NOVEL ALGORITHMS AND MODELS FOR SCALING PARALLEL SPARSE TENSOR AND MATRIX FACTORIZATIONS

A DISSERTATION SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

By

Nabil F. T. Abubaker

July 2022

Novel Algorithms and Models for Scaling Parallel Sparse Tensor and Matrix Factorizations
By Nabil F. T. Abubaker
July 2022

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Cevdet Aykanat(Advisor)

Özcan Öztürk

Murat Manguoğlu

Bora Uçar

Fahreddin Şükrü Torun

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

Copyright Information

ABSTRACT

NOVEL ALGORITHMS AND MODELS FOR SCALING PARALLEL SPARSE TENSOR AND MATRIX FACTORIZATIONS

Nabil F. T. Abubaker
Ph.D. in Computer Engineering
Advisor: Cevdet Aykanat
July 2022

Two important and widely-used factorization algorithms, namely CPD-ALS for sparse tensor decomposition and distributed stratified SGD for low-rank matrix factorization, suffer from limited scalability. In CPD-ALS, the computational load associated with a tensor/subtensor assigned to a processor is a function of the nonzero counts as well as the fiber counts of the tensor when the CSF storage is utilized. The tensor fibers fragment as a result of nonzero distributions, which makes balancing the computational loads a hard problem. Two strategies are proposed to tackle the balancing problem on an existing fine-grain hypergraph model: a novel weighting scheme to cover the cost of fibers in the true load as well as an augmentation to the hypergraph with fiber nets to encode reducing the increase in computational load. CPD-ALS also suffers from high latency overhead due to the high number of point-to-point messages incurred as the processor count increases. A framework is proposed to limit the number of messages to $\mathcal{O}(\log_2 K)$, for a K-processor system, exchanged in $\log_2 K$ stages. A hypergraph-based method is proposed to encapsulate the communication of the new $\log_2 K$-stage algorithm. In the existing stratified SGD implementations, the volume of communication is proportional to one of the dimensions of the input matrix and prohibits the scalability. Exchanging the essential data necessary for the correctness of the SSGD algorithm as point-to-point messages is proposed to reduce the volume. This, although invaluable for reducing the bandwidth overhead, would increase the upper bound on the number of exchanged messages from $\mathcal{O}(K)$ to $\mathcal{O}(K^2)$ rendering the algorithm latency-bound. A novel Hold-and-Combine algorithm is proposed to exchange the essential communication volume with up to $\mathcal{O}(K \log K)$ messages. Extensive experiments on HPC systems demonstrate the importance of the proposed algorithms and models in scaling CPD-ALS and stratified SGD.

Keywords: Parallel Algorithms, Combinatorial Algorithms, HPC, Tensor Decomposition, Matrix Completion, Hypergraph partitioning, Communication cost minimization.

ÖZET

PARALEL SEYREK TENSÖR VE MATRIS AYRIŞIMI IÇIN YENI YÖNTEM VE MODELLER

Nabil F. T. Abubaker
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Cevdet Aykanat
Temmuz 2022

Yaygın olarak kullanılan iki önemli paralel ayrışım algoritmaları, seyrek tensör ayışımı için CPD-ALS ve düşük kerteli seyrek matris ayrışımı için dağıtık tabakalı olasılıksal gradyan alçalma (SGD), ölçeklenebilirlik anlamında zayıf kalmaktadır. CPD-ALS algoritmasında bir işlemciye atanan bir tensör/alt-tensör ile ilişkili hesaplamasal yük, CSF veri yapısı kullanıldığında tensörün sıfırdışı girdilerinin sayısı ve ayrıca tensörün fiber sayılarının bir fonksiyonudur. Tensör fiberleri, sıfırdışı girdilerin bölümlenmesine bağlı olarak parçalanır, bu da işlemcilerin hesaplamasal yüklerini dengelemeyi zor bir problem haline getirir. Bu problemin çözümü için mevcut bir ince taneli hiperçizge modeli üzerine iki yeni strateji önerilmiştir: fiber yüklerini de hesaplayarak gerçek yükü modelleyen özgün bir ağırlıklandırma şeması ve hesaplamasal yükteki artışı azaltmayı hedefleyen yeni fiber hiperkenarlarının hiperçizgeye eklenmesi.CPD-ALS ayrıca işlemci sayısı arttıkça artan gereken çok sayıda işlemciler arası doğrudan mesaj nedeniyle yüksek gecikim maliyeti ortaya çıkarmaktadır. Bu mesajların sayısını, K işlemcili bir bilgisayar için $\mathcal{O}(\log_2 K)$ ile limitleyen ve $\log_2 K$ aşamada gerçekleyen bir yaklaşım önermekteyiz. Ayrıca, bu yeni yaklaşımın gerektirdiği iletişimi modelleyen bir hiperçizge tabanlı bölümleme yöntemi önermekteyiz. Mevcut tabakalı SGD (SSGD) uygulamalarında, iletişim hacmi girdi matrisinin boyutlarından biri ile orantılıdır ve ölçeklenebilirliği engeller. İletişim hacmini azaltmak için SSGD algoritmasının doğruluğu için gerekli olan temel verilerin işlemciler arası doğrudan mesajlar ile değiş tokuş edilmesi önerilmiştir. Bu yöntem, iletişim hacmini azaltmak için paha biçilmez olsa da, mesaj sayısının üst sınırını $\mathcal{O}(K)$'dan $\mathcal{O}(K^2)$'ye artırarak algoritmayı gecikim maliyetlerine bağlı hale getirmektedir. Sadece temel verinin iletişimini $\mathcal{O}(K \log K)$ mesaj ile değiş tokuş eden yeni bir Tut-ve-Birleştir algoritması önerilmiştir. Yüksek başarımlı hesaplama sistemleri üzerinde gerçekleştirilen kapsamlı deneyler, CPD-ALS ve tabakalı SGD algoritmalarını ölçeklendirmede önerilen yöntem ve modellerin önemini göstermektedir.

Anahtar sözcükler: Paralel Algoritmalar, Kombinatoryal Algoritmalar, Yüksek Başarımlı Hesaplama, Tensör Ayrışımı, Matris Tamamlama, Hiperçizge Bölümleme, İletişim Maliyeti Minimizasyonu.

Acknowledgement

My deepest gratitude goes to my supervisor, Prof. Cevdet Aykanat, for his guidance, patience, and support during my graduate studies at Bilkent. Supervisors of his kind are very rare nowadays and I feed privileged to work with him and learn his way of doing scientific research.

I would like to thank the committee members of this thesis: Prof. Özcan Özturk, Prof. Murat Manguoğlu, Dr. Bora Uçar, and Prof. F. Şukru Torun for their valuable discussions and feedback.

I would like to thank Bilkent University for supporting my graduate studies and for their generous scholarships. They are devoted to maintaining excellent academic environment for researchers in every way possible.

I would like to thank my colleagues Dr. Ozan Karsavuran, Dr. Seher Acer for their valuable contributions to this thesis. I'm thankful to Dr. Kadir Akbudak who was my mentor during my M.Sc. studies. Thanks to Dr. R. Oğuz Selvitopi, I learned a lot from going through his codes. Thanks to Mustafa Duymuş for all the lovely and joyful chats we had.

I'm grateful to my friends and family in Ankara for all the good times and the social environment that made my stay tolerable: Tariq (homie), Salman and Elif (and little Vera Melis), Mohammad and Nour (and their sons Hasan and Adam), Mohammed and Fatima (and little Yusuf), Omair and Nada (and little Salwa), Ahmed and Buşra, M. Tarik and his mother, Maen, Ghassan, Fuad, Abulrazzak, Hana, Hammam and others.

I'm forever grateful to my family: my parents Fuad Abubaker and Haijar Mahameed, my sisters Mays and Dima, my brother-in-law Ahmad Nazzal , my nephew Jad, my lovely wife Maha and my daughter Rafif for their continuous love and support.

Finally, I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting me during my PhD under grants EEEAG-116E043, EEEAG-119E035 and EEEAG-121E391.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Scope

The vast majority of the data generated to be analyzed nowadays, whether by scientific or industrial applications, is sparse. For instance, graphs that model various types of networks (e.g., social, biological, industrial) are represented by sparse adjacency matrices. Domains like e-commerce, social networks, tracking applications, generate data in the form of multi-way observations (e.g., {*user, item, timestamp*} triplets for online purchasing) that are naturally represented by sparse tensors.

Factorization of sparse tensors and matrices is essential in many fields, including scientific computing, machine learning, data analysis, medical imaging, and more [1, 2]. The need for efficiently parallel algorithms to compute tensor and matrix factorization is growing by the day. This need stems from two facts: (i) the sparse data being generated is exponentially growing beyond the capacity of a single machine's memory, and (ii) the factorization is becoming a kernel operation within large frameworks/applications that operate on large-scale systems; and the scalability of the kernel factorization operation is crucial to the scalability of the parent framework/application. A milestone in the world of computing has been recently announced as the Frontier supercomputer at the Oak Ridge National Labs, USA, has achieved 1.102 Exaflop/s marking the beginning of the exascale era. Utilizing such valuable resources requires careful design of parallel algorithms such that high computational balance is achieved while minimizing the communication overhead.

In this dissertation, we focus on two factorization algorithms: (i) The CPD-ALS algorithm which computes the Canonical Polyadic Decomposition of a sparse tensor utilizing the alternating least squares method, and (ii) the distributed stratified stochastic gradient descent algorithm which computes a low-rank factorization of a sparse matrix.

Canonical Polyadic (or CANDECOMP/PARAFAC) decomposition (CPD) is an extension of singular value decomposition of matrices to tensors and a fundamental tool for the analysis of multiway data. It approximates a given tensor by the sum of multiple rank-one tensors so that each rank-one tensor corresponds to a structural feature in the data set. CPD is a fundamental tool in unsupervised learning setting [3–6], used for dimensionality reduction, data completion and compression, and finds application in various domains such as neuroscience [7, 8], machine learning [9, 10], chemistry [11], cybersecurity [12], signal processing [13], and network analysis [14]. It has also become an integral part of different machine learning fields either as a method (e.g., regression [15], supervised classification [16]), or as a support tool (e.g., compression for Deep Learning [17–19]) and more [20]. Each iteration of the CPD-ALS algorithm computes a new factor matrix for each mode by performing several computational steps. Among those steps, matricized tensor times Khatri-Rao product (MTTKRP) constitutes the biggest bottleneck because of its high computational cost.

The low-rank matrix factorization utilizing stochastic gradient descent can be utilized in many fields [21–24]. We focus on the most famous one, collaborative filtering for recommender systems, and present our findings in its context. Recommender systems are omnipresent in e-commerce as well as social, professional and academic networks. These systems help businesses improve profit by targeted advertisements to interested parties, facilitate the recruitment process by matching more relevant candidates to jobs, and help academics explore cross-disciplinary research works as well as expand their collaboration networks. Recommender systems can involve one or more techniques, among which Collaborative Filtering (CF) is the most widely used. Collaborative filtering approaches recommend an *item* to a target *user* by using other users' ratings given that those other users and the target user have rated some other items similarly.

Matrix factorization have been successfully used in collaborative filtering via revealing feature vectors that represent the users and the items (latent factors). A sparse rating matrix is factorized into two dense matrices representing the feature vectors of items and users, and these dense matrices are then used to predict missing entries in the original rating matrix. This use of matrix factorization is commonly referred to as *matrix completion*. The matrix factorization can be computed with different methods, including stochastic gradient descent (SGD), alternating least squares (ALS), cyclic coordinate descent (CCD) and more [25].

SGD is very efficient and usually achieves high completion accuracy compared to other methods. However, given its sequential nature it has been a challenge to efficiently parallelize while maintaining accuracy and convergence guarantee. For this reason, serializable parallel SGD algorithms are most desired. Serializability of parallel SGD refers to the existence of an equivalent serially-executed SGD algorithm with the same update order. Serializability guarantees the convergence and assures that no two processors update the same feature vector at the same time (race condition) thus leading to faster convergence [26]. Stratified SGD (SSGD) [27] is the de-facto algorithm for achieving a serializable parallel SGD.

Both CPD-ALS and stratified SGD are usually executed multiple times by practitioners with different factorization ranks. For this reason, it is important that each execution runs in a fast and efficient manner to reduce the turnaround time, which in turn speeds up the process of extracting information, and to reduce energy consumption. This dissertation focuses on addressing three major issues in the two algorithms that degrade their performance and limit their scalability.

1.1.1 Balancing Computational Loads of MTTKRP

When CPD-ALS is performed on a sparse tensor and on a distributed-memory setting, the optimization of the MTTKRP operation becomes more tedious due to the irregular sparsity pattern of the tensor nonzeros. Practitioners usually perform tensor decomposition many times with different ranks, which makes the optimization of MTTKRP even more crucial for reducing the turnaround time of their analysis. For achieving a performant and scalable parallel decomposition, one should take the sparsity information into account in crucial design decisions associated with high communication and computational costs. These decisions involve

(i) how the input tensor is distributed among processors,

(ii) how the tensor nonzeros are stored in each processor,

(iii) and how MTTKRP is realized on the given storage.

To address (i), several successful partitioning models [28–31] have been proposed with the goal of reducing the communication cost of MTTKRP while maintaining a balance on its computational costs on all processors. To address (ii) and (iii), several storage formats [32–34] have been proposed, usually together with a new method to realize MTTKRP on the proposed format. Among those, compressed sparse fiber (CSF) proves to be the most commonly-used storage format due to its efficiency in terms of both memory and computation [30, 32, 33]. CSF is an extension of the compressed sparse row format to tensors and the total flop count in the CSF-oriented MTTKRP is proportional to the total number of nonzeros and fibers (along a specified mode) in the given tensor. The flop count of the CSF-oriented MTTKRP is significantly smaller than the flop count of the MTTKRP based on the coordinate-format [32, 33].

Besides the popularity of the CSF format, tensor partitioning models still assume the computational cost of MTTKRP to be proportional to the total number of nonzeros in the input sparse tensor. This creates a discrepancy when the tensor is stored in a CSF format and hence a CSF-oriented MTTKRP operations are performed in a processor. This is because when CSF is used, the computational cost of a local MTTKRP is a function of the nonzero count as well as the fiber count of the sparse subtensor assigned to that processor. This discrepancy leads to a failure in balancing the computational loads of processors in the distributed-memory parallelization. This failure becomes more prominent as the variance on the nonzero counts of fibers becomes larger, that is, as the tensor becomes more irregular.

1.1.2 High Latency Overhead in CPD-ALS

In distributed-memory parallel CPD-ALS, each MTTKRP operation needs sparse reduce and expand communications as well as two dense reduce communications. The sparse reduce/expand are irregular due to the sparsity pattern of the tensor and they are performed with point-to-point (P2P) messages. On the other hand, the dense reduce communications involve data of sizes proportional to the decomposition rank, which are required by all processors and thus are performed using the collective `ALL-REDUCE` operation from the MPI primitives.

