarly $\beta_1 = \beta_2 = \beta_3 < \beta_4 = \beta_5 = \beta_6 = \infty$, therefore, type-1 dimensions are to be multiplied before type-2 dimensions according to Theorem 4. In Figure 7.1, we provide a ternary dimension tree with a total cost of $(3 + 3\alpha + 3\alpha^2)k + 9\alpha^2 c$. We can choose c arbitrarily large so that the term $9\alpha^2 c$ dominates the cost, and α small enough so that $\alpha^0 c \gg 9\alpha^2 c$ and $\alpha^1 c \gg 9\alpha^2 c$. In this case, any BDT whose cost involves a term of order $\alpha^0 c$ or $\alpha^1 c$ cannot be optimal, having a greater cost than the provided ternary tree.

Figure 7.1: A ternary dimension tree for $\mathcal{X}$. The μ set of the node is provided on the left, and the tensor size coefficient is provided on the right of each tree node. The TTM cost of each level is provided on the right.

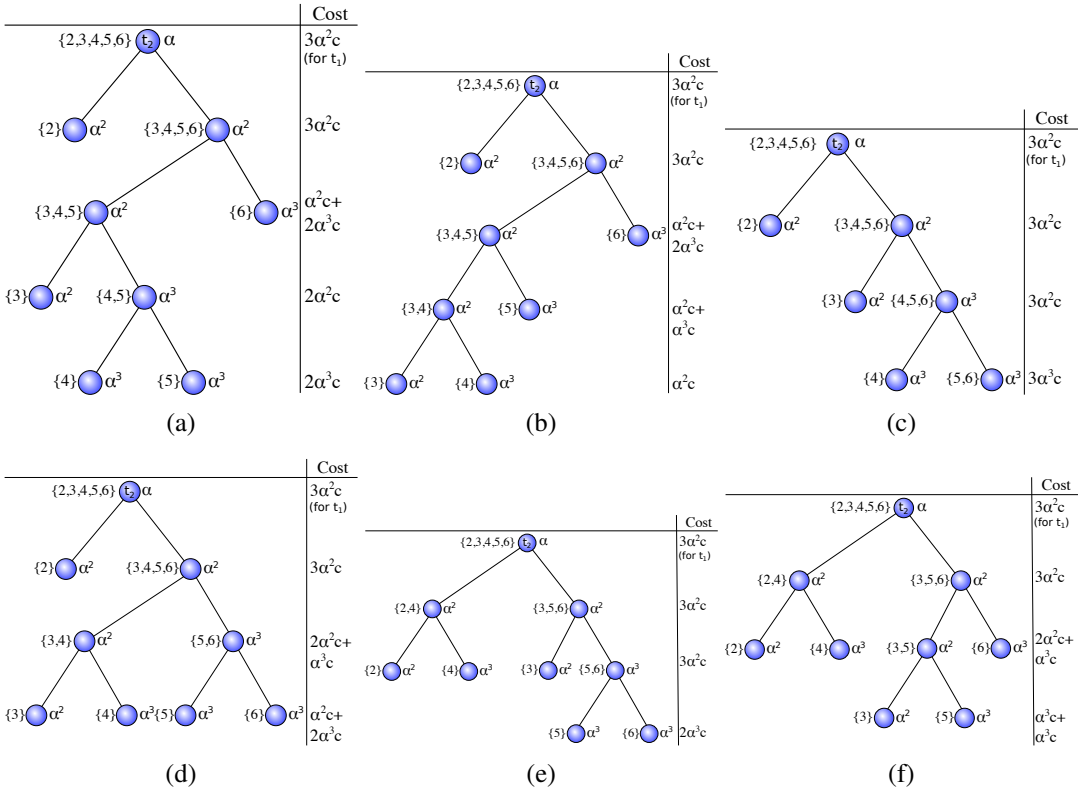


We now show that any BDT using $\mathcal{X}$ either has a cost with a term of order $\alpha^0 c$ or $\alpha^1 c$, or with a term $9\alpha^2 c + d\alpha^3 c$ with $d \geq 1$; hence, cannot be optimal with a sufficiently large c and a sufficiently small α. We do this by exhaustively considering all possible BDTs and analyzing their costs, while aggressively pruning tree configurations that cannot provide optimality. Luckily, most non-optimal configurations can easily be pruned due to symmetry (as we only have two types of dimensions), leaving us with only a handful of instances to consider. We begin by partitioning type-2 dimensions to the children t_1 and t_2 of the root. There are only two possibilities: $4 \in \mu(t_1)$ and $5, 6 \in \mu(t_2)$, or $4, 5, 6 \in \mu(t_2)$. Since we have only three type-1 dimensions, in the former case either $\mu(t_1)$ or $\mu(t_2)$ will have one or zero type-1 dimension, while the other set having two or three of them. In this case, the TTM cost of the other vertex involves a term $\alpha^0 c$ or $\alpha^1 c$, which already renders this configuration non-optimal. Therefore, we only consider the partition $4, 5, 6 \in \mu(t_2)$, which is the only one that can possibly provide an optimal solution. In this case, there are three possible configurations after partitioning type-1 dimensions: $\mu(t_1) = \{1\}$ and $\mu(t_2) = \{2, 3, 4, 5, 6\}$, $\mu(t_1) = \{1, 2\}$ and $\mu(t_2) = \{3, 4, 5, 6\}$, or $\mu(t_1) = \{1, 2, 3\}$ and $\mu(t_2) = \{4, 5, 6\}$. Note that in the second and third configurations, computing TTM for t_1 involves a term $\alpha^1 c$ and $\alpha^0 c$, respectively, which prevents optimality. Hence, in the rest of the discussion we only consider the first configuration, in which t_1 incurs the TTM cost $3\alpha^2 c$. We focus on the cost of the sub-tree rooted at t_2, and count only the cost due to terms with the coefficient c. Note that if the remaining type-1 dimensions, namely 2 and 3, reside in the same child of t_2, the other child of t_2 incurs a cost of at least $\alpha^1 c$; therefore, 2 and 3 must reside in different children of t_2. In Figure 7.2 we provide six such possibilities, all of which incur a cost of $9\alpha^2 c + c\alpha^3 c$ with $c \geq 1$. Therefore, we conclude that a BDT cannot be optimal for the given $\mathcal{X}$ for sufficiently small α and large c.

7.2 Computing dense CP decomposition using dimension trees

In computing Tucker decomposition, we have shown how dimension trees reduce the computational cost of the TTM step by storing and reusing common partial TTM results used across different subiterations of HOOI. In (2.4), we have a TTV formulation of the expensive MTTKRP step in CP-ALS, which similarly enables precomputing and reusing partial TTV results, e.g., for a 4-dimensional tensor $\mathcal{X}$, one can similarly compute $\mathcal{Z} = \mathcal{X} \times_3 \mathbf{u}_r^{(3)} \times_4 \mathbf{u}_r^{(4)}$, then use this partial result to obtain both $\mathbf{M}^{(1)}(:, r) = \mathcal{Z} \times_2 \mathbf{u}_r^{(2)}$ and $\mathbf{M}^{(2)}(:, r) = \mathcal{Z} \times_1 \mathbf{u}_r^{(1)}$. One difference in this case is that a series of R

Figure 7.2: All possible configurations and associated TTM costs at each level of BDTs rooted at t_2. $\mu(t)$ is given on the left of each tree node t, On the right of a tree node, the size coefficient of its tensor is provided, i.e., t_2 has a tensor of size $(I_1I_2I_3I_4I_5I_6)\alpha$.



TTVs is needed to compute all columns of $\mathbf{M}^{(1)}$ and $\mathbf{M}^{(2)}$. For this reason, each non-root tree node t holds tensors $T_r(t)$ for each $r \in \mathbb{N}_R$ corresponding to the partial result for the rth TTV. The root node, however, possesses only the original tensor $\boldsymbol{\mathcal{X}}$.

Algorithm 18 DTREE-TTV: Performing TTV on a dimension tree for a dense tensor

Input: t: A dimension tree node
Output: $T_{:}(t)$ are computed.
1: **if** EXISTS($T_{:}(t)$) **then**
2: **return**
3: $\delta(t) \leftarrow \mu(P(t)) \setminus \mu(t)$
4: DTREE-TTV($P(t)$)
5: **if** IS-ROOT($P(t)$) **then**
6: $\mathbf{M} \leftarrow T(P(t))_{(\mu(t))}(\odot_{(n\in\delta(t))}\mathbf{U}^{(n)})$
7: **for** r = 1 ... R **do**
8: vec($T_r(t)$) $\leftarrow \mathbf{M}(:, r)$
9: **else**
10: **for** r = 1 ... R **do**
11: vec($T_r(t)$) $\leftarrow T_r(P(t))_{(\mu(t))}(\otimes_{(n\in\delta(t))}\mathbf{u}_r^{(n)})$

We provide the method for performing TTVs using a dimension tree in Algorithm 18 for a tree node t. The algorithm similarly returns the tensors $T(t)$ if they are already computed. Otherwise, DTREE-TTV is called at Line 4 to obtain the parent $P(t)$'s tensors. If $P(t)$ is the root, then we perform traditional MTTKRP by properly matricizing $T(P(t))$ (which is $\boldsymbol{\mathcal{X}}$), and multiplying it with the Khatri-Rao product of all matrices corresponding to all dimensions except those in $\mu(t)$. This yields a matrix of size $\prod_{d\in\mu(t)} I_d \times R$, whose rth column corresponds to $T_r(t)$ in vectorized form. This step costs $\prod_{n\in\mu(P(t))} I_n = \prod_{n\in\mathbb{N}_N} I_n$ operations for matricizing the tensor, $R\prod_{n\in\mu(t)} I_n$ operations for forming the Khatri-Rao product, and finally $R\prod_{n\in\mu(P(t))} I_n = R\prod_{n\in\mathbb{N}_N} I_n$ operations for multiplying the matricized tensor with the Khatri-Rao product. If $P(t)$ is not the root, then each $T_r(t)$ is computed using the corresponding parent tensor $T_r(P(t))$. Here, we matricize the tensor, compute the Kronecker product of vectors to be multiplied, and finally execute the matrix-vector multiplication. For computing each tensor $T_r(t)$, this incurs $\prod_{n\in\mu(P(t))} I_n$ operations for matricizing the tensor $T_r(P(t))$, $\prod_{n\in\mu(t)} I_n$ operations for performing the Kronecker product of vectors, and finally $\prod_{n\in\mu(P(t))} I_n$ operations for multiplying the matricized tensor with the Khatri-Rao product, resulting in the overall cost $R(2\prod_{n\in\mu(P(t))} I_n + \prod_{n\in\mu(t)} I_n)$ for computing all R tensors, which we denote as $T_{:}(t)$. This technique is more efficient than multiplying the tensor with vectors one by one, which requires a matricization and a multiplication of the tensor for each vector. This is because TTV is a memory-bound operation, and this way we access the tensor only twice (once for matricization, once for multiplication), and the cost of Kronecker product is negligible compared with the cost of multiplication.

The dimension tree-based Algorithm 5 provided in Chapter 3 equally applies to both sparse and dense tensors. In the dense case, however, Algorithm 18 is to be employed to carry out TTVs at Line 9.

Indeed, the structure of the dimension tree plays a crucial role in the computational cost of MTTKRPs, and we are interested in finding the optimal tree structure minimizing this cost. In the following part, we provide two theorems pertaining to the computation of an optimal dimension tree for CP-ALS.

Theorem 6. *For any N-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times\cdots\times I_N}$, except $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{2\times2\times2}$, an optimal dimen-*

sion tree that minimizes the TTV cost in the execution of CP-ALS must be binary.

Proof. The proof is by construction of a BDT in the following manner. We take any dimension tree $\mathcal{T}$ which is not binary, focus on its any subtree rooted at a node t having $K > 2$ children, and finally replace this subtree with another having $K - 1$ children and a smaller cost. This ensures that a non-binary dimension tree cannot be optimal, as any such tree can be transformed into a BDT having less cost using a sequence of such transformations.

Let $t_1, \ldots, t_K$ be the children of the root t, and $\pi(m) = \prod_{n \in \mu(m)} I_n$ denote the size of the tensor of a tree node m. Let also $\mathcal{C}(t)$ be the TTV cost of an optimal dimension tree rooted at t, excluding the cost of the root. We first consider the case where t is not the root of $\mathcal{T}$. We can express the cost $\mathcal{C}_k$ of the original subtree with K children as

$$\mathcal{C}_k = R\big(2K\pi(t) + \sum_{k=1}^{K} \frac{\pi(t)}{\pi(t_k)}\big) + \sum_{k=1}^{K} \mathcal{C}(t_k), \tag{7.5}$$

where, for each dimension $k \in \mathbb{N}_K$, $R\frac{\pi(t)}{\pi(t_k)}$ is the Kronecker product cost, and $2R\pi(t)$ corresponds to the total cost of matricization and the multiplication of all R tensors with this product.

We now consider a variant of this tree where the nodes t_1 and t_2 are joint using an auxiliary node t_{12}, which is made a child of t with $\mu(t_{12}) = \mu(t_1) \cup \mu(t_2)$ (thus $\pi(t_{12}) = \pi(t_1)\pi(t_2)$). The rest of the children $t_3, \ldots, t_K$ of t are kept the same to obtain a subtree whose root t has $K - 1$ children and the following cost:

$$\begin{aligned}\mathcal{C}_{k-1} =& R\big(2(K-1)\pi(t) + \sum_{i=3}^{K} \frac{\pi(t)}{\pi(t_i)} + \frac{\pi(t)}{\pi(t_{12})} \\ &+ 4\pi(t_{12}) + \frac{\pi(t_{12})}{\pi(t_1)} + \frac{\pi(t_{12})}{\pi(t_2)}\big) + \sum_{i=1}^{K} \mathcal{C}(t_i).\end{aligned} \tag{7.6}$$

Let $L = \frac{\pi(t)}{\pi(t_1)\pi(t_2)}$. By taking the difference of (7.5) and (7.6) we finally obtain

$$\begin{aligned}\mathcal{C}_k - \mathcal{C}_{k-1} =& R\big(2\pi(t) + \frac{\pi(t)}{\pi(t_1)} + \frac{\pi(t)}{\pi(t_2)} - \frac{\pi(t)}{\pi(t_1)\pi(t_2)} \\ &- 4\pi(t_1)\pi(t_2) - \pi(t_2) - \pi(t_1)\big) \\ \geq& R\big(\frac{\pi(t)}{\pi(t_1)} + \frac{\pi(t)}{\pi(t_2)} - \frac{\pi(t)}{\pi(t_1)\pi(t_2)} - \pi(t_2) - \pi(t_1)\big) \\ =& R\big(\pi(t_2)L + \pi(t_1)L - L - \pi(t_2) - \pi(t_1)\big) \\ =& R\big((L-1)\big(\pi(t_2) + \pi(t_1)\big) - L\big) \\ \geq& R\big(4(L-1) - L\big) \geq 2R,\end{aligned} \tag{7.7}$$

as $\prod(t_k) \geq 2$ for all $k \in \mathbb{N}_K$, and $L \geq 2$ (since $I_n \geq 2$ for all $n \in \mathbb{N}_N$). Hence, the modified subtree whose root has $K - 1$ children always provides a smaller cost in this case.

Next, we consider the case where t is the root of $\boldsymbol{\mathcal{T}}$. In this case, the matricization costs only $\pi(t)$ for the root's tensor, hence the cost of the original tree having K children becomes

$$\mathcal{C}_k = K\pi(t) + R\big(K\pi(t) + \sum_{k=1}^{K} \frac{\pi(t)}{\pi(t_k)}\big) + \sum_{k=1}^{K} \mathcal{C}(t_k). \tag{7.8}$$

This time, we assume w.l.g that t_1 and t_2 have the two minimum tensor sizes among all children, i.e., $\pi(t_1), \pi(t_2) \leq \pi(t_k)$ for all $k = 3, \ldots, K$, which also implies that $\pi(t_1), \pi(t_2) \leq L$. For the tree having $K-1$ children, we obtain the following cost:

$$\begin{aligned}\mathcal{C}_{k-1} =&(K-1)\pi(t) + R\big((K-1)\pi(t) + \sum_{i=3}^{K} \frac{\pi(t)}{\pi(t_i)} + \frac{\pi(t)}{\pi(t_{12})} \\ &+ 4\pi(t_{12}) + \frac{\pi(t_{12})}{\pi(t_1)} + \frac{\pi(t_{12})}{\pi(t_2)}\big) + \sum_{i=1}^{K} \mathcal{C}(t_i).\end{aligned} \tag{7.9}$$

By taking the difference of (7.8) and (7.9) we obtain

$$\begin{aligned}\mathcal{C}_k - \mathcal{C}_{k-1} =&\pi(t) + R\big(\pi(t) + \frac{\pi(t)}{\pi(t_1)} + \frac{\pi(t)}{\pi(t_2)} - \frac{\pi(t)}{\pi(t_1)\pi(t_2)} \\ &- 4\pi(t_1)\pi(t_2) - \pi(t_2) - \pi(t_1)\big) \\ =&\pi(t) + R\big(L\pi(t_1)\pi(t_2) + L\pi(t_2) + L\pi(t_1) - L - 4\pi(t_1)\pi(t_2) - \pi(t_2) - \pi(t_1)\big) \\ =&\pi(t) + R\big((L-4)\pi(t_1)\pi(t_2) + (L-1)(\pi(t_1) + \pi(t_2)) - L\big).\end{aligned} \tag{7.10}$$