Inter-processor communication cost ideally consists of latency term and bandwidth term. The latency term is proportional to the number of messages sent, whereas the bandwidth term is proportional to the volume of data transferred. If the number of messages is high, the latency cost might dominate the overall communication component since each message's startup time might be higher than that of sending a few kilobytes of data [35].

The bandwidth overhead of MTTKRP scales with both tensor size and decomposition rank, whereas latency overhead increases with increasing number of processors as well as with increasing irregularity in the sparsity pattern of the tensor. That is, CPD-ALS becomes latency bound for small decomposition rank values. Although current distributed-memory parallel CPD-ALS algorithms, which utilize P2P communication scheme [36–45], scale well up to a certain number of processors, these algorithms fail to scale after some number of processors. We empirically find this number to be around 512–1024 processors as also reported in [37, 44]. Thus, optimizing the latency overhead is a key point for scaling CPD-ALS on large number of processors.

1.1.3 High Bandwidth Overhead in Distributed Stratified SGD

The state-of-the-art methods implementing SSGD [27, 46, 47] achieve the interprocessor communication necessary for the correctness of the SSGD through sending/receiving feature vectors with counts proportional to one of the dimensions of the input rating matrix. These methods do not utilize the sparsity of the rating matrix thus producing a huge amount of extra unnecessary communication especially when the nonzero density of the rating matrix is low. The extra communication did not pose a concern because these methods are tested on a relatively small number of processors (up to 64) in distributed setting. At such scale, the SGD runtime is expected to be dominated by computation and investing in improving the communication component does not significantly affect the overall running time. On larger scale, however, the communication component becomes dominant, and therefore reducing the communication overhead becomes essential. Therefore, eliminating the huge extra communication volume, manifested as bandwidth overhead, in these algorithms is a primary parallelization consideration on large-scale systems.

1.2 Summary of Contributions

This dissertation presents the following contributions to address the three problems mentioned in the previous section:

- To address load balancing of MTTKRP:

- We identify a load-balancing problem of the nonzero-based parallel MTTKRP that utilizes the CSF storage format: (i) balancing the computation loads of processors becomes harder as the fiber counts depend on the nonzero distribution (which causes fiber fragmentation), and (ii) the total fiber count is not constant and might increase depending on the nonzero distribution and the number of processors used.
- We propose a framework to capture true computational loads in a fine-grain (FG) hypergraph model of the MTTKRP. The framework enables the FG model to encapsulate reducing the increase in computational loads due to fiber fragmentation through augmenting the model with nets that represent these fibers. Furthermore, the framework includes a novel weighting scheme, realized with recursive biparitioning, to include the costs of fibers in the computational loads of processors.

- To address the high latency overhead in distributed CPD-ALS:
    - We propose a framework for embedding latency-heavy point-to-point messages into a global collective (an All-reduce operation). The framework begins with a computation/communication reordering scheme for the CPD-ALS algorithm to enable the embedding. The scheme does not affect the mathematical results of reordered operations or the flow of the algorithm. Then, since these point-to-point messages constitute sparse expand and reduce operations, the framework presents an expand-and-reduce-aware embedding algorithm to avoid communicating any duplicate entities during the embedded communication. The algorithm is inspired by the hypecube-based all-reduce and accomplishes both an all-reduce operation and a sparse expand (or reduce) by the end of its execution.
    - We propose a partitioning method based on a hypergraph model for reducing the communication overhead during the embedding algorithm. The HP model encapsulates reducing a concurrent communication metric that mimics the communication behavior of the hypercube-based algorithm. The partitioning method utilizes the RB scheme

    to correctly allow encapsulating the concurrent communication metric through two novel strategies: net anchoring and sibling subnet removal.

- ❖ To address the high bandwidth overhead in distributed SSGD:
    - ✧ We propose a communication-efficient framework for the SSGD algorithm. Our framework consists of efficiently finding the essential feature vectors to be communicated between processors and communicating them through point-to-point (P2P) messages. This contributes to reducing the bandwidth overhead through avoiding the extra unnecessary communication. This approach has the down side of increasing the number of messages sent per processor, thus increasing the latency overhead and possibly affecting the scalability.
    - ✧ To reduce the increase in the number messages, we propose a novel approach called hold and combine that reduces the upper bound on the number of messages from $\mathcal{O}(K^2)$ to $\mathcal{O}(K \lg K)$.
    - ✧ We also propose to further reduce the bandwidth overhead of the P2P messages by using an intelligent partitioning method. This method utilizes a hypergraph model that correctly encapsulates the total volume of communication between processors.

1.3 Organization

The rest of this dissertation is organized as follows: Chapter 2 provides all relevant backgrounds on Tensors, CPD, matrix completion through SGD, the HP problem and the recursive biparitioning paradigm. Chapter 3, 4 and 5 discuss the proposed algorithms and models addressing the problems receptively defined in 1.1.1, 1.1.2 and 1.1.3. Each chapter contains specific preliminaries, detailed descriptions of the proposed methods and algorithms, experimental evaluations, related works and a conclusion. Finally, Chapter 6 concludes the dissertation.

Chapter 2

Background

2.1 Tensors and Notations

We denote tensors by calligraphic letters ($\mathcal{X}$) and matrices by bold capital letters ($\mathbf{A}$). The number of dimensions of a tensor, denoted by N, is called the *mode* of the tensor. Note that matrices and vectors are 2-mode and 1-mode tensors, respectively. For the sake of simplicity, we assume 3-mode tensors of size $I{\times}J{\times}K$.

Fibers are analogous to matrix rows or columns, which can be obtained by fixing all but one indices of the tensor. In 3-mode tensors there are row, column and tube fibers which are respectively denoted by $\mathcal{X}(i,:,k)$, $\mathcal{X}(:,j,k)$ and $\mathcal{X}(i,j,:)$. *Slices* are analogous to matrices and can be obtained by fixing all but two indices. In 3-mode tensors, there are horizontal (e.g., $\mathcal{X}(i,:,:)$), lateral (e.g., $\mathcal{X}(:,j,:)$) and frontal (e.g., $\mathcal{X}(:,:,k)$) slices. *Matricization* of a tensor means unfolding it into a matrix shape along one of its modes. For instance, the matricization of $\mathcal{X}$ along the first mode, denoted by $\mathbf{X}_{(1)}$, is a matrix of size $I \times JK$. We refer the reader to the survey by Kolda and Bader [1] for more details on tensor decompositions.

2.2 The Canonical Polyadic Decomposition

The CPD decouples a tensor into R rank-1 components as $\mathcal{X} \approx \sum_{i=1}^{R} \mathcal{X}_i$, where rank-1 component $\mathcal{X}_i$ is the outer product of 3 vectors (or N vectors in the general case) $\mathbf{a}_i \circ \mathbf{b}_i \circ \mathbf{c}_i$. The $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ components in each of the R rank-one tensors are assembled to respectively form factor matrices $\mathbf{A} \in \mathbb{R}^{I\times R}$, $\mathbf{B} \in \mathbb{R}^{J\times R}$ and $\mathbf{C} \in \mathbb{R}^{K\times R}$. Here, R is called the decomposition rank. We use $\mathbf{A}(i,:)$ to denote the ith row of $\mathbf{A}$. When the mode of the tensor is irrelevant to the discussion, we use r_i to refer to a row in a factor matrix along any mode.

Algorithm 1 CPD-ALS for 3-mode Tensors

```
 1: procedure CPD-ALS($\mathcal{X}$)
 2:     Initialize matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ randomly
 3:     while not converged do
 4:         $\mathbf{A}' \leftarrow \mathbf{X}_{(1)}(\mathbf{B} \odot \mathbf{C})$                    ▷ MTTKRP
 5:         $\mathbf{A} \leftarrow \mathbf{A}'(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{-1}$
 6:         Normalize columns of $\mathbf{A}$ into $\lambda$
 7:         $\mathbf{B}' \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})$                    ▷ MTTKRP
 8:         $\mathbf{B} \leftarrow \mathbf{B}'(\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^{-1}$
 9:         Normalize columns of $\mathbf{B}$ into $\lambda$
10:         $\mathbf{C}' \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})$                    ▷ MTTKRP
11:         $\mathbf{C} \leftarrow \mathbf{C}'(\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^{-1}$
12:         Normalize columns of $\mathbf{C}$ into $\lambda$
return $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$
```

The goal of an algorithm computing the CPD is to find the best approximation of a tensor $\mathcal{X}$ using R components that minimizes a norm of $\mathcal{X} - \sum_{i=1}^{R} \lambda_i \mathcal{X}_i$. Here, the vectors used to construct $\mathcal{X}_i$ are normalized to length 1, and the value λ_i is used as a scaling factor to the normalized rank-1 component tensor $\mathcal{X}_i$. The most commonly used algorithm to compute the CPD is CPD-ALS, which uses the *Alternating Least Squares* approach. Algorithm 1 shows the steps of the CPD-ALS algorithm. During each iteration, two factor matrices are fixed to find the remaining one by solving a linear alternating least squares problem. For instance, $min_{\mathbf{A}}||\mathbf{X}_{(1)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T||_R^2$ is solved to find $\mathbf{A}$ by computing $\mathbf{X}_{(1)}(\mathbf{C}\odot\mathbf{B})(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})$. The columns of factor matrices are then normalized to length one, and the actual lengths are stored in λ. The operations $\odot$ and $*$ denote the Khatri-Rao and the Hadamard products, respectively. In the algorithm, the result of the MTTKRP operation along the first, the second and the third mode is respectively stored in $\mathbf{A}'$, $\mathbf{B}'$ and $\mathbf{C}'$.

2.3 Matrix Completion with SGD

We define the matrix completion problem in the context of collaborative filtering as follows: Given a set $\mathcal{U}$ of N users, a set $\mathcal{I}$ of M items, and a set of ratings Ω as the known entries of a sparse rating matrix $\mathbf{R} \in \mathbb{R}^{N\times M}$. The problem is to find two dense factor matrices $\mathbf{W} \in \mathbb{R}^{N\times F}$ and $\mathbf{H} \in \mathbb{R}^{M\times F}$ such that a low-rank approximation $\mathbf{R} \approx \mathbf{W}\mathbf{H}^\top$ is achieved. Here, $F \ll M, N$ is called the dimension or the rank of the factorization. Then, a missing rating $\hat{r}_{ij} \notin \Omega$ can be approximated as

$$\hat{r}_{ij} = \mathbf{w}_i \mathbf{h}_j^\top, \tag{2.1}$$

where $\mathbf{w}_i$ and $\mathbf{h}_j$ respectively denote the ith row of $\mathbf{W}$ and the jth row of $\mathbf{H}$. The quality of the approximation is usually measured by an application-dependent loss function $\mathcal{L}$, thus the problem becomes $\arg\min_{\mathbf{W},\mathbf{H}} \mathcal{L}(\mathbf{R}, \mathbf{W}, \mathbf{H})$. For collaborative filtering, $\mathcal{L}$ is usually the Euclidean distance and thus the problem becomes

$$\operatorname*{arg\,min}_{\mathbf{W},\mathbf{H}} \sum_{(i,j)\, s.t.\, r_{ij}\in\Omega} \left((r_{ij} - \hat{r}_{ij})^2 + \gamma(\|\mathbf{w}_i\|^2 + \|\mathbf{h}_j\|^2\right), \tag{2.2}$$

where γ is a regularization parameter to avoid over-fitting, and $\hat{r}_{ij}$ is an approximation of an existing r_{ij} and is computed with (2.1).

Since the minimization problem in (2.2) has two unknowns $\mathbf{W}$ and $\mathbf{H}$, $\mathcal{L}$ is a non-convex function [21]. SGD has been widely used to optimize (minimize) such functions due to its ability to escape local minimas. At an SGD epoch, each rating $r_{ij} \in \Omega$ is used to update the objective function's parameters. The gradient of the objective function at point r_{ij} is calculated ($\nabla_{r_{ij}} \mathcal{L}_{r_{ij}}(\mathbf{R}, \mathbf{W}, \mathbf{H})$) and the corresponding $\mathbf{w}_i$ and $\mathbf{h}_j$ rows are updated as

$$\mathbf{w}_i = \mathbf{w}_i - \epsilon[(r_{ij} - \hat{r}_{ij})\mathbf{h}_j + \gamma \mathbf{w}_i], \tag{2.3}$$

$$\mathbf{h}_j = \mathbf{h}_j - \epsilon[(r_{ij} - \hat{r}_{ij})\mathbf{w}_i + \gamma \mathbf{h}_j] \tag{2.4}$$

where ϵ is the step size.

2.4 Hypergraph Partitioning (HP)

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. Each net $n\in\mathcal{N}$ connects a subset of vertices, which is denoted by $Pins(n)$. Each net n is assigned a cost $c(n)$, whereas each vertex v maybe assigned C weights denoted by $w^c(v)$ where $c \in \{1, 2, .., C\}$. For $C > 1$, the HP problem is commonly known as multi-constraint HP.

$\Pi^K(\mathcal{H}) = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_K\}$ denotes a K-way partition of $\mathcal{H}$ if the vertex parts are mutually exclusive and exhaustive. For a given partition Π,

$$W^c(\mathcal{V}_k) = \sum_{v_i\in\mathcal{V}_k} w^c(v_i), \forall c \in \{1, 2, .., C\}. \tag{2.5}$$

denotes the c^{th} weight of part $\mathcal{V}_k$. In Π, a net is said to connect a part if it connects at least one vertex in that part. We define $\Lambda(n)$ as the set of parts that n connects. That is,

$$\Lambda(n) = \{\mathcal{V}_k \in \Pi^K(\mathcal{H}) \mid \mathcal{V}_k \cap Pins(n) \neq \emptyset\}.$$

Furthermore, $\lambda(n) = |\Lambda(n)|$ is called the connectivity of n. Net n is called *cut* if it connects at least two parts, i.e., $\lambda(n) > 1$, and called *internal* otherwise.

The HP problem refers to obtaining $\Pi^K(\mathcal{H})$ while optimizing an objective function defined over $\mathcal{N}$ and maintaining a balance constraint between the parts defined over $\mathcal{V}$. The two mostly common objective functions are minimizing the cut-net metric

$$\mathit{cutnet}(\Pi) = \sum_{n \ni \lambda(n)>1} c(n) \tag{2.6}$$

and the connectivity-1 metric

$$\mathit{conn}{-}1(\Pi) = \sum_{n \ni \lambda(n)>1} c(n)(\lambda(n) - 1). \tag{2.7}$$

while maintaining balance on the part weights as

$$W^c(\mathcal{V}_k) \leq W^c_{avg}(1 + \epsilon), \forall\mathcal{V}_k \in \Pi, \forall c \in \{1, 2, \ldots, C\}, \tag{2.8}$$

where $W^c_{avg}{=}\sum_{k=1}^{K} W^c(\mathcal{V}_k)/K$ denotes the average part weight for the c^{th} constraint and ϵ denotes the maximum allowed imbalance ratio. The keyword *cutsize* is used to refer to the total cutsize of $\Pi^K(\mathcal{H})$ with the used metric. In HP with fixed vertices, part assignment of some vertices are given a priori to partitioning.

2.5 Recursive Bipartitioning (RB)

Recursive bipartitioning (RB) is a common scheme to obtain K-way partitions by recursively bipartitioning an input hypergraph. The RB scheme generates a tree of $\lg K$ levels. Given an initial hypergraph $\mathcal{H}^0_0$, at each level $\ell \in \{0, 1, \ldots, \lg K{-}1\}$ there are 2^ℓ subhypergraphs $\mathcal{H}^\ell_0, \mathcal{H}^\ell_1, \ldots, \mathcal{H}^\ell_{2^\ell-1}$; each two of which are constructed from two-way partitioning (Π^2) of a parent hypergraph in level $\ell-1$ and will be bipartitioned to construct two subhypergraphs in level $\ell+1$. RB has been successfully used to allow HP methods encode complex partitioning objectives for scientific computing and machine learning applications [44, 48–51].

Given $\mathcal{H}^\ell_k$, $0 \leq k \leq 2^\ell-1$, constructing subhypergraphs $\mathcal{H}^{\ell+1}_{2k}$ and $\mathcal{H}^{\ell+1}_{2k+1}$ using $\Pi^2(\mathcal{H}^\ell_k) = \{\mathcal{V}_L, \mathcal{V}_R\}$ is achieved as follows: The vertex sets of $\mathcal{H}^{\ell+1}_{2k}$ and $\mathcal{H}^{\ell+1}_{2k+1}$ are respectively $\mathcal{V}_L$ and $\mathcal{V}_R$. The net sets of $\mathcal{H}^{\ell+1}_{2k}$ and $\mathcal{H}^{\ell+1}_{2k+1}$ are constructed according to the net categorization (cut or internal) by $\Pi^2(\mathcal{H}^\ell_k)$ while following a strategy to maintain a correct cutsize metric. For the sake of simplicity, whenever the discussion is about $\mathcal{H}^\ell_k$ at an arbitrary ℓ and arbitrary k, we use $\mathcal{H}$ to represent $\mathcal{H}^\ell_k$ and $\mathcal{H}_L = (\mathcal{V}_L, \mathcal{N}_L)$ and $\mathcal{H}_R = (\mathcal{V}_R, \mathcal{N}_R)$ to respectively represent $\mathcal{H}^{\ell+1}_{2k}$ and $\mathcal{H}^{\ell+1}_{2k+1}$. Here, L and R are used to refer to the two parts as *L*eft and *R*ight, respectively.

In order to maintain the cut-net metric given in (2.6), cut nets are removed and internal nets are inherited to their respective subhypergraphs following $\mathcal{V}_L$ and $\mathcal{V}_R$. This way, each cut net contributes its cost once to the cutsize. If each cut net is split into two sub-nets assigned to the left and right sub-hypergaphs, that is, a cut net n in Π^2 is split into two subnets n' and $n\prime\prime$, where $Pins(n') = Pins(n) \cap \mathcal{V}_L$ and $Pins(n'') = Pins(n) \cap \mathcal{V}_R$, then each net can be split at most $K-1$ times during RB, where K is the desired number of partitions. This directly allows encoding the "connectivity-1" metric given in (2.7) as proposed in [52] since net n contributes its cost to the cutsize $\lambda(n)-1$ times during RB.

Chapter 3

True Load Balancing for Matricized Tensor Times Khatri-Rao Product

3.1 Overview

In this chapter, we propose a tensor partitioning model with true load balancing for the MTTKRP operation when the CSF format is used. Our model is based on the fine-grain model [28], which is (theoretically and practically) the most successful model in reducing the total communication volume and balancing the number of tensor nonzeros in processors.

The rest of the chapter is organized as follows. In Section 3.2 preliminaries on efficient MTTKRP and the HP-based fine-grain method for CPD-ALS are given. The deficiencies of the HP-based fine-grain method are discussed in Section 3.3. In Section 3.4, our proposed framework is presented and discussed in details. Experimental results are given and discussed in Section 4.6. Related work is given in Section 3.6 and the chapter is concluded in Section 3.7.

3.2 Preliminaries

3.2.1 Efficient computation of MTTKRP

The MTTKRP operation, which is the target operation in this chapter, takes place in lines 4, 7, and 10 of Algorithm 1, each of which is for computing a factor matrix along a different mode. Although it is shown as a multiply of an unfolded (matricized) tensor (e.g., $\mathbf{X}_{(1)}$) with a large matrix (e.g., $(\mathbf{B} \odot \mathbf{C})$), this is basically for simplicity and the corresponding multiply is impractical for sparse tensors. Many implementations prefer to realize the MTTKRP operation $\mathbf{A}' \leftarrow \mathbf{X}_{(1)}(\mathbf{B} \odot \mathbf{C})$ in a rowwise way for $\mathbf{A}'$, such as

$$\mathbf{A}'(i,:) = \sum_{\mathcal{X}(i,j,k) \neq 0} \mathcal{X}(i,j,k)[\mathbf{B}(j,:) * \mathbf{C}(k:)]. \tag{3.1}$$

This computation style is preferred when the tensor is stored as a list of (i, j, k, val) coordinates, called the COO format. Note that in this formulation, the corresponding rows of $\mathbf{B}$ and $\mathbf{C}$ are retrieved and multiplied for each nonzero.

As a better alternative, the software toolkit SPLATT [32] uses the flops-reducing formulation

$$\mathbf{A}'(i,:) = \sum_{j \ni nnz(\mathcal{X}(i,j,:)) \neq 0} \mathbf{B}(j,:) * \sum_{k \ni \mathcal{X}(i,j,k) \neq 0} \mathcal{X}(i,j,k)\mathbf{C}(k:) \tag{3.2}$$

which uses a fiber-centric data structure (to be discussed in the next subsection). Hereafter, $nnz(.)$ refers to the number of nonzeros in a (sub)tensor. In (3.2), the outer and inner summations respectively run over all nonzero fibers of slice $\mathcal{X}(i,:,:)$, and all nonzero entries of fiber $\mathcal{X}(i,j,:)$.

The efficient formulation in (3.2) can be realized using the Compressed Sparse Fibers (CSF) scheme, which was first introduced by Smith and Karypis [33]. The CSF storage scheme can be considered as a natural extension of the Compressed Sparse Rows/Columns schemes widely used for sparse matrices. Fig. 3.1 shows an illustration of the CSF storage format for a 3-mode sparse tensor. In the

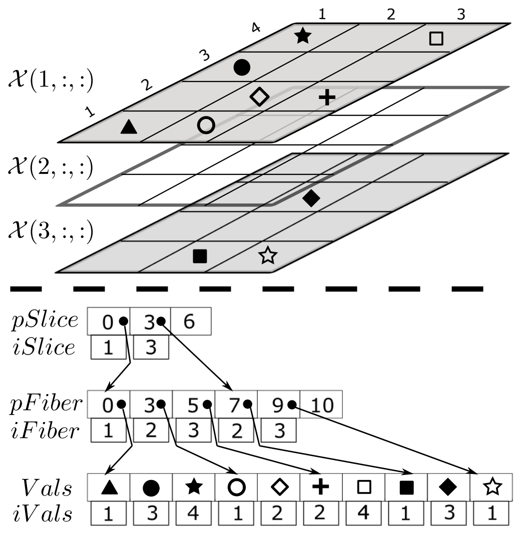


Figure 3.1: A 3-mode tensor (top) and the corresponding CSF storage (bottom).

figure, the *pSlice* and *pFiber* arrays respectively represent the compressed slices and fibers. The *pSlice* array consists of pointers to the starting indices of the compressed fibers of the respective slices in the *pFiber* array. Similarly, the *pFiber* array consists of pointers to the starting indices of the nonzeros of the respective fibers in *Vals*. The *iSlice*, *iFiber* and *iVals* arrays respectively store the i, j and k indices of the respective nonzero slices, fibers and entries.

Two CSF-based computational schemes are used for computing the MTTKRP operations in Algorithm 1 using the formulation in (3.2). These two schemes will be referred to as CSF-S and CSF-D, where "-S" and "-D" refer to the use of Single and Double storage, as will be explained shortly.

The CSF-S scheme operates on a single CSF storage of the tensor, where the compressed fibers are the tensor's fibers along the longest mode. This scheme is proposed by Smith and Karypis [33] and currently used in SPLATT. CSF-S utilizes Algorithm 2 for computing the MTTKRP operations along all modes but the longest mode, while it utilizes Algorithm 3 to compute the MTTKRP operation along the longest mode.

The CSF-D scheme uses two different CSF storages of the tensor. The first storage s_1 is the same as of CSF-S, while the second storage s_2 utilizes the fibers along the second longest mode as the compressed *pFiber* array. This scheme was used in several works that target computing the MTTKRP in distributed settings [30, 32, 53]. CSF-D utilizes Algorithm 2 for computing the MTTKRP operations along all modes but the longest mode by feeding s_1, and it utilizes the same algorithm to compute the MTTKRP operation along the longest mode by feeding s_2. Although CSF-D has a larger memory footprint compared to CSF-S, it has the advantage of avoiding the use of costly mutexes when used in hybrid (distributed + shared) settings [33].

Algorithm 2 MTTKRP used by both CSF-D and CSF-S

```
Require: Tensor $\mathcal{X}$ stored in CSF, $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$
 1: for $i \leftarrow 1$ to $size(pSlice)$ do
 2:     $is \leftarrow iSlice[i]$
 3:     for $j \leftarrow pSlice[i]$ to $pSlice[i+1]-1$ do
 4:         $jf \leftarrow iFiber[j]$
 5:         if $pFiber[j+1] - pFiber[j] = 1$ then
 6:             $k \leftarrow pFiber[j]$
 7:             $\hat{\mathbf{A}}(is,:) += Vals[k] * \mathbf{C}(iVals[k]) * \mathbf{B}(if,:)$
 8:         else
 9:             $acc(:) \leftarrow 0$
10:             for $k \leftarrow pFiber[j]$ to $pFiber[j+1]-1$ do
11:                 $acc(:) += acc(:) + Vals[k] * \mathbf{C}(iVals[k],:)$
12:             $\hat{\mathbf{A}}(is,:) += acc(:) * \mathbf{B}(jf,:)$
```

Algorithm 3 MTTKRP used by CSF-S

```
Require: Tensor $\mathcal{X}$ stored in CSF, $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$
 1: for $i \leftarrow 1$ to $size(pSlice)$ do
 2:     $is \leftarrow iSlice[i]$
 3:     for $j \leftarrow pSlice[i]$ to $pSlice[i+1]-1$ do
 4:         $jf \leftarrow iFiber[j]$
 5:         if $pFiber[j+1] - pFiber[j] = 1$ then
 6:             $k \leftarrow pFiber[j]$
 7:             $\hat{\mathbf{C}}(iVals[k],:) += Vals[k] * \mathbf{A}(is,:) * \mathbf{B}(jf,:)$
 8:         else
 9:             $acc(:) \leftarrow \mathbf{A}(is,:) * \mathbf{B}(jf,:)$
10:             for $k \leftarrow pFiber[j]$ to $pFiber[j+1]-1$ do
11:                 $\hat{\mathbf{C}}(iVals[k],:) += acc(:) * Vals[k]$
```

3.2.2 Fine-Grain (FG) Partitioning for MTTKRP

In the work by Kaya and Uçar [28], a fine-grain task is defined as the multiplication of a tensor nonzero by the Hadamard product of the corresponding rows of the factor matrices along all but the mode of the factor matrix being computed (according to (3.1)).

A fine-grain hypergraph model $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is proposed [28] for fine-grain task partitioning. $\mathcal{H}$ contains a vertex v_{ijk} for each tensor nonzero $\mathcal{X}(i,j,k)$, and nets n_i^H, n_j^L and n_k^F for respectively each nonempty horizontal, lateral and frontal slice of the tensor. Each vertex v_{ijk} is connected by three nets n_i^H, n_j^L and n_k^F.

All vertices of $\mathcal{H}$ are assigned a unit weight under the assumption that every nonzero of $\mathcal{X}$ incurs the same amount of computation during the MTTKRP operations. All nets of $\mathcal{H}$ are assigned a cost of R since factor-matrix rows of size R words are communicated between processors. Then, the partitioning objective of minimizing the cutsize encodes minimizing the total volume of communication due to expand-type communications on input matrix rows as well as reduce-type communications on output matrix rows.

3.3 Deficiencies of the Fine-Grain Model

3.3.1 Failure to encode processors' computational loads

The COO-based implementation of the MTTKRP operation according to (3.1) incurs $3Rm$ flops for tensor $\mathcal{X}$ with m nonzero elements. Here, $2Rm$ flops are performed for the initial products and Rm flops are performed for the summation operations. In other words, each tensor nonzero incurs $3R$ flops. So although vertices are assigned unit weight in the conventional FG model, we assume the vertices are assigned a weight of

$$w(v_{ijk}) = 3R. \tag{3.3}$$

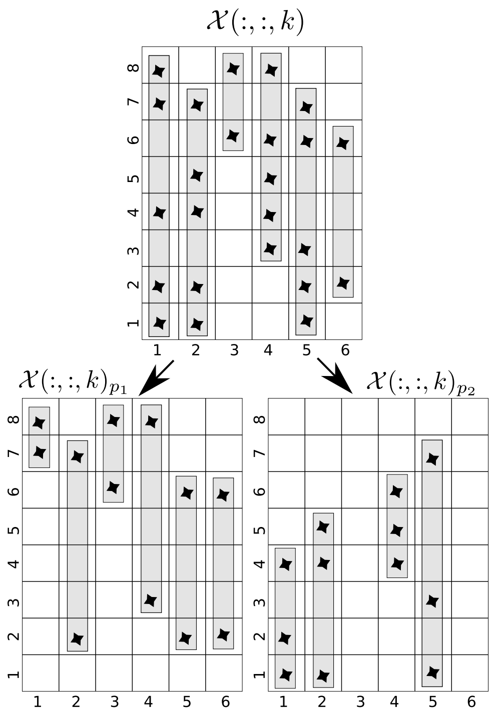


Figure 3.2: A bipartition of slice $\mathcal{X}(:,:,k)$ to processors p_1 and p_2, having the same nonzero count but different flop counts.