For $L \geq 4$, (7.10) yields

$$\begin{aligned}\mathcal{C}_k - \mathcal{C}_{k-1} >&R\big((L-1)(\pi(t_1) + \pi(t_2)) - L\big) \\ \geq&R\big(4(L-1) - L\big) \geq 8R.\end{aligned}$$

For $L = 3$, we obtain

$$\mathcal{C}_k - \mathcal{C}_{k-1} =\pi(t) + R\big(- \pi(t_1)\pi(t_2) + 2(\pi(t_1) + \pi(t_2)) - 3\big).$$

As $\pi(t_1), \pi(t_1) \leq L = 3$, we only have four possibilities for $\pi(t_1)$ and $\pi(t_2)$. For $\pi(t_1) = \pi(t_2) = 3$, we get

$$\mathcal{C}_k - \mathcal{C}_{k-1} =\pi(t) + R\big(- 9 + 2(6) - 3\big) = \pi(t) > 0.$$

$\pi(t_1) = 2$,$\pi(t_2) = 3$ and $\pi(t_1) = 3$,$\pi(t_2) = 2$ yields

$$\mathcal{C}_k - \mathcal{C}_{k-1} = \pi(t) + R\big(-6 + 2(5) - 3\big) = \pi(t) + R > 0.$$

The last possibility $\pi(t_1) = 2$ and $\pi(t_2) = 2$ gives

$$\mathcal{C}_k - \mathcal{C}_{k-1} = \pi(t) + R\big(-4 + 2(4) - 3\big) = \pi(t) + R > 0.$$

Finally, for $L = 2$, (7.10) becomes

$$\mathcal{C}_k - \mathcal{C}_{k-1} = \pi(t) + R\big(-3\pi(t_1)\pi(t_2) + \pi(t_1) + \pi(t_2) - 2\big).$$

Note that in this case, the only option is to have $\pi(t_1) = \pi(t_2) = 2$, which also implies that $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{2\times 2\times 2\times 2}$, hence $\pi(t) = 8$. As a result, we obtain

$$\mathcal{C}_k - \mathcal{C}_{k-1} = 8 + R\big(-3(4) + 2 + 2 - 2\big) = 8 - 10R < 0$$

for all $R \geq 1$. Therefore, we conclude that for all tensors except those in $\mathbb{R}^{2\times 2\times 2\times 2}$, the optimal dimension tree is binary. □

In the next theorem, we show that finding an optimal dimension tree is NP-hard.

Theorem 7 (Optimal dimension tree for CP decomposition)**.** *Finding an optimal dimension tree that minimizes the cost of TTVs in computing CP decomposition is NP-hard.*

Proof. We perform a reduction from PRODUCT-PARTITION using a set $S = \{s_1, \ldots, s_N\}$ of positive integers. We construct an $N + 4$-dimensional tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times\cdots\times I_{N+4}}$ whose first N dimensions correspond to the elements of S, i.e., $I_n = s_n$ for $n \in \mathbb{N}_N$. We let $I_{N+1} = I_{N+2} = I_{N+3} = I_{N+4} = k$ where k is to be determined appropriately in the course of the proof. We similarly name the first N and the last 4 dimensions as *type-1* and *type-2*, respectively.

We now analyze the cost of an optimal dimension tree that minimizes the MTTKRP cost in executing CP-ALS for $\boldsymbol{\mathcal{X}}$. This tree must be a BDT as suggested by Theorem 6. Let t be the root this with children t_1 and t_2. In an optimal BDT, we expect $\mu(t_1)$ to consist of dimensions corresponding to elements in an optimal solution $S' \subset S$ of PRODUCT-PARTITION. All dimensions corresponding to $S \setminus S'$ are similarly expected to belong to $\mu(t_2)$. In this scenario, we would only have three configurations for the partitioning of type-2 dimensions to $\mu(t_1)$ and $\mu(t_2)$ due to symmetry, namely $N+1, N+2, N+3, N+4 \in \mu(t_1)$, $N+1, N+2, N+3 \in \mu(t_1)$ and $N+4 \in \mu(t_2)$, and $N+1, N+2 \in \mu(t_1)$ and $N+3, N+4 \in \mu(t_4)$.

We first consider the case where $N+1, N+2 \in \mu(t_1)$ and $N+3, N+4 \in \mu(t_4)$, and analyze the cost of a BDT for a given partition of type-1 dimensions to $\mu(t_1)$ and $\mu(t_2)$. Without loss of generality, let $\mu(t_1) = \{1, \ldots, K, N+1, N+2\}$ and $\mu(t_2) = \{K+1, \ldots, N, N+3, N+4\}$ represent a such partition for some $K, 0 \leq K \leq N$ (all possible partitions can be obtained by a proper K and a permutation of the elements of S). We can express the cost of this BDT as

$$\begin{aligned}\mathcal{C} =&(2R+2)k^4 \prod_{n\in\mathbb{N}_N} s_n + Rk^2 \prod_{n=1}^{K} s_n + Rk^2 \prod_{n=K+1}^{N} s_n \\ &+ C(1,\dots,K,N+1,N+2) + C(K+1,\dots,N,N+3,N+4),\end{aligned} \tag{7.11}$$

where the first summand is the cost of the matricization and the multiplication of $\boldsymbol{\mathcal{X}}$ with the Khatri-Rao products of corresponding matrices for t_1 and t_2, the second and the third summands correspond to the cost of forming the Khatri-Rao product for t_2 and t_1, respectively. Here, $C(1,\dots,K,N+1,N+2)$ and $C(K+1,\dots,N,N+3,N+4)$ denote the total MTTKRP cost of subtrees rooted at t_1 and t_2, respectively, excluding the cost of t_1 and t_2. Both t_1 and t_2 have two children as $|\mu(t_1)|, |\mu(t_2)| \geq 2$. Each child of t_1 incurs a cost $2Rk^2 \prod_{n=1}^{K} s_n$ for matricizing the tensor of t_1, then multiplying it with the Kronecker products. Similarly, each of two children of t_2 has a cost of at least $2Rk^2 \prod_{n=K+1}^{N} s_n$. With these costs at hand (which exclude the cost of forming the Kronecker products for the children of t_1 and t_2, and further costs down the tree), we obtain the following lower bound from (7.11):

$$\mathcal{C} > (2R+2)k^4 \prod_{n\in\mathbb{N}_N} s_n + 5Rk^2(\prod_{n=1}^{K} s_n + \prod_{n=K+1}^{N} s_n). \tag{7.12}$$

Next, we aim to find an upper bound for an optimal $C(1,\dots,K,N+1,N+2)$ and $C(K+1,\dots,N,N+3,N+4)$, which in turn would yield an upper bound for $\mathcal{C}$. Let t_{11} and t_{12} be the children of t_1. We consider the cost for the case where $N+1 \in \mu(t_{11})$ and $N+2 \in \mu(t_{12})$. In this case, the size of the tensor of t_1 is $Rk^2 \prod_{n=1}^{K}$; therefore, cost of both t_{11} and t_{12} are upper bounded by $2Rk^2 \prod_{n=1}^{K} + Rk \prod_{n=1}^{K}$ as the cost of forming the Kronecker products cannot exceed $Rk \prod_{n=1}^{K}$ with the given partition of type-2 dimensions. In addition, note that t_1 is the root of a subtree having $K+2$ leaf nodes, and having up to $K+1$ non-leaf nodes each of which has two children; hence, it cannot have more than $2K$ nodes excluding t_1, t_{11}, and t_{12}. These nodes are descendants of either t_{11} or t_{12}; therefore, each of these nodes incurs a cost which cannot exceed $2Rk \prod_{i=1}^{K} s_i + Rk \prod_{i=1}^{K} s_i = 3Rk \prod_{i=1}^{K} s_i$, where the first and the second summands are upper bounds for the cost of the matricization of the tensor and its multiplication with the Kronecker product, and the cost of forming the Kronecker product, respectively. This results in the upper bound $C(1,\dots,k,N+1,N+2) < 4Rk^2 \prod_{n=1}^{K} s_n + (6K+2)Rk \prod_{n=1}^{K} s_n$. We similarly obtain the upper bound $4Rk^2 \prod_{n=K+1}^{N} s_n + (6N-6K+2)Rk \prod_{n=K+1}^{N} s_n$ for $C(k+1,\dots,N,N+3,N+4)$ for the case where $N+3 \in \mu(t_{21})$ and $N+4 \in \mu(t_{22})$. Finally, setting $k = l(6N+4) \prod_{i=1}^{N} s_i$ for any $l \geq 1$, and replacing these two upper bounds in (7.11) yields

$$\begin{aligned}\mathcal{C} <&(2R+2)k^4 \prod_{n\in\mathbb{N}_N} s_n + Rk^2\Big(5 \prod_{n=1}^{K} s_n + 5 \prod_{n=K+1}^{N} s_n + \\ &k^{-1}((6K+2) \prod_{i=1}^{K} s_i + (6N-6K+2) \prod_{i=K+1}^{N} s_i)\Big) \\ <&(2R+2)Rk^4 \prod_{n\in\mathbb{N}_N} s_n + 5Rk^2(\prod_{n=1}^{K} s_n + \prod_{n=k+1}^{N} s_n + 1).\end{aligned} \tag{7.13}$$