In this way, the part weights computed by using (3.3) in (2.5) will correctly encapsulate processor's computational loads .

On the other hand, the CSF format enables reducing the amount of total computation from $3Rm$ flops to $2R(m+F)$ flops, where F denotes the number of fibers, using the formulation in (3.2). This is because, in (3.2), each nonzero $\mathcal{X}(i,j,k)$ incurs $2R$ flops due to $\mathbf{C}(k,:)$, whereas each fiber $\mathcal{X}(i,j,:)$ incurs $2R$ flops due to $\mathbf{B}(j,:)$. That is, the amount of computation associated with each nonzero may differ depending on the fiber fragmentation introduced by the partitioning algorithm. So, the part weights computed according to (3.3) fail to correctly encapsulate the computational loads of processors.

The top part of Fig. 3.2 shows a sample 8×6 frontal slice $\mathcal{X}(:,:,k)$ with 24 nonzeros. In the figure, stars represent nonzeros while shaded rectangles represent fibers along the longest mode which is the second mode. The bottom part shows a bipartition of the nonzeros of the slice between two processors p_1 and

p_2. The subslices assigned to p_1 and p_2 are respectively denoted by $\mathcal{X}(:,:,k)_{p_1}$ and $\mathcal{X}(:,:,k)_{p_2}$. This bipartition shows an even nonzero partition since each subslice has 12 nonzeros. However, the nonzeros of the subslices $\mathcal{X}(:,:,k)_{p_1}$ and $\mathcal{X}(:,:,k)_{p_2}$ respectively belong to 6 and 4 subfibers. Thus, $\mathcal{X}(:,:,k)_{p_1}$ will incur $2R(12+6) = 36R$ flops associated with the MTTKRP operation on p_1, whereas $\mathcal{X}(:,:,k)_{p_2}$ will incur $32R$ flops on p_2. So, despite the even nonzero distribution, the partition incurs a significant amount of computational load imbalance.

3.3.2 Increase in total computation

As mentioned in Section 3.3.1, the number of flops performed using the fiber-centric MTTKRP formulation in (3.2) is equal to $2R(m+F)$ in serial and shared-memory settings. However, in distributed-memory settings, since the fibers are local to the processors, the number of flops increases as a result of fragmenting fibers among processors.

Assuming single precision floating-point values, the COO-based MTTKRP incurs $16m + 12Rm$ memory byte accesses [54], whereas the CSF-based MTTKRP with S slices incurs $8(S + F + m) + 12R(m + F)$ accesses. Note that fiber fragmentation increases the number of flops as well as the number of memory accesses at the same rate, thus it does not affect the arithmetic intensity (flops per byte) of the CSF-based MTTKRP. Therefore, any further discussion on increasing/decreasing flop counts also applies to the associated number of memory accesses.

In an ideal situation, each fiber is assigned to a single processor as a whole without any fragmentation thus resulting in no increase in the number of flops, which can be set as the lower bound for distributed-memory settings. However, any fiber whose nonzeros are fragmented among λ processors will incur $2R(\lambda-1)$ additional flops. In the worst-case, if every nonzero of each fiber is assigned to a different processor, then each fiber will have a single nonzero, resulting in a loose upper bound of $3Rm$ total flops following the *if*-statement in Algorithm 2.

The bipartition shown in the bottom part of Fig. 3.2 incurs the fragmentation of 4 out of 6 fibers of $\mathcal{X}(:,:,k)$. So, this bipartition incurs an increase in the total number of fibers from 6 to 10 thus increasing the total number of flops from $2R(24+6)=60R$ to $36R+32R=68R$ during the MTTKRP operations associated with $\mathcal{X}(:,:,k)$.

Since the fine-grain HP-based method described in Section 3.2.2 is not aware of the role of tensor fibers while partitioning the HP model, it may incur a significant amount of fiber fragmentation leading to significant increase in the total computational load.

3.4 Improving Fine-Grain HP Model

3.4.1 A novel vertex weighting scheme

As mentioned earlier, balancing on the flop counts of processors cannot be enforced during partitioning the fine-grain HP model. This is because of the vertex weighting scheme that only encodes balancing nonzero counts of processors while failing to encode the fiber counts.

Here, we propose a novel vertex weighting scheme for estimating correct flop counts of processors during partitioning the fine-grain model. For this purpose, we propose an Inverse-Fiber-Size (IFS) heuristic for estimating the fiber counts of processors. In the IFS scheme, the $2R$ flop contribution of a fiber is distributed uniformly, as vertex weights, among the vertices representing the tensor nonzeros constituting that fiber. That is, a fiber $\mathcal{X}(i,j,:)$ of size $nnz(\mathcal{X}(i,j,:))$ contributes $2R/nnz(\mathcal{X}(i,j,:))$, as a weight, to each vertex representing its constituent nonzeros.

In a given nonzero partition, if the nonzeros of a given fiber are all assigned to the same part, the IFS scheme correctly encodes the contribution of a fiber count $(2R)$ to the respective part. If, however, the nonzeros of a given fiber are fragmented between two parts, then the IFS scheme will incur fractional fiber count contributions to these two parts with a sum of $2R$.

Since the two efficient schemes described in Section 3.2.1 (CSF-S and CSF-D) for computing the MTTKRP have different algorithms and different fiber types, we describe how the IFS scheme is applied to each of them separately. Without loss of generality, we assume that tube fibers ($\mathcal{X}(i,j,:)$) and row fibers ($\mathcal{X}(i,:,k)$) are the tensor's fibers along the longest and second longest modes, respectively.

3.4.1.1 IFS scheme for CSF-S

In this scheme, the tensor is stored only once as fibers of the longest mode. While computing the MTTKRP for $N-1$ modes as described in Algorithm 2, the number of fibers times $2R$ correctly encapsulates the flop count of the Hadamard product and the addition operation involving $\mathbf{B}(jf,:)$ (line 12). On the other hand, while computing the MTTKRP for the longest mode using Algorithm 3, the number of fibers times $2R$ correctly encapsulates the flop count of only the Hadamard product operations (line 9). Therefore, we use the IFS scheme for updating the weights of the vertices as follows: for each $v_{ijk} \in \mathcal{V}$

$$w(v_{ijk}) = 2R + \frac{2R}{nnz(\mathcal{X}(i,j,:))}. \tag{3.4}$$

Here, "$2R$" refers to the number of flops associated with the respective nonzero, whereas $2R/nnz(\mathcal{X}(i,j,:))$ refers to the number of flops associated with the fiber that contains the respective nonzero.

3.4.1.2 IFS scheme for CSF-D

In this scheme, the tensor is stored twice in fiber-centric fashion. As discussed in Section 3.2.1, the first storage utilizes the fibers of the longest mode, while the second storage utilizes the fibers of the second longest mode. For both fiber types, the number of fibers times $2R$ correctly encapsulate flop count of the Hadamard product and addition operations (Algorithm 2 line 12). The distinction is, the number of fibers along the longest mode correctly encapsulates the number of flops during each of the $N-1$ MTTKRP operations performed along all but the longest mode, whereas number of fibers along the second longest mode correctly encapsulates the number of flops during the MTTKRP operation along the longest mode.

A two-constraint formulation is needed for balancing the computational loads of processors in the computational scheme that utilizes CSF-D. This is because, in the CPD-ALS algorithm, MTTKRP operations are performed in different phases interleaved with synchronizing communication operations, and the MTTKRP operations are performed with two different types of fibers.

We use the IFS scheme to compute the two weights of the vertices for the two-constraint formulation as follows: for each $v_{ijk} \in \mathcal{V}$

$$w^1(v_{ijk}) = 2R + \frac{2R}{nnz(\mathcal{X}(i,j,:))} \tag{3.5a}$$

$$w^2(v_{ijk}) = 2R + \frac{2R}{nnz(\mathcal{X}(i,:,k))}. \tag{3.5b}$$

At each iteration, $W^1(\mathcal{V}_p)$ (computed using (2.5)) encodes the computational load of processor p during $N-1$ MTTKRP operations, whereas $W^2(\mathcal{V}_p)$ encodes the computational load of processor p during only one MTTKRP operation. So, the success of this two-constraint scheme depends on giving more importance to the first over second constraint. This can only be achieved by relaxing the maximum allowed imbalance ratio (ϵ) of the second constraint. Unfortunately, the state-of-the-art HP tools do not support different ϵ values for different constraints. For this reason, we propose the following alternative single-constraint weighting scheme that can emulate the above mentioned two-constraint scheme:

$$w(v_{ijk}) = (N-1)w^1(v_{ijk}) + w^2(v_{ijk}) \tag{3.6a}$$

$$= 2RN + \frac{2R(N-1)}{nnz(\mathcal{X}(i,j,:))} + \frac{2R}{nnz(\mathcal{X}(i,:,k))} \tag{3.6b}$$

for each $v_{ijk} \in \mathcal{V}$. In (3.6a), the relative importance of $w^1(v_{ijk})$ over $w^2(v_{ijk})$ is modeled by multiplying $w^1(v_{ijk})$ by $N-1$ as the CSF storage along the longest mode is used in $N-1$ MTTKRP operations at each CPD-ALS iteration. Note that in (3.6a) and (3.6b) the value of N should be set to 3 in case of a 3-mode tensor, but we prefer to use N for a more general presentation.

Algorithm 4 RB-based FG HP with IFS scheme

```
Require: Sparse tensor $\mathcal{X}$
 1: $\mathcal{H} \leftarrow$ Fine-grain hypergraph of $\mathcal{X}$
 2: ▷ $\mathcal{F}$ is the set of nonzero fibers along the longest mode.
 3: $\mathcal{F} \leftarrow \{f_{ij} = \mathcal{X}(i,j,:) : nnz(\mathcal{X}(i,j,:)) \neq 0\}$
 4: if $\mathcal{X}$ is stored as CSF-S then
 5:     RB-STEP-S($\mathcal{H}$, $\mathcal{F}$)
 6: else                                                ▷ $\mathcal{X}$ is stored as CSF-D
 7:     ▷ $\mathcal{F}2$ is the set of nonzero fibers along $2^{nd}$ longest mode.
 8:     $\mathcal{F}2 \leftarrow \{f_{ik} = \mathcal{X}(i,:,k) : nnz(\mathcal{X}(i,:,k)) \neq 0\}$
 9:     RB-STEP-D($\mathcal{H}$, $\mathcal{F}$, $\mathcal{F}2$)
10: function RB-STEP-S($\mathcal{H}$, $\mathcal{F}$)
11:     $\Pi_2 = (\mathcal{V}_L, \mathcal{V}_R) \leftarrow$ BIPARTITION($\mathcal{H}$)
12:     Form $\mathcal{H}_L = (\mathcal{V}_L, \mathcal{N}_L)$ and $\mathcal{H}_R = (\mathcal{V}_R, \mathcal{N}_R)$
13:     $(\mathcal{F}_L, \mathcal{F}_R)$ = SPLIT-FIBERS($\Pi_2$, $\mathcal{F}$)
14:     UPDATE-WEIGHTS-S($\Pi_2, \mathcal{F}_L, \mathcal{F}_R$ )
15:     RB-STEP-S($\mathcal{H}_L$, $\mathcal{F}_L$)
16:     RB-STEP-S($\mathcal{H}_R$, $\mathcal{F}_R$)
17: function RB-STEP-D($\mathcal{H}$, $\mathcal{F}$, $\mathcal{F}2$)
18:     $\Pi_2 = (\mathcal{V}_L, \mathcal{V}_R) \leftarrow$ BIPARTITION($\mathcal{H}$)
19:     Form $\mathcal{H}_L = (\mathcal{V}_L, \mathcal{N}_L)$ and $\mathcal{H}_R = (\mathcal{V}_R, \mathcal{N}_R)$
20:     $(\mathcal{F}_L, \mathcal{F}_R)$ = SPLIT-FIBERS($\mathcal{F}, \Pi_2$)
21:     $(\mathcal{F}2_L, \mathcal{F}2_R)$ = SPLIT-FIBERS($\mathcal{F}2, \Pi_2$)
22:     UPDATE-WEIGHTS-D($\Pi_2$, $\mathcal{F}_L$, $\mathcal{F}_R$, $\mathcal{F}2_L$, $\mathcal{F}2_R$ )
23:     RB-STEP-D($\mathcal{H}_L$, $\mathcal{F}_L$, $\mathcal{F}2_L$)
24:     RB-STEP-D($\mathcal{H}_R$, $\mathcal{F}_R$, $\mathcal{F}2_R$)
```

3.4.2 Improving IFS through utilizing RB

The accuracy of the IFS heuristic depends on keeping track of the correct fibers sizes, which could change significantly as a result of fiber fragmentation during partitioning. We propose to utilize the RB scheme to increase the accuracy of the IFS heuristic in estimating the fiber counts of parts. After each bipartitioning step, the sizes of the fragmented fibers are updated for recomputing the vertex weights according to the IFS heuristic.

Algorithm 5 SPLIT-FIBERS

```
Require: $(\Pi_2, \mathcal{F})$
1: $\mathcal{F}_L, \mathcal{F}_R \leftarrow \emptyset$
2: for all $\mathcal{X}(i,j,:) = f_{ij} \in \mathcal{F}$ do
3:     $f_{ij}^L = \mathcal{X}^L(i,j,:) = \mathcal{X}(i,j,:) \cap \{\mathcal{X}(i,j,k) : v_{ijk} \in \mathcal{V}_L\}$
4:     $f_{ij}^R = \mathcal{X}^R(i,j,:) = \mathcal{X}(i,j,:) \cap \{\mathcal{X}(i,j,k) : v_{ijk} \in \mathcal{V}_R\}$
5:     if $nnz(f_{ij}^L) > 0$ then
6:         $\mathcal{F}_L \leftarrow \mathcal{F}_L \cup \{f_{ij}^L\}$
7:     if $nnz(f_{ij}^R) > 0$ then
8:         $\mathcal{F}_R \leftarrow \mathcal{F}_R \cup \{f_{ij}^R\}$
   return $(\mathcal{F}_L, \mathcal{F}_R)$
```

Algorithm 6 UPDATE-WEIGHTS-S

```
Require: $\Pi_2$ , $\mathcal{F}_L$, $\mathcal{F}_R$
1: for all $v_{ijk} \in \mathcal{V}_L$ do
2:     $w(v_{ijk}) \leftarrow 2R + \frac{2R}{nnz(\mathcal{X}^L(i,j,:))}$
3: for all $v_{ijk} \in \mathcal{V}_R$ do
4:     $w(v_{ijk}) \leftarrow 2R + \frac{2R}{nnz(\mathcal{X}^R(i,j,:))}$
```

Algorithm 4 shows the proposed RB-based IFS scheme. In the algorithm, $\mathcal{H}$ refers to the current hypergraph to be bipartitioned, whereas $\mathcal{F}$ and $\mathcal{F}2$ refer to the current set of nonzero fibers along the first and second longest modes, respectively. The sets of (fragmented) fibers are maintained during the RB scheme for recomputing the vertex weights according to the correct fiber sizes. Note that both $\mathcal{F}$ and $\mathcal{F}2$ are used for the CSF-D scheme while only $\mathcal{F}$ is required for the CSF-S scheme. The algorithm checks whether CSF-S or CSF-D is used and respectively invokes RB-STEP-S or RB-STEP-D accordingly.