Next, we analyze the MTTKRP cost using two other partitionings of type-2 dimensions. Without loss of generality, we only consider the cases where $\mu_1 = \{1, \dots, K\}$ and $\mu_2 = \{K+1, \dots, N+1, N+2, N+3, N+4\}$, and $\mu_1 = \{1, \dots, K, N+1\}$ and $\mu_2 = \{K+1, \dots, N, N+2, N+3, N+4\}$. Considering only the cost of t_1 and t_2, the former case yields

$$\mathcal{C} > (2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + R \prod_{n=1}^{k} s_n + Rk^4 \prod_{n=k+1}^{N} s_n, \tag{7.14}$$

while the latter gives

$$\mathcal{C} > (2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + Rk \prod_{n=1}^{k} s_n + Rk^3 \prod_{n=k+1}^{N} s_n. \tag{7.15}$$

Setting $l = 5 \prod_{i=1}^{N} s_i + 1$ provides

$$\begin{aligned} \mathcal{C} &> (2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + 5Rk^2(6N+2)(\prod_{n=1}^{N} s_n + 1) \\ &> (2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + 5Rk^2(\prod_{n=1}^{K} s_n + \prod_{n=K+1} N s_n + 1) \end{aligned} \tag{7.16}$$

in both cases, which exceeds the upper bound provided in (7.13). Therefore, we conclude that these two partitionings cannot provide an optimal BDT.

We can now perform the reduction from the decision version of PRODUCT-PARTITION, knowing that an optimal BDT assigns exactly two dimensions of type-2 to each children t_1 and t_2 of the root t, and has the lower and the upper bounds provided in (7.12) and (7.13) with respect to the partitioning of type-1 dimensions to $\mu(t_1)$ and $\mu(t_2)$. We claim that a $S' \subseteq S$ with $\prod_{s \in S'} s + \prod_{s \in S \setminus S'} \leq C$ exists for some $C \geq 1$ if and only if there is a BDT for $\boldsymbol{\mathcal{X}}$ constructed in the aforementioned manner whose MTTKRP cost is smaller than $(2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + 5Rk^2(C+1)$ for any positive integer R. If there exists a such S', then (7.13) suggests that we can construct a BDT whose cost is inferior to $(2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + 5Rk^2(C+1)$. If $\prod_{s \in S'} s + \prod_{s \in S \setminus S'} \geq C+1$ of all subsets $S' \subseteq S$, then (7.12) implies that the cost of all BDTs exceed $(2R+2)k^4 \prod_{n \in \mathbb{N}_N} s_n + 5Rk^2(C+1)$, which concludes the proof. □

7.3 Conclusion

In this chapter, we introduced efficient algorithms to compute Tucker and CP decomposition of dense tensors. The algorithms utilize an novel computational scheme based on a dimension tree structure, and thereby enable re-using common partial TTM and MTTKRP results across different subiterations of the tensor decomposition algorithms. We also provided the first complexity results for this problem regarding an optimal dimension tree structure minimizing the associated computational costs. In

particular, we proved that finding an optimal binary tree to minimize the cost of TTM and MTTKRP operations respectively in DTREE-HOOI and DTREE-CP-ALS algorithms is NP-hard. We further showed that the optimal tree must be binary for TTV, except in one degenerate instance. On the contrary, we provided a counterexample using an optimal ternary tree for TTM. All these results lay the foundation for a complete analysis of the use of dimension trees for tensor decomposition. Our further work will involve finding fast exact algorithms as well as effective heuristics for finding an optimal/good dimension tree, and a thorough experimentation with the implementations of these two algorithms using dimension trees.

Part III

Sparse Tensor Software

Chapter 8

Sparse Tensor Software

8.1 An overview of recent sparse tensor software

At the beginning of this thesis, the only available parallel tensor factorization software was DFACTO; A C++ implementation for computing the CP decomposition of 3-dimensional tensors using MPI parallelism [22]. Even though this library significantly accelerated computing CP decomposition compared with the previous Hadoop implementations, it a major drawback preventing it from being scalable for analyzing massive tensor datasets using thousands of processors. The algorithm was not fully distributed in the sense that the factor matrices were entirely replicated in all MPI processes. This not only prevented scalability due to memory requirements, but also incurred a significant amount of redundant communication to transfer factor matrices to all processes after each update. In the meantime, SPLATT (C, OpenMP) [98] software came up with an efficient implementation and parallelization in shared memory, outperforming DFACTO by a large margin.

HYPERTENSOR [50] first came up in 2015 as a distributed-memory parallel implementation (MPI, C++) for computing the CP decomposition of sparse tensors. Its main advantage over DFACTO was to be able to distribute the tensor as well as factor matrices, and to employ a point-to-point messaging scheme which communicates only the required rows of factor matrices. This helped overcome memory limitations as well as reduce the communication cost dramatically. On top of this, employing effective hypergraph partitioning routines further enhanced scalability of HYPERTENSOR by incurring even less communication. A similar MPI parallelism approach using point-to-point messaging was also employed by SPLATT later on in its distributed memory parallelization [96]. Next, HYPERTENSOR added a parallel algorithm for computing the Tucker decomposition of sparse tensors with distributed as well as shared memory parallelism, and it is still the only high performance parallel software computing the Tucker decomposition of sparse tensors [51]. Finally, a dimension tree-based computational scheme for computing CP decomposition and an associated sparse tensor storage scheme with an effective shared memory parallelization was introduced to HYPERTENSOR [52]. This rendered HYPERTENSOR the fastest tensor factorization software both in shared and distributed memory, even though a very recent extension, called ADATM [69], also introduced the idea of storing and reusing partial results to SPLATT to reduce the computational cost.

In the following sections, we provide an overview of HYPERTENSOR and PACOS, which drives the partitioning and communication of HYPERTENSOR, alongside with some code samples.

8.2 HYPERTENSOR: A high performance parallel sparse tensor factorization library

HYPERTENSOR is the parallel sparse tensor library we developed during this thesis as the software outcome of the research work. It is a C++ implementation using OpenMP and MPI for shared and distributed memory parallelism. The communication and partitioning routines driving the distributed memory parallelization of HYPERTENSOR are later on extracted in a generic software package called PACOS in order to simplify the code as well as to enable developing likewise parallel applications.

HYPERTENSOR is fully templated to support single and double floating point numbers as well as 32- and 64-bit integers for tensor indexes. It can be compiled and run with and without MPI and OpenMP support. It does, however, require a compatible C++11 compiler.

The two principal routines provided by HYPERTENSOR are parallel CP-ALS and HOOI algorithms for sparse tensors. HOOI involves a parallel truncated SVD step, which is realized with the help of PETSC and SLEPC software packages. HYPERTENSOR employs an object-driven paradigm where all tensors and matrices are represented as objects with associated functions. Following are the main data structures used by HYPERTENSOR:

- **SpTensor:** Sparse tensor data structure, in coordinate or dimension tree format. It has member functions for computing CP and Tucker decompositions.
- **KTensor:** Kruskal tensor, obtained from a CP decomposition algorithm. It involves N dense factor matrices as well as a vector representing the column norms.
- **TTensor:** Tucker tensor, obtained from a Tucker decomposition algorithm. It consists of an N-dimensional dense core tensor and N dense factor matrices.
- **DMatrix:** Dense matrix data structure providing a wrapper for many BLAS, LAPACK and other matrix operations.

Tensor data structures are also accompanied by some utilities such as norm calculation, print functions, I/O routines for loading from and saving to disk, and many others. DMatrix provides a lightweight wrapper on top of dense matrix routines, with the principle goal of providing templated matrix routines to be employed by other templated functions.

In Figure 8.1, we provide a sample code snippet computing the CP and Tucker decomposition of a sparse tensor read from a file. Each MPI rank reads a part of a partitioned tensor along with a vector partition table for all dimensions. Both the partitioned tensor and the partition table are obtained in a preceding step involving the partitioner of PACOS. Next, the sparse tensor is loaded from the file, and the parameters of the tensor decomposition algorithms are set. Finally, executing the member CP-ALS and HOOI algorithms output Kruskal and Tucker tensors representing the CP and Tucker decompositions of the tensor with the provided rank of approximation.

```cpp
#include<hypertensor>

int main(int argc, char **argv) {

  auto tensorFileName = argv[1]; // File name of the sparse tensor
  auto dimPartitionFileName = argv[2]; // Partition file for dimensions
  auto ktensorFileName = argv[3]; // Output of the CP-ALS, a Kruskal tensor
  auto ttensorFileName = argv[4]; // Output of the HOOI, a Tucker tensor
  auto rank = atoi(argv[5]); // The rank of approximation

  HyperTensor_Init(&argc, &argv, MPI_COMM_WORLD); // Initialize the library

  // Load the tensor, form the communicators, convert global indices to local
  SpTensor<double> spten(tensorFileName, dimPartitionFileName);
  // Setup parameters for tensor decomposition algorithms
  int maxIters = 50;
  double tol = 1e-6;
  std::vector<int> ranks(spten._order, rank);
  // Run the CP-ALS and HOOI algorithms, and output the results to files
  auto kten = spten.cpAls(rank, maxIters, tol);
  kten.write(ktensorFileName);
  auto tten = spten.hooi(ranks, maxIters, tol);
  tten.write(ttensorFileName);

  HyperTensor_Finalize(); // Finalize the library

  return 0;
}
```

Figure 8.1: A HYPERTENSOR code snippet for computing the CP decomposition of a sparse tensor.

8.3 PACOS: A partitioning and communication framework for sparse irregular applications

PACOS is a library providing a unified partitioning and communication framework for developing parallel sparse (multi)linear algebra routines, including but not limited to sparse tensor factorization (CP, Tucker, tensor completion, etc.), sparse matrix factorization variants (PCA, non-negative), and algorithms involving sparse matrix-vector multiplication (iterative solvers, page rank, etc.).

The execution of a parallel algorithm using PACOS involves two steps. The first one is the partitioning step, in which nonzero elements of a sparse struct (tensor or matrix) as well as vector entries (or matrix rows) in each dimension are partitioned. It supports many partitioning routines for nonzero elements such as randomized checkerboard partitioning, checkerboard hypergraph partitioning, fine-grain hypergraph partitioning, and fine-grain random partitioning. It provides effective heuristics for partitioning vector entries in a way that balances the number of owned vector entries by processes as well as the amount of communication volume per process. Finally, it partitions the sparse struct accordingly, and creates per-process sparse struct files along with vector partition maps for all dimensions.

Once the sparse struct is partitioned, each process can read its data and vector mappings, and create a communicator for each dimension using PACOS, which is formed conformally with the partition created in the partitioning phase. Finally, each process can execute its computations on its local data, and call PACOS communication routines to carry out sends and receives at the synchronization points. PACOS provides *fold* and *expand* communication routines enabling a fine-grain algorithm, whereas coarse-grain variants can similarly be developed by skipping one of the two communication routines. PACOS employs non-blocking MPI routines underneath, giving the possibility of overlapping communication with local computations. It also uses a lightweight MPI wrapper that provides templating and 64-bit integer compatibility with MPI.

Indeed, there are other software packages providing similar functionalities, ZOLTAN and PETSC to name a few, yet the main convenience of PACOS is providing a unified partitioning and communication framework geared towards matrix and tensor factorization variants, and giving a drop-in support for effective nonzero and vector partitioning routines through hypergraph partitioning. It is owing to this functionality that we were able to replace the entire set of partitioning and communication routines of NMFLIBRARY with PACOS using less than 100 lines of code modifications, and thereby achieve scalability up to 32K cores on an IBM BlueGene/Q supercomputer, which were simply impossible with its existing parallelization scheme.

To illustrate the usage, in Figure 8.2, we provide a PACOS code snippet performing iterative TTMVs in an alternating manner over the dimensions. Note that this concise code is the backbone of most iterative tensor factorization algorithms. PACOS effectively handles all the communications underneath, and also provides the potential to overlap computations with communications.

```
#include<hypertensor>

int main(int argc, char **argv) {

  Pacos_Init(&argc, &argv, MPI_COMM_WORLD);

  auto tensorFileName = argv[1];
  auto dimPartitionFileName = argv[2];
  // Load the tensor
  Pacos_SparseStruct ss(tensorFileName);
  // Load partition data for dimensions
  std::vector<std::vector<Pacos_IntPair>> dimPart;
  Pacos_Communicator<double>::loadDistributedDimPart(dimPartFileName, dimPart);
  // Set up the communicators and local vectors
  std::vector<Pacos_Communicator<double>> dimComm; // Communicators
  std::vector<std::vector<double>> vecs; // Vectors
  for (Pacos_Int dim = 0; dim < ss.order; dim++) {
    // Create the communicator for the current dim,
    // and replace global tensor indices with local
    dimComm.emplace_back(MPI_COMM_WORLD, ss.idx[dim], dimPart[dim]);
    // Create the local vector
    vecs.emplace_back(dimComm[dim].localRowCount());
  }
  for (Pacos_Int iter = 0; iter < 10; iter++) {
    for (Pacos_Int dim = 0; dim < ss.order; dim++) {
      // Perform TTMV using local tensor
      computeTtmv(ss, vecs, dim);
      // Send/receive partial results
      dimComm[dim].foldCommBegin(vecs[dim]);
      // Overlapping communication and computation is possible here...
      dimComm[dim].foldCommEnd(vecs[dim]);
      // Perform further computations after TTMV,
      // then modify ``owned'' local vector entries
      computePostTtv(...)
      // Send/receive the finalized vector entries
      dimComm[dim].expandCommBegin(vecs[dim]);
      // Overlapping communication and computation is possible here...
      dimComm[dim].expandCommEnd(vecs[dim]);
    }
  }

  Pacos_Finalize();

  return 0;
}
```

Figure 8.2: A PACOS code snippet for performing TTMVs in parallel in an iterative manner.

Chapter 9

Conclusion

In this thesis, we have investigated efficient algorithms to compute tensor decompositions for both sparse and dense tensors. We have achieved this efficiency through shared and distributed memory parallelism, effective partitioning and load balancing methods for a scalable parallel execution, memory efficient sparse tensor representations, and a new computational scheme asymptotically reducing the computational cost of tensor decomposition algorithms.

We employed most of these techniques for computing the CP decomposition of high dimensional sparse tensors in Chapter 3, which serves as the "porte-parole" for the subsequent chapters with the diversity of techniques involved in the chapter. First, in this chapter, we introduce a novel computational scheme using a dimension tree structure to reduce the number of TTV operations within a CP-ALS iteration by a factor of $O(N/\log N)$ for an N-dimension tensor. We show that this approach provides significant computational gains even for 3- and 4-dimensional tensors, and these gains only improve as the tensor dimensionality grows. To effectively carry out sparse tensor operations, we provided a sparse tensor data structure associated with a dimension tree, which similarly reduces the number of index arrays used to store the tensor to $O(N \log N)$ instead of $O(N^2)$, which is found in some state-of-the-art implementations. We give a fast shared memory parallel algorithm using this data structure and computational scheme. Finally, we perform an effective parallelization of this algorithm in distributed memory, which also benefits from novel hypergraph partitioning techniques for establishing load balance while minimizing communication requirements. The work in this chapter is pioneer in establishing the bridge between sparse linear and multilinear algebra by adopting and generalizing fundamental parallelization and partitioning techniques in the existing sparse matrix literature to parallel sparse tensor factorization, and thereby sets the bar for the state-of-the-art in the field.

In Chapter 4, we present a shared memory parallel algorithm for computing a nonzero CP decomposition. This introduces a constraint on factor matrices to not involve any numeric zeros, which in turn enables a very fast computation of MTTKRP (or equivalently, TTV) operations for high dimensional sparse tensors. Specifically, the new method amounts to performing $O(N)$ TTVs in a CP-ALS iteration, whereas the existing methods require $O(N^2)$ TTVs, and even our dimension tree-based approach takes $O(N \log N)$ TTVs per iteration. Also, this method performs significantly less precomputation to set up tensor data structures, which provides, for an N-dimensional tensor having k nonzero entries, $O(\log k)$ and $O(\log N \log k)$ faster preprocessing in compare to SPLATT and our dimension tree-based method. We also discuss load balancing strategies and memory initialization

techniques to obtain high performance on a NUMA parallel architecture. In overall, in comparison to the state of the art, this yields a parallel algorithm achieving up to 16.7x speedup in sequential and 10.5x speedup in parallel executions, requiring up to 24x less data preprocessing time, and using up to $O(\log N)$ less memory for storing intermediate computations.

Chapter 5 adopts our parallelization techniques for computing CP decomposition to the computation of the Tucker decomposition for sparse tensors. In this chapter, we introduce a coarse- and a fine-grain algorithm for computing the Tucker of sparse tensors. The coarse-grain variant avoids an expensive *fold* communication step whereas limiting the possibility for a good partition, whereas the fine-grain variant provides the most flexibility in partitioning for establishing load balance and reducing communication costs, but incurs the fold communication step. To render the fine-grain algorithm amenable, we couple the fold communication with an iterative truncated SVD algorithm, which effectively reduces the unit communication cost of the fold communication from R^{N-1} to K where K is the number of iterations of the truncated SVD algorithm in computing in a rank-R Tucker decomposition. Finally, we similarly employ hypergraph partitioning to significantly reduce the number of unit communications required to execute the parallel algorithm. All these techniques yield scalability up to 4096 cores on an IBM BlueGene/Q supercomputer using real-world sparse tensors.