In lines 11 and 18 of Algorithm 4, the hypergraph partitioning tool is invoked to obtain a bipartition Π_2 on the vertices of $\mathcal{H}$. In lines 12 and 19, the left hypergraph $\mathcal{H}_L$ and right hypergraph $\mathcal{H}_R$ are constructed according to the net-splitting strategy mentioned in Section 2.4. In line 13, SPLIT-FIBERS function is invoked to form the fiber sets $\mathcal{F}_L$ and $\mathcal{F}_R$ of the left and right parts, for the CSF-S scheme. In lines 20 and 21, SPLIT-FIBERS is invoked to compute $\mathcal{F}_L$ and $\mathcal{F}_R$ as well as $\mathcal{F}2_L$ and $\mathcal{F}2_R$ of the left and right parts, respectively, for the

Algorithm 7 UPDATE-WEIGHTS-D

```
Require: $\Pi_2$, $\mathcal{F}_L$, $\mathcal{F}_R$, $\mathcal{F}2_L$, $\mathcal{F}2_R$
1: for all $v_{ijk} \in \mathcal{V}_L$ do
2:     $w(v_{ijk}) = 2RN + \frac{2R(N-1)}{nnz(\mathcal{X}^L(i,j,:))} + \frac{2R}{nnz(\mathcal{X}^L(i,:,k))}$
3: for all $v_{ijk} \in \mathcal{V}_R$ do
4:     $w(v_{ijk}) = 2RN + \frac{2R(N-1)}{nnz(\mathcal{X}^R(i,j,:))} + \frac{2R}{nnz(\mathcal{X}^R(i,:,k))}$
```

CSF-D scheme.

The SPLIT-FIBERS function (Algorithm 5) implements the fiber fragmentation strategy as follows. The *for*-loop in lines 2-8 computes the intersection of each fiber of the current fiber set $\mathcal{F}$ with the nonzeros corresponding to the vertices of the left and right parts. Then, it assigns an unfragmented fiber to either $\mathcal{F}_L$ or $\mathcal{F}_R$, whereas it adds the subfibers of a fragmented fiber to both $\mathcal{F}_L$ and $\mathcal{F}_R$.

Then, in lines 14 and 22 of Algorithm 4, the vertex weighting scheme is invoked in order to recompute the weights of vertices according to the IFS scheme with correct (fragmented) fiber sizes. Algorithm 6 (UPDATE-WEIGHTS-S) is used to update the weights for CSF-S according to (3.4), whereas Algorithm 7 (UPDATE-WEIGHTS-D) is used to update the weights for CSF-D according to (3.6).

3.4.3 Fiber-net augmentation for reducing total flops

In conventional graph/hypergraph partitioning formulations used for irregular scientific applications in distributed settings, the total amount of computational work is constant. So, in these formulations the partitioning constraint of balancing the part weights correctly corresponds to reducing the computational load of the maximally loaded processor (bottleneck processor). This correspondence will refer to minimizing the computational load of the bottleneck processor as the maximum allowed imbalance ratio (ϵ) is reduced. This is in fact the case for finding a fine-grain partitioning formulation for parallel tensor decomposition which utilizes the COO format for local MTTKRP computations (formulation (3.1)). However, the total amount of computational work is not constant in the fine-grain partitioning formulation that utilizes the CSF format for local MTTKRP computations (formulation (3.2)). Hence, the partitioning constraint of balancing the part weights loosely relates to reducing the computational load of the bottleneck processor.

The partitioning constraint of balancing part weights correctly refers to reducing the computational load of the bottleneck processor if the partitioning formulation targets at reducing the increase in the total computational load due to fiber fragmentation while minimizing the total communication volume. For this purpose, the standard fine-grain hypergraph model, which contains slice nets that encode communication volume, is augmented with fiber nets. Each fiber net connects all vertices corresponding to the nonzeros constituting the fiber.

For the CSF-D scheme, a net n^f_{ij} is created for each nonzero fiber $\mathcal{X}(i, j, :)$ along the longest mode. Similarly, a net n^f_{ik} is created for each nonzero fiber $\mathcal{X}(i, :, k)$ along the second longest mode. The sets of vertices connected by n^f_{ij} and n^f_{ik} are respectively defined as:

$$Pins(n^f_{ij}) = \{v_{ijk} : \mathcal{X}(i,j,k) \neq 0 \;\; \forall k \in \{1, \ldots, K\}\} \tag{3.7a}$$

$$Pins(n^f_{ik}) = \{v_{ijk} : \mathcal{X}(i,j,k) \neq 0 \;\; \forall j \in \{1, \ldots, J\}\}. \tag{3.7b}$$

For the CSF-S scheme, constructing the nets for the longest-mode fibers suffices, and the set of vertices connected by each fiber net is the same as in (3.7a).

The fiber-net augmentation can be easily integrated into the RB-based framework given in Algorithm 4. After constructing the hypergraph model, the sets of fibers $\mathcal{F}$ (line 3) and $\mathcal{F}2$ (line 8) provide the sufficient nonzero-to-fiber relations that can be used to construct the fiber nets. No other modifications are needed in the RB-STEP routines.

Consider a partition Π of an augmented hypergraph for the CSF-S scheme. In Π, a cut slice net n^s with connectivity $\lambda(n^s)$ will incur a communication of $R(\lambda(n^s)-1)$ words during each MTTKRP operation as in the standard fine-grain hypergraph model. In Π, internal fiber nets do not incur any increase in the total number of flops. However, a cut fiber net n^f with connectivity $\lambda(n^f)$ encodes an increase of $2R(\lambda(n^f) - 1)$ flops during each MTTKRP operation. A similar discussion holds for the CSF-D scheme.

For the CSF-S scheme, the cost of fiber nets along the longest mode is set to $2R$, whereas the cost of slice nets is set to αR. For the CSF-D scheme, the cost of fiber nets along the longest and second longest modes are set to $2R(N-1)$ and $2R$ respectively, whereas the cost of slice nets are set to αRN. Here, α refers to the scaling factor between the cost of increasing the communication volume by R words and the cost of increasing the total flop count by $2R$.

In the augmented fine-grain hypergraph model, the partitioning objective of minimizing the cut size will simultaneously encode minimizing both the communication volume and the increase in total flop count. The partitioning constraint of maintaining balance on part weighs (according to the proposed vertex weighting schemes described in Section 3.4.1) will encode minimizing the flop count of the bottleneck processor with decreasing ϵ because of the proposed fiber-net augmentation.

The augmentation of fiber nets is also expected to contribute to improving the accuracy of the IFS scheme. Reducing the number of cut fiber nets relates to maximizing the number of internal nets, where internal nets correspond to unfragmented fibers. So, increasing the number of unfragmented fibers enables the IFS scheme to correctly encode the contribution of larger number of fibers to the part weights. So, the objective of reducing fiber fragmentation decreases the number of erroneous vertex weight contributions incurred by fragmented fibers. This decrease is expected to improve the accuracy of the IFS scheme thus leading to better load balancing.

3.5 Experiments

3.5.1 Setting

There are several successful hypergraph partitioning tools [52, 55, 56]. We use PaToH [52] (version 3.2) in speed mode and the value of ϵ is set to 0.10. Since PaToH contains randomized algorithms, we partition each tensor three times for each partitioning method, and we report the average of the three instances.

The topologies of the hypergraph models are orthogonal to the value of R. On the other hand, the vertex weighting schemes as well as the net costs presented in this chapter involve R, which acts as a scaling factor, for the sake of clarity of presentation. Thus, removing this scaling factor affects neither the cutsize nor the balancing qualities, so in our partitioning implementation the R value is set to one.

For the parallel experiments, we use the parallel CPD-ALS code developed and used in the work by Acer et al. [30]. The code is implemented in C, uses MPI for interprocess communication and compiled with `gcc` version 8.3.0 using `O3` optimization flag. The MTTKRP implementation in the code is based on the flop-efficient formulation in (3.2), which is identical to CSF-D. We have modified the code to include CSF-S. The runtimes of CPD-ALS are reported as per-iteration times by taking the average of total runtime of 50 iterations.

Our parallel experiments are conducted on Bull Sequana X1000 system. A node in this system operates on dual Intel Xeon Skylake 8168 with total of 48 cores, 96 GB of memory and 2.70 GHz clock frequency. The nodes are connected with the high speed network EDR-Infiniband (Connect-X4).

Table 3.1: Properties of Test Tensors

	size of dimensions					
Tensor	I	J	K	L	nnz	Density
`Enron`	6.0K	5.6K	244.2K	1.1K	54.2M	$5.5 \cdot 10^{-9}$
`Flickr`	319.6K	28.1M	1.6M	730	112.9M	$1.1 \cdot 10^{-14}$
`Movies-amazon`	87.8K	4.4K	226.5K	—	15.0M	$1.7 \cdot 10^{-7}$
`Nell-1`	2.9M	2.1M	25.5M	—	143.6M	$9.1 \cdot 10^{-13}$
`Nell-2`	12.1K	9.2K	28.8K	—	76.8M	$2.4 \cdot 10^{-5}$
`Yelp`	686.5K	85.5K	773.2K	—	185.5M	$4.1 \cdot 10^{-9}$

3.5.2 Dataset

Our dataset is composed of six real-world sparse tensors commonly used as a benchmark for parallel sparse tensor research. Table 3.1 shows the properties of the tensors. `Enron` [57] consists of words of email exchanges in the form of *sender-receiver-word-date* quadruplets. It has been used with tensor decomposition methods for social network analysis and link prediction [58]. `Flickr` is a binary tensor representing *user-image-tag-date* quadruplets, which was first crawled by Görlitz et al. [59] from flickr.com.

Bhargava et al. [60] factorize `Flickr` using CPD-ALS for forming multi-dimensional collaborative recommendations. `Movies-amazon` represents *user-movie-word* triplets extracted from the user reviews of movies in Amazon [61]. `Movies-amazon` is one of the datasets used for evaluating recommender systems research, including CPD-based systems. `Nell-1` and `Nell-2` [62] represent *entity-relation-entity* tuples of the Never Ending Language Learner knowledge base. Kang et al. [63] used both tensors for concept discovery and contextual synonym detection using CPD-ALS. `Yelp` contains *user-business-word* triplets obtained from business reviews in Yelp academic dataset [1]. Yelp is generally used in the context of tensor decomposition for community detection and recommender systems.

[1]https://www.yelp.com/dataset/challenge

3.5.3 Performance comparison

We compare the performance of the proposed improvement schemes against the baseline FG method in terms of computational and communication cost metrics as well as parallel MTTKRP and CPD-ALS times on the 6 tensors given in Table 3.1. We use $P = 512$ and $\alpha = 10$ in all tables unless specified otherwise.

The computational cost metrics consist of maximum and average number of flops performed by a processor. The communication cost metrics consist of maximum and average send volume handled by a processor. The latency-based communication cost metrics regarding maximum and average number of messages sent by a processor are not reported as all methods display almost the same performance on these metrics. Here, average flop count and average volume values refer to the total flop count and total volume values, respectively, divided by the number of processors. We prefer to report average values instead of total values because the former give a better view on the deviation of maximum from average. When a normalized value is presented, it means the value of the respective method divided by that of other method (usually the baseline). Since we aim at minimizing all performance metrics considered in this chapter, a normalized value of < 1 means an improvement over the baseline, and deterioration otherwise.

3.5.3.1 Results of CSF-S experiments

Table 3.2 displays the performance improvement rates attained by the optimization schemes in Section 3.4 in an incremental way. Note that "Avg." row at the bottom of the table and all other tables refers to the geometric mean. As seen in the table, on average, utilizing IFS for vertex weighting (Algorithm 4) in FG+ improves the maximum and average flop counts by 8.0% and 4.0%, respectively, compared to FG. Fiber-net augmentation used in FG++ significantly decreases maximum and average flop counts respectively by 16.3% and 14.6% compared to FG+. As seen in the table, utilizing the two optimization schemes in FG++ leads to a significant decrease in the maximum and average flop counts respectively by

23.2% and 17.8% compared to the baseline FG method.

As seen in Table 3.2, in terms of communication volume metrics, FG+ attains slightly better performance compared to FG. That is, FG+ reduces the maximum and average communication volumes by 6.2% and 7.0% compared to FG, on average. Comparing FG++ against FG+ shows that although they display comparable performance in terms of average communication volume, FG++ achieves considerably better performance in terms of maximum communication volume by an amount of 9.6%. As seen in the table, FG++ achieves a considerable decrease in the maximum and average communication volume respectively by 15.5% and 7.3% compared to the baseline FG method. These findings show that the use of fiber nets do not lead to performance degradation in communication volume metrics. This can be attributed to the fact that fiber nets are subnets of the slice nets. Relatively better performance obtained by FG++ against FG in terms of maximum volume compared to average volume can be attributed to the expectation that better computational load balancing achieved by FG++ leads to a better communication volume balancing. Here and hereafter, the proposed FG++ will be referred to as impFG.

Table 3.2: Performance comparison in terms of computational and communication cost metrics on $P = 512$ processors for CSF-S.

	Actual values (in terms of R)				Normalized with respect to FG							
	FG				FG+IFS (FG+)				FG+IFS+FNA (FG++ $\equiv$ impFG)			
	flops		comm. vol.		flops		comm. vol.		flops		comm. vol.	
Tensor	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg
`Enron`	718,123	648,890	43,723	22,436	0.99	0.91	1.00	0.93	0.72	0.70	0.86	0.81
`Flickr`	1,477,230	1,157,132	105,871	52,912	0.80	0.99	0.86	0.96	0.75	0.93	0.75	0.85
`Movies-amazon`	126,153	107,987	40,290	22,137	0.93	0.96	1.03	0.89	0.87	0.92	1.00	0.94
`Nell-1`	1,491,700	1,421,460	409,820	279,581	0.98	0.98	0.98	0.98	0.82	0.84	0.92	1.04
`Nell-2`	769,255	734,175	82,056	40,171	1.00	0.98	0.90	0.92	0.74	0.73	0.79	1.15
`Yelp`	1,689,961	1,320,840	384,328	113,941	0.85	0.94	0.86	0.93	0.71	0.85	0.79	0.82
Avg.	**798,660**	**694,041**	**115,793**	**56,814**	**0.92**	**0.96**	**0.94**	**0.93**	**0.77**	**0.82**	**0.85**	**0.93**

IFS: Inverse-Fiber-Size for vertex weighting with RB (Secs. 3.4.1 & 3.4.2); FNA: Fiber Net Augmentation (Sec. 3.4.3) with $\alpha = 10$.