In Chapter 6, we propose a distributed memory parallelization of the non-negative matrix factorization problem, which shares a great amount of similarities with the CP decomposition in 2 dimensions. Our algorithm is the first to use a fine-grain parallelism and a point-to-point messaging scheme. We perform an MPI-only parallelization of the existing implementation in the NMFLIBRARY, and use 2D partitionings that eliminate the need for intra-node communication in one dimension. We similarly used appropriate hypergraph models to reduce communication with an effective partitioning. All these contributions yield a parallel algorithm achieving scalability up to 32768 cores on an IBM BlueGene/Q supercomputer.

We investigate the use of dimension trees in computing CP and Tucker decompositions of dense tensors Chapter 7. We provide the dimension tree-based CP-ALS and HOOI algorithms for dense tensors, and show many theoretical results related to NP-completeness of finding an optimal binary tree in both cases. We also introduce a greedy algorithm that finds the optimal ordering for a series of TTMs, which constitutes the most expensive step in HOOI algorithm.

In the course of this thesis, we developed two software packages, namely HYPERTENSOR and PACOS, that address the challenges in high performance parallel computations of sparse CP and Tucker decompositions, which we detailed in Chapter 8. HYPERTENSOR uses efficient tree-based sparse tensor data structures and computational schemes to execute CP-ALS and HOOI algorithms. It uses OpenMP for shared memory parallelization, and depends on PACOS for distributed memory parallelization and partitioning routines. PACOS provides a generic communication and partitioning framework for sparse (multi)linear algebra and machine learning applications, including but not limited to tensor factorization, matrix factorization, iterative solvers, and iterative graph algorithms. With these two software packages, direct outcomes of this thesis research, we provide the fastest available sparse tensor factorization software as of today.

Altogether, with all algorithmic contributions, parallelization techniques, and optimized software implementations, the work in this thesis enables performing tensor- and matrix-based analysis of big data problems having billions of entries in a matter of seconds on modern HPC platforms, and sets a new bar for the state of the art in the domain of high performance parallel tensor computations.

Perspectives

Even though we have not focused on it in this thesis, there has been recent work on the parallelization of tensor factorization on accelerators, including GPUs and Xeon Phis. These methods use the traditional representations of sparse tensors and employ tiling strategies to better utilize the available high-bandwidth memory of the accelerator. Our tree-based tensor storage and computational schemes provide significant computational advantages over the traditional techniques, and require non-trivial techniques to be adapted for the accelerator use. This extension would be a natural future direction for our work.

In the longer term, there are still computational challenges left to be addressed in tensor decompositions. HPC techniques for computing non-negative tensor factorization and coupled tensor-matrix and -tensor factorizations are still to be tackled, which might find significant value in modern big data applications. From an application perspective, the work in this thesis renders tensor decomposition methods amenable to use on datasets of massive scale, which was simply impractical before. As a result, we hope the outcome of this thesis to significantly expand the applications of tensor decompositions on big data analysis and machine learning problems including but not limited to graph analysis, web link analysis, healthcare data analytics, and recommender systems, in the years to come.

Acknowledgements

The work in this thesis is performed using the compute resources generously provided by ENS de Lyon (workstations and PSMN cluster), IDRIS Grants (i2016067501) for Ada and Turing supercomputers, and Rhea cluster at the Oak Ridge Leadership Computing Facility (OLCF).

Publications[1]

International conferences

[1] O. KAYA, *A parallel nonzero CP decomposition algorithm for higher order sparse data analysis*, in Proceedings of the Seventh International Conference on Advanced Communications and Computation, INFOCOMP '17, June 2017.

[2] 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, New York, NY, USA, 2015, ACM, pp. 77:1–77:11.

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

International journals

[1] O. KAYA, Y. ROBERT, AND B. UÇAR, *Computing dense tensor decompositions using dimension trees*, Submitted to Linear Algebra and its Applications on July 14, '17, (2017).

[2] O. KAYA AND B. UÇAR, *Parallel CP decomposition of sparse tensors using dimension trees*, Submitted to SIAM Journal on Scientific Computing on Nov. 8, '16, revised on Jun 30, '17, (2017).

In submission

[1] O. KAYA, R. KANNAN, G. BALLARD, AND H. PARK, *Distributed sparse non-negative matrix factorization*, (2017).

[1] Authors are listed alphabetically, except for 'in submission' 1.

Bibliography

[1] E. ACAR, D. M. DUNLAVY, AND T. G. KOLDA, *A scalable optimization approach for fitting canonical tensor decompositions*, Journal of Chemometrics, 25 (2011), pp. 67–86.

[2] A. C. R. Z. S. AMARI, *Hierarchical ALS algorithms for nonnegative matrix and 3d tensor factorization*, in Independent Component Analysis and Signal Separation: 7th International Conference, ICA 2007, London, UK, September 9-12, 2007. Proceedings, Springer Berlin Heidelberg, 2007, pp. 169–176.

[3] C. A. ANDERSSON AND R. BRO, *The N-way toolbox for MATLAB*, Chemometrics and Intelligent Laboratory Systems, 52 (2000), pp. 1–4.

[4] C. J. APPELLOF AND E. R. DAVIDSON, *Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents*, Analytical Chemistry, 53 (1981), pp. 2053–2056.

[5] W. AUSTIN, G. BALLARD, AND T. G. KOLDA, *Parallel tensor compression for large-scale scientific data*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Chicago, IL, USA, May 23–27, IPDPS '16, 2016, pp. 912–922.

[6] B. W. BADER AND T. G. KOLDA, *Efficient MATLAB computations with sparse and factored tensors*, SIAM Journal on Scientific Computing, 30 (2007), pp. 205–231.

[7] B. W. BADER, T. G. KOLDA, ET AL., *Matlab tensor toolbox version 2.6*. Available online, February 2015.

[8] 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.

[9] M. BASKARAN, B. MEISTER, N. VASILACHE, AND R. LETHIN, *Efficient and scalable computations with sparse tensors*, in Proceedings of the IEEE Conference on High Performance Extreme Computing, HPEC 2012, Sept 2012, pp. 1–6.

[10] M. M. BASKARAN, B. MEISTER, AND R. LETHIN, *Low-overhead load-balanced scheduling for sparse tensor computations*, in Proceedings of the IEEE Conference on High Performance Extreme Computing, HPEC 2014, Waltham, MA, USA, Sept. 2014, IEEE, pp. 1–6.

[11] J. BENNETT AND S. LANNING, *The netflix prize*, in Proceedings of the KDD Cup and Workshop, vol. 2007, 2007, p. 35.

[12] D. P. BERTSEKAS, *Nonlinear programming*, Athena Scientific, 1999.

[13] S. BIRD, E. LOPER, AND E. KLEIN, *Natural Language Processing with Python*, O'Reilly Media Inc., 2009.

[14] J. BUURLAGE, *Self-improving sparse matrix partitioning and bulk-synchronous pseudo-streaming*, Master's thesis, Utrecht University, 2016.

[15] A. CARLSON, J. BETTERIDGE, B. KISIEL, B. SETTLES, E. R. H. JR., AND T. M. MITCHELL, *Toward an architecture for never-ending language learning*, in Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI '10, AAAI Press, 2010, pp. 1306–1313.

[16] D. J. CARROLL AND J. CHANG, *Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition*, Psychometrika, 35 (1970), pp. 283–319.

[17] 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.

[18] Ü. V. ÇATALYÜREK AND C. AYKANAT, *A hypergraph-partitioning approach for coarse-grain decomposition*, in Proceedings of the 2011 ACM/IEEE Conference on Supercomputing, Denver, Colorado, 2001, p. 42.

[19] Ü. V. ÇATALYÜREK, C. AYKANAT, AND B. UÇAR, *On two-dimensional sparse matrix partitioning: Models, methods, and a recipe*, SIAM Journal on Scientific Computing, 32 (2010), pp. 656–683.

[20] Ü. V. ÇATALYÜREK, *Hypergraph Models for Sparse Matrix Partitioning and Reordering*, PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.

[21] E. C. CHI AND T. G. KOLDA, *On tensors, sparsity, and nonnegative factorizations*, SIAM Journal on Matrix Analysis and Applications, 33 (2012), pp. 1272–1299.

[22] J. H. CHOI AND S. V. N. VISHWANATHAN, *DFacTo: Distributed factorization of tensors*, in 27th Advances in Neural Information Processing Systems, Montreal, Quebec, Canada, 2014, pp. 1296–1304.

[23] A. CICHOCKI, D. MANDIC, L. D. LATHAUWER, G. ZHOU, Q. ZHAO, C. CAIAFA, AND H. A. PHAN, *Tensor decompositions for signal processing applications: From two-way to multiway component analysis*, IEEE Signal Processing Magazine, 32 (2015), pp. 145–163.

[24] A. CICHOCKI, R. ZDUNEK, A. H. PHAN, AND S.-I. AMARI, *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation*, Wiley, 2009.

[25] O. DEBALS, M. V. BAREL, AND L. D. LATHAUWER, *Nonnegative matrix factorization using nonnegative polynomial approximations*, IEEE Signal Processing Letters, 24 (2017), pp. 948–952.

[26] 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 (2009), pp. 248–271.

[27] C. FALOUTSOS, A. BEUTEL, E. P. XING, E. E. PAPALEXAKIS, A. KUMAR, AND P. P. TALUKDAR, *Flexi-FaCT: Scalable flexible factorization of coupled tensors on Hadoop*, in Proceedings of the 2014 SIAM International Conference on Data Mining, SDM14, 2014, pp. 109–117.

[28] A. GITTENS, A. DEVARAKONDA, E. RACAH, M. F. RINGENBURG, L. GERHARDT, J. KOTTALAM, J. LIU, K. J. MASCHHOFF, S. CANON, J. CHHUGANI, P. SHARMA, J. YANG, J. DEMMEL, J. HARRELL, V. KRISHNAMURTHY, M. W. MAHONEY, AND PRABHAT, *Matrix factorization at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies*, in Proceedings of the IEEE International Conference on Big Data, IEEE BigData 2016, Dec 2016, pp. 204–213.

[29] S. GOREINOV, E. TYRTYSHNIKOV, AND N. ZAMARASHKIN, *A theory of pseudoskeleton approximations*, Linear Algebra and its Applications, 261 (1997), pp. 1 – 21.

[30] 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, Berkeley, CA, USA, 2008, USENIX Association, p. 19.

[31] L. GRASEDYCK, *Hierarchical singular value decomposition of tensors*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 2029–2054.

[32] D. GROVE, J. MILTHORPE, AND O. TARDIEU, *Supporting array programming in X10*, in Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY'14, 2014, pp. 38:38–38:43.

[33] N. GUAN, D. TAO, Z. LUO, AND B. YUAN, *Nenmf: An optimal gradient method for nonnegative matrix factorization*, IEEE Transactions on Signal Processing, 60 (2012), pp. 2882–2898.

[34] N. HALKO, P. G. MARTINSSON, AND J. A. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), pp. 217–288.

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

[36] J. HÅSTAD, *Tensor rank is np-complete*, Journal of Algorithms, 11 (1990), pp. 644 – 654.

[37] V. HENNE, *Label propagation for hypergraph partitioning*, Master's thesis, Karsruhe Institute of Technology, Germany, 2015.

[38] P. O. HOYER, *Non-negative matrix factorization with sparseness constraints*, JMLR, 5 (2004), pp. 1457–1469.

[39] I. JEON, E. E. PAPALEXAKIS, U. KANG, AND C. FALOUTSOS, *Haten2: Billion-scale tensor decompositions*, in IEEE 31st International Conference on Data Engineering, ICDE 2015, 2015, pp. 1047–1058.

[40] 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, New York, NY, USA, 2012, ACM, pp. 316–324.

[41] R. KANNAN, G. BALLARD, AND H. PARK, *A high-performance parallel algorithm for nonnegative matrix factorization*, in Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, New York, NY, USA, February 2016, ACM, pp. 9:1–9:11.

[42] R. KANNAN, G. BALLARD, AND H. PARK, *MPI-FAUN: an MPI-based framework for alternating-updating nonnegative matrix factorization*, CoRR, abs/1609.09154 (2016).

[43] L. KARLSSON, D. KRESSNER, AND A. USCHMAJEW, *Parallel algorithms for tensor completion in the CP format*, Parallel Computing, 57 (2016), pp. 222–234.

[44] G. KARYPIS AND V. KUMAR, *Multilevel algorithms for multi-constraint hypergraph partitioning*, Tech. Rep. 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998.

[45] K. KAYA, F. H. ROUET, AND B. UÇAR, *On partitioning problems with complex objectives*, in Euro-Par 2011: Parallel Processing Workshops, vol. 7155 of LNCS, Springer Berlin / Heidelberg, 2012, pp. 334–344.

[46] O. KAYA, *A parallel nonzero CP decomposition algorithm for higher order sparse data analysis*, in Proceedings of the Seventh International Conference on Advanced Communications and Computation, INFOCOMP '17, June 2017.

[47] O. KAYA, R. KANNAN, G. BALLARD, AND H. PARK, *Distributed sparse non-negative matrix factorization*, (2017).

[48] O. KAYA, Y. ROBERT, AND B. UÇAR, *Computing dense tensor decompositions using dimension trees*, Submitted to Linear Algebra and its Applications on July 14, '17, (2017).

[49] O. KAYA AND B. UÇAR, *High-performance parallel algorithms for the Tucker decomposition of higher order sparse tensors*, Tech. Rep. RR-8801, Inria, Grenoble – Rhône-Alpes, Oct 2015.

[50] 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, New York, NY, USA, 2015, ACM, pp. 77:1–77:11.

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

[52] ——, *Parallel CP decomposition of sparse tensors using dimension trees*, Research Report RR-8976, Inria - Research Centre Grenoble – Rhône-Alpes, Nov. 2016.

[53] ——, *Parallel CP decomposition of sparse tensors using dimension trees*, Submitted to SIAM Journal on Scientific Computing on Nov. 8, '16, revised on Jun 30, '17, (2017).

[54] 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 (2003), pp. 119–125.

[55] H. KIM AND H. PARK, *Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis*, Bioinformatics, 23 (2007), pp. 1495–1502.

[56] J. KIM, Y. HE, AND H. PARK, *Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework*, Journal of Global Optimization, 58 (2014), pp. 285–319.

[57] T. G. KOLDA AND B. BADER, *The TOPHITS model for higher-order web link analysis*, in Proceedings of Link Analysis, Counterterrorism and Security, 2006, pp. 26–29.

[58] ——, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.

[59] T. G. KOLDA AND J. SUN, *Scalable tensor decompositions for multi-aspect data mining*, in Proceedings of the 8th IEEE International Conference on Data Mining, ICDM 2008, Pisa, Italy, 2008, pp. 363–372.

[60] A. KRISHNAMURTHY AND A. SINGH, *Low-rank matrix and tensor completion via adaptive sampling*, in Proceedings of the 26th International Conference on Neural Information Processing Systems, NIPS'13, 2013, pp. 836–844.

[61] D. KUANG, C. DING, AND H. PARK, *Symmetric nonnegative matrix factorization for graph clustering*, in Proceedings of the SIAM Conference on Data Mining, SDM 2012, 2012, pp. 106–117.

[62] D. KUANG, S. YUN, AND H. PARK, *SymNMF: Nonnegative low-rank approximation of a similarity matrix for graph clustering*, Journal of Global Optimization, (2013), pp. 1–30.

[63] L. D. LATHAUWER AND B. D. MOOR, *From matrix to tensor: Multilinear algebra and signal processing*, in Proceedings of the Institute of Mathematics and Its Applications Conference Series, vol. 67, 1998, pp. 1–16.

[64] L. D. LATHAUWER, B. D. MOOR, AND J. VANDEWALLE, *A multilinear singular value decomposition*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1253–1278.

[65] ——, *On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1324–1342.

[66] T. LENGAUER, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley–Teubner, Chichester, U.K., 1990.

[67] J. LESKOVEC AND A. KREVL, *SNAP Datasets: Stanford large network dataset collection*. http://snap.stanford.edu/data, June 2014.

[68] 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, Austin, Texas, 2015, ACM, New York, NY, USA, pp. 76:1–76:12.

[69] J. LI, J. CHOI, I. PERROS, J. SUN, AND R. VUDUC, *Model-driven sparse CP decomposition for higher-order tensors*, in Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, IPDPS '17, Orlando, FL, USA, May 2017, pp. 1048–1057.

[70] R. LIAO, Y. ZHANG, J. GUAN, AND S. ZHOU, *CloudNMF: A MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets*, Genomics, proteomics & bioinformatics, 12 (2014), pp. 48–51.

[71] C. LIU, H.-C. YANG, J. FAN, L.-W. HE, AND Y.-M. WANG, *Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce*, in Proceedings of the WWW, ACM, 2010, pp. 681–690.

[72] Y. LOW, D. BICKSON, J. GONZALEZ, C. GUESTRIN, A. KYROLA, AND J. M. HELLERSTEIN, *Distributed GraphLab: A framework for machine learning and data mining in the cloud*, Proc. VLDB Endow., 5 (2012), pp. 716–727.

[73] 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 2011, July 2011, pp. 203–210.

[74] E. MEJÍA-ROA, D. TABAS-MADRID, J. SETOAIN, C. GARCÍA, F. TIRADO, AND A. PASCUAL-MONTANO, *NMF-mGPU: Non-negative matrix factorization on multi-GPU systems*, BMC bioinformatics, 16 (2015), p. 43.