Table 3.3 shows how the above-mentioned performance improvements lead to improving the actual parallel runtimes. In the table, the values under FG are actual runtimes, while those under impFG are normalized with respect to those of FG. Under MTTKRP tab, the "comp" column refers to the computational part of the MTTKRP operation, whereas "tot" refers to the total runtime of the MTTKRP operation including communication. Comparing "max flops" column of impFG in Table 3.2 with the "comp" column of impFG in Table 3.3 show that there exist close correlation between the amount of improvement in maximum flop count and the amount of improvement in parallel MTTKRP computation time. That is, the 23% improvement attained by impFG in maximum flop count reflects as approximately 22% improvement in parallel MTTKRP computation time. In fact, this close relation also applies to individual tensors except for `Nell-2`. For example, 28%, 25%, 13%, 18% and 29% reduction in max flop counts obtained by impFG for the tensors `Enron, Flickr, Amazon, Nell-1` and `Yelp` respectively reflect as approximately 36%, 23%, 11%, 15% and 32% improvement in parallel MTTKRP computation times. This confirms the validity of the maximum flop count metric in determining the parallel computation time.

Table 3.3 also shows relative runtime performance variation of impFG over FG with increasing R. We use the same partitioned tensor, for each tensor, to obtain the parallel running times with different R values. Keep in mind that with increasing R value, the improvement ratios of impFG over FG remain the same in terms of computational and communication cost metrics (as in Table 3.2). As seen in the normalized columns of Table 3.3, the relative performance of impFG over FG slightly increases with increasing R in terms of both parallel MTTKRP and CPD-ALS runtimes. This is expected because, with increasing R, while latency-based communication costs remain the same, communication volume and computational costs increase.

Table 3.4 shows the effect of augmenting fiber nets on the total communication volume along the longest mode as well as the other $N-1$ modes. In the table, the values under FG are actual communication volume values (in words), while those under impFG are normalized with respect to those of FG. Comparing the relative performance of impFG over FG, the fiber-net augmentation along the

Table 3.3: Performance comparison in terms of parallel runtimes on $P = 512$ processors for CSF-S.

Tensor	R	FG (times in ms)			impFG (normalized)		
		MTTKRP		CP-ALS	MTTKRP		CP-ALS
		comp	tot		comp	tot	
	32	55.9	152.4	156.8	0.68	0.83	0.83
`Enron`	64	124.0	245.0	260.9	0.64	0.77	0.77
	128	302.2	477.5	527.6	0.58	0.69	0.72
	32	128.5	246.6	330.1	0.77	0.84	0.93
`Flickr`	64	246.3	435.3	628.8	0.77	0.80	0.92
	128	485.0	824.9	1,312.1	0.76	0.80	0.95
	32	12.5	92.2	100.0	0.91	0.93	0.93
`Movies-a`	64	28.2	135.5	153.4	0.89	0.93	0.93
	128	60.4	218.7	276.3	0.89	0.93	0.92
	32	295.3	718.7	888.4	0.86	0.81	0.85
`Nell-1`	64	570.1	1,422.2	1,772.8	0.85	0.82	0.87
	128	1,249.0	3,106.2	3,975.8	0.84	0.85	0.89
	32	67.7	165.7	176.4	0.99	1.02	1.03
`Nell-2`	64	159.7	295.4	325.2	0.95	0.99	0.99
	128	419.8	643.6	733.1	0.89	0.93	0.94
	32	202.8	386.2	457.6	0.63	0.75	0.74
`Yelp`	64	379.3	716.5	891.3	0.66	0.73	0.72
	128	725.8	1,365.8	1,861.1	0.72	0.76	0.74
	32	**84.5**	**232.9**	**268.1**	**0.79**	**0.86**	**0.88**
Avg.	64	**176.0**	**404.0**	**484.5**	**0.78**	**0.84**	**0.86**
	128	**387.2**	**785.7**	**1,006.2**	**0.77**	**0.82**	**0.85**

Table 3.4: Performance comparison in terms of total volume during MTTKRP along the longest mode and other $N-1$ modes on $P = 512$ processors for CSF-S.

Tensor	FG (total volume, in terms of R)		impFG (normalized)	
	Longest mode	$N-1$ modes	Longest mode	$N-1$ modes
`Enron`	4,476,252	7,006,890	1.06	0.65
`Flickr`	968,022	26,119,510	1.11	0.84
`Movies-a`	4,926,116	6,405,454	1.09	0.83
`Nell-1`	55,605,248	87,536,435	1.24	0.91
`Nell-2`	10,726,748	9,837,220	1.37	0.91
`Yelp`	18,634,076	39,700,541	1.10	0.68
Avg.	**7,868,030**	**18,499,059**	**1.16**	**0.80**

Table 3.5: Computational and communication cost metrics (in terms of R) of the impFG method with different α values on $P = 512$ for CSF-S.

	α	flops		comm. vol.	
		max	avg	max	avg
Avg. of all tensors	5	$600,591$	$564,647$	$99,463$	$54,371$
	10	$613,372$	$572,075$	$97,739$	$52,903$
	50	$623,855$	$578,992$	$94,929$	$51,453$
	100	$626,631$	$580,123$	$95,516$	$51,085$

longest mode incurs an increase in communication volume during the MTTKRP operation along that mode, whereas it achieves a decrease in communication volume during MTTKRP operations along all other $N-1$ modes. As seen in the table, on average, impFG incurs 16% increase in total volume during MTTKRP along the longest mode, whereas it achieves 20% decrease in that along all other $N-1$ modes.

The above-mentioned experimental finding can be attributed to the fact that the nets representing the fibers along the longest mode are subnets of the nets that represent slices of the other $N-1$ modes. That is, trying to keep fiber nets along the longest mode internal can be expected to increase the possibility of keeping the nets representing slices along other $N-1$ modes internal as well. Recall that the communication volume during MTTKRP operations along different modes differ depending on the number and connectivities of the cut nets representing slices along those modes. As seen in the last column of Table 3.3, impFG achieves an average decrease of 7% compared to FG in total volume during all MTTKRP operations. However, for some tensors such as `Nell-1` and `Nell-2`, fiber-net augmentation respectively incurs overall communication volume increase of 4% and 15%. This is because for FG on `Nell-1` and `Nell-2`, the total volume along the longest mode is larger than or very close to that of all other $N-1$ modes. In other tensors, such as `Flickr`, the communication volume along all $N-1$ modes is significantly larger than that of the longest mode. Therefore, the overall improvement achieved by fiber-net augmentation depends on two factors; the relative communication volume during the longest and $N-1$ modes, and the increase/decrease incurred/achieved along the longest and other $N-1$ modes.

Table 3.5 shows the effect of different α values on the performance of fiber-net augmentation in impFG. We ran the impFG method with $\alpha = 5$, 10, 50 and 100. We report the average flop count and communication volume statistics in Table 3.5 as actual values. As seen in the table, increasing the α value (giving more importance to decreasing total communication volume over decreasing fiber fragmentation) results in increasing the total flops while the communication volume is decreased. As a trade-off between total flops and communication volume, we choose $\alpha = 10$ for the rest of tables and figures in this section.

3.5.3.2 Results of CSF-D experiments

The performance comparisons in the previous section regarding the incremental performance improvement attained by the optimization schemes in Section 3.4, the effect of different R values as well as the effect of α value apply to the CSF-D scheme as well. In order to present a wider spectrum of results, here we study the effect of applying the CSF-S-based and the CSF-D-based optimization schemes on the computational and total volume cost metrics for the CSF-D-based MTTKRP. We perform this in order to justify proposing a separate optimization techniques for CSF-D. Here and hereafter, the superscripts ‘S’ and ‘D’ will be used to distinguish the CSF-S-based and CSF-D-based optimization schemes on

Table 3.6: Performance comparison in terms of computational and total volume metrics on $P = 512$ processors for CSF-D.

	FG + IFS^S +FNA^S (impFG^S)									FG+IFS^D+FNA^D (impFG^D)								
	Longest mode			$N-1$ modes			All modes			Longest mode			$N-1$ modes			All modes		
	flops		tot. vol.	flops		tot. vol.	flops		tot. vol.	flops		tot. vol.	flops		tot. vol.	flops		tot. vol.
Tensor	max	avg		max	avg		max	avg		max	avg		max	avg		max	avg	
`Enron`	1.06	1.00	1.06	0.72	0.70	0.65	0.82	0.79	0.81	1.01	0.97	1.07	0.74	0.71	0.65	0.82	0.79	0.81
`Flickr`	1.23	1.00	1.11	0.75	0.93	0.84	0.86	0.95	0.85	1.10	1.00	0.46	0.81	0.94	0.85	0.88	0.95	0.84
`Movies-amazon`	1.04	1.00	1.09	0.87	0.92	0.83	0.94	0.95	0.94	0.87	0.72	0.94	0.85	0.92	0.76	0.86	0.83	0.84
`Nell-1`	1.18	1.00	1.24	0.82	0.84	0.91	0.95	0.89	1.04	1.05	0.95	1.21	0.84	0.86	0.92	0.91	0.89	1.04
`Nell-2`	1.01	1.02	1.37	0.74	0.73	0.91	0.84	0.83	1.15	0.81	0.69	0.96	0.74	0.71	0.66	0.77	0.70	0.81
`Yelp`	1.01	1.00	1.10	0.71	0.85	0.68	0.83	0.92	0.82	0.81	0.71	0.97	0.73	0.87	0.74	0.76	0.80	0.81
Avg.	**1.09**	**1.00**	**1.16**	**0.77**	**0.82**	**0.80**	**0.87**	**0.89**	**0.93**	**0.93**	**0.83**	**0.90**	**0.78**	**0.83**	**0.76**	**0.83**	**0.82**	**0.86**

IFS^S, FNA^S and impFG^S denote the improvement schemes denoted at the bottom of Table 3.2

the FG method. All the experiments in this section utilize the CSF-D-based MTTKRP regardless of the type of optimization scheme applied.

Table 3.6 compares impFG^D against impFG^S in terms of computational and total communication volume statistics normalized with respect to those of FG. In the table, these statistics are detailed along the longest mode, remaining $N-1$ modes, and all modes. As seen in the table, on average, utilizing the IFS scheme and fiber-net augmentation for the longest mode only (CSF-S-based improvements) in the impFG^S method improves the maximum and average flop counts by 13.0% and 11.0%, respectively, compared to the FG method in all modes. Utilizing the IFS scheme and fiber-net augmentation for both longest and second longest modes (CSF-D-based improvements) in the impFG^D method improves the maximum and average flop counts by 4.6% and 7.8%, respectively, compared to the impFG^S method in all modes. Although the maximum and average flop counts along $N-1$ modes are almost the same for both impFG^S and impFG^D, the relative improvements in all modes come from improving the maximum and average flop counts along the longest mode by 14.6% and 17.0%, respectively. As also seen in the table, the impFG^D method respectively achieves 17.0% and 18.0% improvements in maximum and average flop counts compared to the baseline FG method.

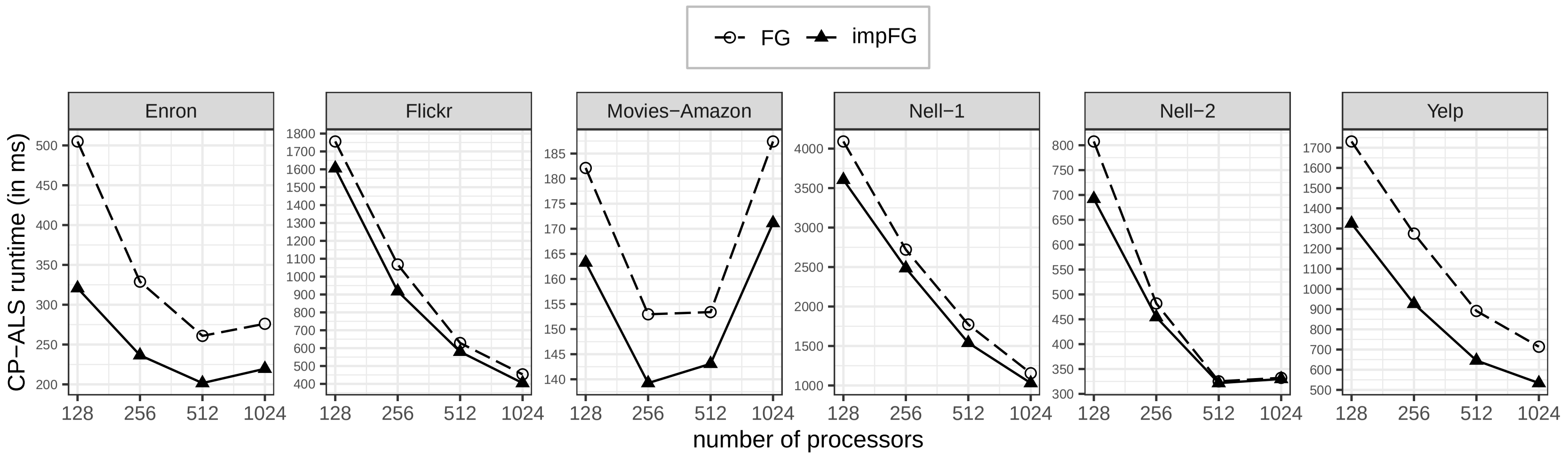
Comparison in terms of total volume metric shows a similar behavior as the comparison in terms of computational cost metrics discussed above. That is, utilizing the CSF-S-based improvements for the CSF-D-based MTTKRP in the impFG$^{\mathrm{S}}$ method incurs an increase of 16.0%, on average, in total volume along the longest mode compared to FG. On the other hand, impFG$^{\mathrm{D}}$ achieves an improvement of 10.0% in total volume during the MTTKRP along the longest mode as a result of augmenting the fiber nets along the second longest mode. The effect of this improvement can be seen in the table as 7.5% improvement in terms of total volume of impFG$^{\mathrm{D}}$ compared to impFG$^{\mathrm{S}}$ along all modes. As seen in the table, impFG$^{\mathrm{D}}$ achieves 14% improvement in terms of total volume compared to FG along all modes.

Table 3.7: Performance comparison in terms of parallel runtimes on $P = 512$ processors for CSF-D with $R = 64$.

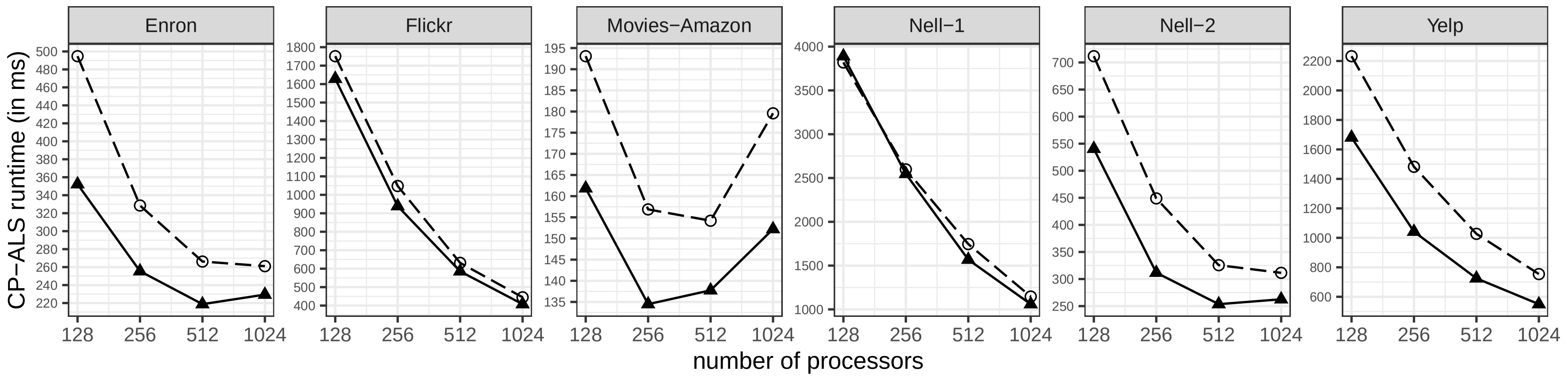
	FG (in ms)			impFG$^{\mathrm{D}}$ (normalized)		
	MTTKRP		CP-ALS	MTTKRP		CP-ALS
Tensor	comp	tot		comp	tot	
`Enron`	131.2	252.7	266.2	0.73	0.81	0.82
`Flickr`	252.0	443.6	631.5	0.81	0.84	0.93
`Movies-amazon`	29.5	136.1	154.2	0.90	0.88	0.89
`Nell-1`	521.6	1,393.8	1,746.0	0.92	0.83	0.90
`Nell-2`	155.7	296.2	325.6	0.71	0.78	0.78
`Yelp`	502.7	851.8	1,027.2	0.69	0.73	0.71
Avg.	**184.78**	**418.40**	**497.36**	**0.79**	**0.81**	**0.83**

Table 3.7 shows how the performance improvement achieved by the proposed impFG$^{\mathrm{D}}$ method in computational and total volume metrics lead to improvements in actual parallel runtimes. In the table, the values under FG are actual runtimes, while those under impFG$^{\mathrm{D}}$ are normalized with respect to those of FG. Comparing "max flop" column of impFG$^{\mathrm{D}}$ in Table 3.6 (along all modes) with the "MTTKRP comp" column of impFG$^{\mathrm{D}}$ in Table 3.7 shows the close correlation between the amount of improvement in maximum flop count and the amount of improvement in parallel MTTKRP computation time. That is, the 17.0% improvement attained by impFG$^{\mathrm{D}}$ in maximum flop count reflects as approximately 21.0% improvement in parallel MTTKRP computation time, on average. As also seen in the table, the CPD-ALS runtime improves, on average, by 17.0% as a result of applying the optimization schemes for CSF-D.

Figures 3.3a and 3.3b respectively display the strong scaling curves of impFG vs FG and impFG$^{\mathrm{D}}$ vs FG. Note that in 3.3a the CSF-S scheme is utilized for computing the MTTKRP, whereas in 3.3b the CSF-D is utilized instead. The curves display parallel runtimes of the CPD-ALS algorithm on $P = 128$ up to $P = 1024$ processors with $R = 64$. As seen in the figure, impFG increases the scalability of FG for both CSF-S and CSF-D schemes on all tensors. The relative scalability between impFG and FG for CSF-S and CSF-D schemes shows similar trend for all tensors, except for `Nell-2` which favors the CSF-D scheme. That is, for `Nell-2`, although impFG and FG show very close scaling performance for CSF-S scheme, impFG$^{\mathrm{D}}$ displays significantly better performance than FG for CSF-D. A grasp of the actual runtime values can be taken from comparing the values of the CPD-ALS column in Table 3.3 for $R = 64$ with the same column values in Table 3.7.



(a) CSF-S



(b) CSF-D

Figure 3.3: Strong scaling curves for parallel CPD-ALS obtained by FG and impFG using (a) CSF-S and (b) CSF-D

3.6 Related Work

In the literature, there are various CPD-ALS implementations adopting different parallelism paradigms [32, 36, 53, 63–66]. On distributed-memory systems, DMS [53] is the most commonly-used implementation. DMS adopts a multi-dimensional cartesian partitioning approach, however it does not support different partitioning techniques coming in more irregular forms.

To devise intelligent tensor partitioning models, sparse matrix partitioning community adapted well-known sparse matrix partitioning models for tensors. These models came in different granularities: coarse-grain [28], multi-dimensional cartesian model [30], fine-grain [28] and medium-grain [31]. The multidimensional cartesian model is derived from the hypergraph model proposed earlier for 2D checkerboard partitioning of sparse matrices [67, 68]. The fine-grain model can be considered as an extension of the fine-grain hypergraph model for 2D nonzero-based sparse matrix partitioning [67, 69, 70] to multi-dimensional tensor partitioning. The recent general medium-grain model [31] can be considered as an extension of the medium-grain model for 2D sparse matrix partitioning [71] to tensors. Among these, fine-grain model achieves the minimum communication volume as well as the best computational balance on the tensor nonzeros assigned to processors.

Sparse tensor storage formats include COO (coordinate) [28], CSF (compressed sparse fiber) [33], and HiCOO (hierarchical coordinate) [34]. COO corresponds to a list of tensor nonzeros, where each nonzero represented by a list of indices and the value. Besides its simplicity, COO stores repeated indices (within a fiber or a slice) redundantly and the MTTKRP on COO performs redundant flops (see section 3.3.1). CSF and HiCOO are motivated by reducing the storage used by COO, due to the limited memory in shared-memory architectures. While the (sequential) MTTKRP algorithm on HiCOO has the same flop count as that on COO, the algorithm on CSF achieves a much better flop count compared to those on COO-based formats. This improvement in the flop count makes CSF the most favorable alternative for the local MTTRKP computation on distributed-memory systems.

3.7 Conclusion

We proposed two improvement schemes to the existing fine-grain hypergraph model in order to address the deficiencies introduced by utilizing the CSF-oriented MTTKRP for distributed-memory CPD-ALS computation. The improvement schemes target at achieving true computational load balancing among processors, thus leading to faster parallel runtime. The improvement schemes do not deteriorate the communication overhead. In fact, the total volume overhead decreases as a result of better load balancing, while the latency overhead stays the same as that of the FG method. On average, applying the proposed improvement schemes to the FG method improves the parallel MTTKRP computation time and the overall CPD-ALS time respectively by 22.0% and 14.0% on 512 processors, and with similar percentages on 128, 256 and 1024. As future work, we plan to extend the proposed true balancing method for other nonzero-based tensor partitioning models.

Chapter 4

Latency Hiding in Distributed Sparse Tensor Decomposition

4.1 Overview

In this chapter, we propose hiding the latency overhead of sparse expand and reduce operations of CPD-ALS by embedding them into `ALL-REDUCE`. Although CPD-ALS has an `ALL-REDUCE` for each sparse expand and reduce communication, it is not possible to embed each sparse expand/reduce due to the dependencies between the sparse operations and `ALL-REDUCE`. We propose a novel computation/communication rearrangement scheme of the CPD-ALS that removes the dependencies and enables embedding each of the sparse expand/reduce operations into an `ALL-REDUCE`.

We use the hypercube-based `ALL-REDUCE` which utilizes the E-cube routing for embedding and we denote the embedding scheme by EMB hereafter. The utilized hypercube topology is virtual and transparent to the actual network topology of the target system. In the naive implementation of EMB, each individual P2P message of a sparse expand/reduce operation is considered separately. This may lead to multiple copies of the same expanded/reduced factor-matrix row be in the same message between two processors during the embedded-`ALL-REDUCE`. Therefore, we propose an expand-and-reduce-aware embedding in which each message contains only one copy of a factor-matrix row in each step of `ALL-REDUCE`. We also extend the existing communication duality between sparse reduce and expand operations into EMB by proposing to use increasing dimension E-cube routing during the expand-embedded-`ALL-REDUCE`, while using decreasing dimension E-cube routing during reduce-embedded-`ALL-REDUCE`, or vice versa.

The proposed EMB totally avoids the latency overhead associated with the sparse expand and reduce operations and reduces both maximum and average number of messages handled by a processor to $2\log_2 K$ for each MTTKRP for a K-processor system independent of the sparsity pattern of the tensor. The only trade-off between the proposed EMB and conventional P2P schemes is the increase in the communication volume incurred by embedding the P2P communications into the `ALL-REDUCE` communications.

In order to model the communication requirement of EMB, we define a concurrent communication cost metric which counts how many times each shared factor-matrix row is concurrently communicated along hypercube dimensions during the E-cube routing. Then we propose a novel recursive bipartitioning (RB) framework that enables simultaneous hypergraph partitioning (HP) and subhypergraph-to-subhypercube mapping to achieve task-to-processor assignment which encodes minimizing the concurrent communication volume metric. In this HP model, we propose and use sibling subnet removal and net-anchoring schemes at each level of RB. We also propose a novel bin-packing adaptation for the factor-matrix row to processor assignment in order to minimize the maximum volume handled by a processor during both expand-embedded and reduce-embedded `ALL-REDUCE` operations. The proposed extension of duality to EMB enables the proposed bin-packing to encode the minimization of maximum volume for only one sparse embedding which holds for the other.

Experimental results with thirteen tensors on up to 4096 processors show the validity of the proposed models and methods. These results show that EMB scales well up to 4096 processors, whereas state-of-the-art P2P scales down after

1024 processors.

The rest of the chapter is organized as follows: Sec. 4.2 contains the communication details of the nonzero-based parallel CPD-ALS algorithm. The proposed rearrangement scheme that enables embedding is discussed in Sec. 4.3. Sec. 4.4 presents the proposed embedding scheme. The proposed RB-based HP model for task-to-processor assignment is described in Sec. 4.5. Sec. 4.6 displays and discusses the experimental results. The related work is given in Sec. 4.7. Finally, Sec. 4.8 concludes the chapter.

4.2 Preliminaries: Parallel CPD-ALS

We adopt nonzero-based parallelization of CPD-ALS. In this parallelization, tensor nonzeros are distributed among processors and processors locally compute (partial) results for factor matrices using those nonzeros according to the owner-computes rule. For processor p_k, the factor matrix rows are classified into three categories according to the tensor nonzeros distribution as follows: Factor-matrix row r_i is said to be local if the nonzeros that contribute to its computation reside in p_k. r_i is said to be local-shared if the nonzeros that contribute to its computation reside in a set of sharing processors $\hat{S}_i \subseteq P, |\hat{S}_i| > 1 \wedge p_k \in \hat{S}_i$, and the processor responsible for holding the final value of r_i is p_k. In such case, p_k is called the owner of r_i and denoted by $owner(r_i)$. We use $S_i = \hat{S}_i \setminus owner(r_i)$ to identify the set of sharing processors without the owner. A local factor matrix that contains local and local-shared rows is denoted by $\mathbf{A}_k$. r_i is said to be nonlocal if p_k has one or more nonzeros that contribute to its computation but p_k is not its owner. A local factor matrix that contains $\mathbf{A}_k$ in addition to nonlocal rows is distinguished by the hat notation as $\hat{\mathbf{A}}_k$.

We use 3-mode tensors here and in Sec. 4.3 for a convenient presentation. The discussions easily extend to higher dimensional tensors (i.e., $N > 3$). Algorithm 8 describes the parallel CPD-ALS for 3-mode tensors. The communication requirement in this algorithm is detailed for updating $\mathbf{A}$ per processor p_k as follows.

After the local MTTKRP (line 3), partial results of local-shared factor matrix rows are received while partial results of nonlocal rows are sent to their owner processors. The received partial results are reduced using an associative operation to form the up-to-date local-shared rows. This communication operation is referred to as *sparse reduce*. Using the up-to-date local and local-shared factor matrix values, the product in line 5 can be computed locally. Then, column normalization requires computing λ that depends on all factor matrix columns through `ALL-REDUCE` in line 6. The normalized local-shared row r_i is needed by the processors in S_i for the computation of the factor matrix along the next tensor mode. Therefore, the local-shared rows are sent (expanded) to and the nonlocal rows are received from their respective owner processors (line 7). This operation is referred to as *sparse expand*. Finally, the partial $\mathbf{A}^\top\mathbf{A}$ product can be computed locally using local and local-shared rows and an `ALL-REDUCE` operation is used for computing the final product (line 8).