[75] X. MENG, J. BRADLEY, B. YAVUZ, E. SPARKS, S. VENKATARAMAN, D. LIU, J. FREEMAN, D. B. TSAI, M. AMDE, S. OWEN, D. XIN, R. XIN, M. J. FRANKLIN, R. ZADEH, M. ZAHARIA, AND A. TALWALKAR, *MLlib: Machine Learning in Apache Spark*, May 2015.

[76] C. NG, M. BARKETAU, T. CHENG, AND M. Y. KOVALYOV, *Product partition and related problems of scheduling and systems reliability: Computational complexity and approximation*, European Journal of Operational Research, 207 (2010), pp. 601 – 604.

[77] D. NION, K. N. MOKIOS, N. D. SIDIROPOULOS, AND A. POTAMIANOS, *Batch and adaptive PARAFAC-based blind separation of convolutive speech mixtures*, IEEE Transactions on Audio, Speech, and Language Processing, 18 (2010), pp. 1193–1207.

[78] D. NION AND N. D. SIDIROPOULOS, *Tensor algebra and multidimensional harmonic retrieval in signal processing for mimo radar*, IEEE Transactions on Signal Processing, 58 (2010), pp. 5693–5705.

[79] E. E. PAPALEXAKIS, C. FALOUTSOS, AND N. D. SIDIROPOULOS, *ParCube: Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition*, ACM Transactions on Knowledge Discovery from Data, 10 (2015), pp. 3:1–3:25.

[80] V. P. PAUCA, F. SHAHNAZ, M. W. BERRY, AND R. J. PLEMMONS, *Text mining using non-negative matrix factorizations*, in Proceedings of SDM, 2004.

[81] I. PERROS, R. CHEN, R. VUDUC, AND J. SUN, *Sparse hierarchical Tucker factorization and its application to healthcare*, in Proceedings of the 2015 IEEE International Conference on Data Mining, ICDM 2015, Nov 2015, pp. 943–948.

[82] A. H. PHAN, P. TICHAVSKÝ, AND A. CICHOCKI, *Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations*, IEEE Transactions on Signal Processing, 61 (2013), pp. 4834–4846.

[83] A. PINAR AND C. AYKANAT, *Fast optimal load balancing algorithms for 1D partitioning*, Journal of Parallel and Distributed Computing, 64 (2004), pp. 974 – 996.

[84] A. PINAR AND B. HENDRICKSON, *Partitioning for complex objectives*, in Proceedings of the 15th International Parallel & Distributed Processing Symposium, IPDPS '01, Washington, DC, USA, 2001, IEEE Computer Society, p. 121.

[85] S. RENDLE AND T. S. LARS, *Pairwise interaction tensor factorization for personalized tag recommendation*, in Proceedings of the Third ACM International Conference on Web Search and Data Mining, WSDM '10, New York, NY, USA, 2010, ACM, pp. 81–90.

[86] S. RENDLE, B. M. LEANDRO, A. NANOPOULOS, AND L. SCHMIDT-THIEME, *Learning optimal ranking with tensor factorization for tag recommendation*, in Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09, New York, NY, USA, 2009, ACM, pp. 727–736.

[87] J. E. ROMAN, C. CAMPOS, E. ROMERO, AND A. TOMAS, *SLEPc users manual*, Tech. Rep. DSIC-II/24/02 - Revision 3.6, D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2015.

[88] C. SANDERSON, *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments*, tech. rep., NICTA, 2010.

[89] N. SATISH, N. SUNDARAM, M. M. A. PATWARY, J. SEO, J. PARK, M. A. HASSAAN, S. SENGUPTA, Z. YIN, AND P. DUBEY, *Navigating the maze of graph analytics frameworks using massive graph datasets*, in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, 2014, pp. 979–990.

[90] D. SEUNG AND L. LEE, *Algorithms for non-negative matrix factorization*, NIPS, 13 (2001), pp. 556–562.

[91] N. D. SIDIROPOULOS, R. BRO, AND G. B. GIANNAKIS, *Parallel factor analysis in sensor array processing*, IEEE Transactions on Signal Processing, 48 (2000), pp. 2377–2388.

[92] N. D. SIDIROPOULOS, L. D. LATHAUWER, X. FU, K. HUANG, E. E. PAPALEXAKIS, AND C. FALOUTSOS, *Tensor decomposition for signal processing and machine learning*, IEEE Transactions on Signal Processing, 65 (2017), pp. 3551–3582.

[93] G. M. SLOTA, K. MADDURI, AND S. RAJAMANICKAM, *PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks*, in Proceedings of the 2014 IEEE International Conference on Big Data, IEEE BigData 2014, Oct 2014, pp. 481–490.

[94] S. SMITH AND G. KARYPIS, *DMS: Distributed sparse tensor factorization with alternating least squares*, Tech. Rep. 15-007, Department of Computer Science and Engineering, University of Minnesota, May 2015.

[95] ——, *Tensor-matrix products with a compressed sparse tensor*, in Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3 2015, ACM, 2015, p. 7.

[96] S. SMITH AND G. KARYPIS, *A medium-grained algorithm for sparse tensor factorization*, in 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, 2016, pp. 902–911.

[97] S. SMITH, J. PARK, AND G. KARYPIS, *An exploration of optimization algorithms for high performance tensor completion*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 31:1–31:13.

[98] S. SMITH, N. RAVINDRAN, N. D. SIDIROPOULOS, AND G. KARYPIS, *SPLATT: Efficient and parallel sparse tensor-matrix multiplication*, in Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium, IPDPS '15, Hyderabad, India, May 2015, IEEE Computer Society, pp. 61–70.

[99] L. SORBER, M. V. BAREL, AND L. D. LATHAUWER, *Optimization-based algorithms for tensor decompositions: Canonical polyadic decomposition, decomposition in rank-$(l_r,l_r,1)$ terms, and a new generalization*, SIAM Journal on Optimization, 23 (2013), pp. 695–720.

[100] S. R. SUKUMAR, R. KANNAN, S. LIM, AND M. A. MATHESON, *Kernels for scalable data analysis in science: Towards an architecture-portable future*, in 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016, 2016, pp. 1026–1031.

[101] S. R. SUKUMAR, M. A. MATHESON, R. KANNAN, AND S. LIM, *Mini-apps for high performance data analysis*, in Proceedings of the IEEE International Conference on Big Data, IEEE BigData 2016, 2016, pp. 1483–1492.

[102] D. L. SUN AND C. FÉVOTTE, *Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence*, in Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, May 2014, pp. 6201–6205.

[103] P. SYMEONIDIS, A. NANOPOULOS, AND Y. MANOLOPOULOS, *Tag recommendations based on tensor dimensionality reduction*, in Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys '08, New York, NY, USA, 2008, ACM, pp. 43–50.

[104] J. UGANDER AND L. BACKSTROM, *Balanced label propagation for partitioning massive graphs*, in Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM '13, New York, NY, USA, 2013, ACM, pp. 507–516.

[105] M. A. O. VASILESCU AND D. TERZOPOULOS, *Multilinear analysis of image ensembles: TensorFaces*, in Computer Vision—ECCV 2002, Springer, 2002, pp. 447–460.

[106] N. VERVLIET, O. DEBALS, AND L. D. LATHAUWER, *Tensorlab 3.0 - numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization*, in 2016 50th Asilomar Conference on Signals, Systems and Computers, Nov 2016, pp. 1733–1738.

[107] N. VERVLIET, O. DEBALS, L. SORBER, AND L. D. LATHAUWER, *Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis*, IEEE Signal Processing Magazine, 31 (2014), pp. 71–79.

[108] N. VERVLIET AND L. D. LATHAUWER, *A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors*, IEEE Journal of Selected Topics in Signal Processing, 10 (2016), pp. 284–295.

[109] Z. XIANYI, *OpenBLAS*, Last Accessed 03-Dec-2015.

[110] Y. XU, L. ZHANG, AND W. LIU, *Cubic analysis of social bookmarking for personalized recommendation*, in Frontiers of WWW Research and Development-APWeb 2006, Springer, 2006, pp. 733–738.

[111] J. YIN, L. GAO, AND Z. ZHANG, *Scalable nonnegative matrix factorization with block-wise updates*, in Proceedings of the Machine Learning and Knowledge Discovery in Databases, vol. 8726 of LNCS, 2014, pp. 337–352.

[112] H. YUN, H.-F. YU, C.-J. HSIEH, S. VISHWANATHAN, AND I. DHILLON, *Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion*, Proceedings of the VLDB Endowment, 7 (2014), pp. 975–986.

[113] M. ZAHARIA, M. CHOWDHURY, M. J. FRANKLIN, S. SHENKER, AND I. STOICA, *Spark: Cluster computing with working sets*, in Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, USENIX Association, 2010, pp. 10–10.