Algorithm 8 Parallel CPD-ALS ($\mathcal{X}$) for 3-mode Tensors

```
 1: Randomly initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
 2: while not converged do
 3:     $\hat{\mathbf{A}}'_k \leftarrow \mathbf{X}_k^{(1)}(\hat{\mathbf{B}}_k \odot \hat{\mathbf{C}}_k)$          ▷ MTTKRP
 4:     Sparse REDUCE on shared $\mathbf{A}$-matrix rows
 5:     $\mathbf{A}_k \leftarrow \mathbf{A}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{B}^\top\mathbf{B})^{-1}$
 6:     ALL-REDUCE to normalize cols of $\mathbf{A}$ into $\lambda$
 7:     Sparse EXPAND on shared $\mathbf{A}$-matrix rows
 8:     ALL-REDUCE to compute $\mathbf{A}^\top\mathbf{A}$
 9:     $\hat{\mathbf{B}}'_k \leftarrow \mathbf{X}_k^{(2)}(\hat{\mathbf{A}}_k \odot \hat{\mathbf{C}}_k)$          ▷ MTTKRP
10:     Sparse REDUCE on shared $\hat{\mathbf{B}}$-matrix rows
11:     $\mathbf{B}_k \leftarrow \mathbf{B}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{A}^\top\mathbf{A})^{-1}$
12:     ALL-REDUCE to normalize cols of $\mathbf{B}$ into $\lambda$
13:     Sparse EXPAND on shared $\mathbf{B}$-matrix rows
14:     ALL-REDUCE to compute $\mathbf{B}^\top\mathbf{B}$
15:     $\hat{\mathbf{C}}'_k \leftarrow \mathbf{X}_k^{(3)}(\hat{\mathbf{A}}_k \odot \hat{\mathbf{B}}_k)$          ▷ MTTKRP
16:     Sparse REDUCE on shared $\hat{\mathbf{C}}$-matrix rows
17:     $\mathbf{C}_k \leftarrow \mathbf{C}'_k(\mathbf{B}^\top\mathbf{B} * \mathbf{A}^\top\mathbf{A})^{-1}$
18:     ALL-REDUCE to normalize cols of $\mathbf{C}$ into $\lambda$
19:     Sparse EXPAND on shared $\mathbf{C}$-matrix rows
20:     ALL-REDUCE to compute $\mathbf{C}^\top\mathbf{C}$
    return $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$
```

4.3 Rearrangement of Parallel CPD-ALS to Enable Embedding

In the parallel CPD-ALS shown in Algorithm 8, there are two sparse reduce and expand operations per tensor mode to satisfy the computational requirement of the MTTKRP operation. The dual sparse reduce and expand operations (respectively in lines 4 and 7) are performed to complete the computation of local and local-shared **A**-matrix rows. Similarly, the dual sparse reduce and expand in lines 10, 13 and lines 16, 19 do so respectively for **B**- and **C**-matrix rows. Furthermore, there are two `ALL-REDUCE` operations attached with the computation of factor matrices along each mode. Despite having an `ALL-REDUCE` for each sparse expand/reduce, it is not possible to embed each sparse expand/reduce in the current form of Algorithm 8. This is due to the dependencies of the two `ALL-REDUCE` operations in lines 6 and 8 to the sparse reduce in line 4. That is, the sparse reduce cannot be embedded into the `ALL-REDUCE` in line 6 because the $\mathbf{A}_k$ rows, which are computed in line 5, are required for the computation of λ. Furthermore, the sparse expand in line 7 cannot be embedded into the `ALL-REDUCE` in line 6 because distributed column normalization need to be performed before the expand. On the other hand, the sparse expand can be embedded into the `ALL-REDUCE` in line 8. Although embedding the sparse expand alone is important, it is insufficient for hiding latency since the sparse reduce, performed as P2P, will still be a bottleneck due to the high number of messages.

We propose to rearrange the computation and communication steps in Algorithm 8 to enable the embedding of all sparse expand/reduce operations without any dependency issues. We highlight two important observations that facilitate the rearrangements for successful embedding.

First observation: It is possible to expand non-normalized $\mathbf{A}_k$-matrix rows just after the operation in line 5, and then normalize $\hat{\mathbf{A}}_k$-matrix rows. In other words, instead of expanding normalized local-shared $\mathbf{A}_k$ rows, which requires the λ vector to be ready in advance, the non-normalized local-shared $\mathbf{A}_k$ rows are expanded while computing global λ using `ALL-REDUCE`. The extra cost here is that each processor will take the responsibility of normalizing nonlocal rows in addition to local and local-shared rows. With this observation the dependency between the `ALL-REDUCE` (line 6) and the sparse expand (line 7) can be removed, allowing the latter operation to be embedded into the former. The same argument applies to the normalization of **B**- and **C**-matrix columns in lines 12 and 18, respectively.

Second observation: The $\mathbf{A}^\top\mathbf{A}$ product (line 8 of Algorithm 8) is not required until the operation in line 11. The associated `ALL-REDUCE` neither has dependency on the sparse expand in line 7 nor on the sparse reduce in line 10, thus it can be used to embed the sparse reduce of *the next mode*. Similar discussion holds for the `ALL-REDUCE` associated with $\mathbf{B}^\top\mathbf{B}$. The $\mathbf{C}^\top\mathbf{C}$ product (line 20) is not required until the operation in line 5 *of the next iteration*. The associated `ALL-REDUCE` neither has dependency on the sparse expand in line 19 nor on the sparse reduce in line 4 of the next iteration, and therefore it can be placed anywhere between line 19 of the current iteration to before the operation in line 5 of the next iteration. This inter-mode and inter-iteration rearrangement is similar to the *software pipelining* used in compiler design and operating systems.

Algorithm 9 shows the rearranged version of Algorithm 8. Lines 7, 8, 10, 11 of Algorithm 9 realize the column normalization of matrix **A**, performed in line 6 of Algorithm 8, utilizing the first observation. In a similar way, lines 16, 17, 19, 20 and 25, 26, 28, 29 realize the column normalization of **B** and **C**, respectively. The `ALL-REDUCE` operation for computing $\mathbf{A}^\top\mathbf{A}$ is shifted forward to be a neighbor to the sparse reduce of the second mode (lines 13, 14). The same applies to $\mathbf{B}^\top\mathbf{B}$ and the sparse reduce of the third mode (lines 22, 23). On the other hand, $\mathbf{C}^\top\mathbf{C}$ is shifted to be a neighbor to the sparse reduce of the first mode in the next CPD-ALS iteration. The highlighted boxes show the sparse operations to be embedded in the preceding/following `ALL-REDUCE`. Since there are two sparse reduce and expand operations per tensor mode, the rearranged algorithm shows six boxes to indicate that all sparse operations are to be embedded for 3-mode tensors.

Algorithm 9 Rearranged Parallel CPD-ALS ($\mathcal{X}$) for 3-mode Tensors

```
 1: Randomly initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
 2: while not converged do
 3:     $\hat{\mathbf{A}}'_k \leftarrow \mathbf{X}_k^{(1)}(\hat{\mathbf{B}}_k \odot \hat{\mathbf{C}}_k)$          ▷ MTTKRP
 4:     Sparse REDUCE on shared $\mathbf{A}$-matrix rows
 5:     ALL-REDUCE to compute $\mathbf{C}^\top\mathbf{C}$
 6:     $\mathbf{A}_k \leftarrow \mathbf{A}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{B}^\top\mathbf{B})^{-1}$
 7:     $\lambda'_c \leftarrow \langle \mathbf{A}_k(:,c), \mathbf{A}_k(:,c) \rangle$, $\forall c \in [1..R]$
 8:     ALL-REDUCE to compute $\lambda'$
 9:     Sparse EXPAND on shared $\mathbf{A}$-matrix rows
10:     $\lambda_r \leftarrow \sqrt{\lambda'_c}$, $\forall c \in [1..R]$
11:     $\hat{\mathbf{A}}_k(:,c) \leftarrow \hat{\mathbf{A}}_k(:,c)/\lambda_c$, $\forall c \in [1..R]$
12:     $\hat{\mathbf{B}}'_k \leftarrow \mathbf{X}_k^{(2)}(\hat{\mathbf{A}}_k \odot \hat{\mathbf{C}}_k)$          ▷ MTTKRP
13:     Sparse REDUCE on shared $\mathbf{B}$-matrix rows
14:     ALL-REDUCE to compute $\mathbf{A}^\top\mathbf{A}$
15:     $\mathbf{B}_k \leftarrow \mathbf{B}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{A}^\top\mathbf{A})^{-1}$
16:     $\lambda'_c \leftarrow \langle \mathbf{B}_k(:,c), \hat{\mathbf{B}}_k(:,c) \rangle$, $\forall c \in [1..R]$
17:     ALL-REDUCE to compute $\lambda'$
18:     Sparse EXPAND on shared $\mathbf{B}$-matrix rows
19:     $\lambda_c \leftarrow \sqrt{\lambda'_c}$, $\forall c \in [1..R]$
20:     $\hat{\mathbf{B}}_k(:,c) \leftarrow \hat{\mathbf{B}}_k(:,c)/\lambda_c$, $\forall c \in [1..R]$
21:     $\hat{\mathbf{C}}'_k \leftarrow \mathbf{X}_k^{(3)}(\hat{\mathbf{A}}_k \odot \hat{\mathbf{B}}_k)$          ▷ MTTKRP
22:     Sparse REDUCE on shared $\mathbf{C}$-matrix rows
23:     ALL-REDUCE to compute $\mathbf{B}^\top\mathbf{B}$
24:     $\mathbf{C}_k \leftarrow \mathbf{C}'_k(\mathbf{B}^\top\mathbf{B} * \mathbf{A}^\top\mathbf{A})^{-1}$
25:     $\lambda'_c \leftarrow \langle \hat{\mathbf{C}}_k(:,c), \hat{\mathbf{C}}_k(:,c) \rangle$, $\forall c \in [1..R]$
26:     ALL-REDUCE to compute $\lambda'$
27:     Sparse EXPAND on shared $\mathbf{C}$-matrix rows
28:     $\lambda_c \leftarrow \sqrt{\lambda'_c}$, $\forall c \in [1..R]$
29:     $\hat{\mathbf{C}}_k(:,c) \leftarrow \hat{\mathbf{C}}_k(:,c)/\lambda_c$, $\forall c \in [1..R]$
    return $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$
```

4.4 Embedding Sparse Expand and Reduce

In order to realize the sparse expand and reduce operations using P2P messages, processor p_x should maintain two processor sets: workers set (WS) and masters set (MS) respectively defined as

$$\mathrm{WS}(p_x) = \bigcup_{i \ni r_i \text{ is local-shared}} S_i,$$
$$\mathrm{MS}(p_x) = \{owner(r_i) \mid r_i \text{ is nonlocal}\}.$$

That is, $\mathrm{WS}(p_x)$ contains the processors that contribute to the computation of any row that p_x owns, whereas $\mathrm{MS}(p_x)$ contains the processors that p_x is partially contributing to the computation of a row they own. Then, a sparse expand (reduce) on row r_i is achieved as messages from (to) p_x to (from) every processor in $\mathrm{WS}(p_x)(\mathrm{MS}(p_x))$.

4.4.1 Naive P2P Embedding

The hypercube-based `ALL-REDUCE` can be performed in $\log_2 K$ steps for a system with $K = 2^D$ processors. The K processors are virtually organized as a D-dimensional hypercube topology H. In H, each processor is represented by a D-bit binary number. We interchangeably use p_x to refer both index of a processor and its D-bit binary representation. Two processors are said to be neighbors along dimension i if their binary representation differ only in least significant bit i. In a D-dimensional hypercube, a d-dimensional subcube $(0 \leq d < D)$ is represented by d don't care bits (`X`) and $D-d$ fixed 0/1 bits thus having 2^d processors. Tearing along dimension i is defined as halving H into two disjoint $(D-1)$-dimensional subcubes such that the processors in the two sets are identified by the ith bit. For example, a tearing along dimension $i=1$ on processor set $P_{\texttt{XXXX}}$ organized as a 4-dimensional hypercube can be shown by processor sets $P_{\texttt{XX0X}}$ and $P_{\texttt{XX1X}}$. The hypercube-based `ALL-REDUCE` is well known and comes with several names such as E-cube routing, bidirectional exchange and exchange-add [72–74]. We adopt this `ALL-REDUCE` scheme and we use $\mathfrak{R}(H)$ to refer to it hereafter. A step s_i of

$\mathfrak{R}(H)$ represents the exchange of messages between neighboring processors along dimension i.

The naive embedding of P2P into `ALL-REDUCE` utilizing $\mathfrak{R}(H)$ is described as follows: A message $m(p_x, p_z)$ originating from p_x is sent from p_x to the neighbor at dimension i where i is the position of the least significant 1 bit in the XOR product $p_x \oplus p_z$. If the neighbor p_y at dimension i is the destination processor ($p_y = p_z$), then $m(p_x, p_z)$ is received and need not to be in any exchange in any upcoming step. Otherwise, p_y stores $m(p_x, p_z)$ in a forward buffer and sends it to its neighbor at dimension $j > i$, where j is the position of the least significant 1 bit in $p_y \oplus p_z$. A message is guaranteed to arrive to its destination in at most D steps.

4.4.2 Expand-and-Reduce-Aware Embedding

Consider expanding a local-shared factor matrix row r_i from p_0 to p_3 and p_5. In the naive EMB implementation, this expand consists of two different messages $m(p_0, p_3)$ and $m(p_0, p_5)$. Using $\mathfrak{R}(H)$, these messages will respectively take the routes $p_0 \rightarrow p_1 \rightarrow p_3$ and $p_0 \rightarrow p_1 \rightarrow p_5$. This means that r_i is sent (forwarded) twice in the message from p_0 to p_1. In general, a message between processor p_x and its neighbor p_y in any step can contain up to $D-1$ duplicates of the same row r_i. This is because the naive EMB described in Sec. 4.4.1 is unaware of the nature of the sparse expand and reduce. We can reduce the increase in the communication volume in EMB by exploiting the nature of the sparse expand and reduce operations via avoiding transmitting the same row more than once in a message between hypercube neighbors.

We propose an intelligent expand-and-reduce-aware EMB that avoids transmitting more than one copy of any row between hypercube neighbors as follows: During an embedded sparse expand, multiple copies of row r_i at step s of $\mathfrak{R}(H)$ are sent only once. During an embedded sparse reduce, multiple copies of row r_i at step s of $\mathfrak{R}(H)$ are reduced locally, and then sent as one copy. So, the reduce on r_i in the intelligent EMB is done during the routing steps of $\mathfrak{R}(H)$, whereas in naive EMB it is done at the receiving end by $owner(r_i)$ when all reduce messages are received.

4.4.3 Communication Duality in Embedding

In CPD-ALS, each shared factor-matrix row r_i is reduced from processors in S_i to $owner(r_i)$ and then the updated r_i (through local operations) is expanded from the same $owner(r_i)$ to the same set of processors S_i. That is, the same set of processors contribute to and need row r_i. We call such reduce and expand operations as dual communications.

In the P2P implementation, dual communications incur dual communication patterns. That is, if processor p_x sends r_i to p_y in the reduce communication, p_x will receive r_i from p_y in the expand communication. This means that the maximum expand send volume is equal to the maximum reduce receive volume. The same holds for maximum expand receive and maximum reduce send volumes.

We extend the duality definition of the P2P implementation to the EMB implementation as follows: The embeddings Γ_e and Γ_r of dual P2P expand/reduce are said to be dual if for each send message at step s_i of Γ_e, there exists a step s_j of Γ_r which involves a receive message with the same constituent rows, and vice versa. This duality ensures that the maximum send/receive volumes at step s_i of Γ_e are equal to the maximum receive/send volumes at step s_j of Γ_r, and both Γ_e and Γ_r incur the same amount of communication, including the forwarding overhead due to message routing.

According to the definition of duality in EMB, if both embeddings Γ_r and Γ_e utilize the $\mathfrak{R}(H)$ routing then they are not dual. Here we propose an EMB implementation that satisfies the duality definition and attains the nice properties of the dual reduce-and-expand communications. As the E-cube routing algorithm $\mathfrak{R}(H)$ defined earlier proceeds in increasing dimension order, we then define an inverse routing algorithm $\mathfrak{R}^{-1}(H)$ that proceeds in decreasing dimension order.

That is, in step s_i of $\mathfrak{R}(H)$ neighboring processors exchange messages along dimension i, whereas in step s_i of $\mathfrak{R}^{-1}(H)$ processors exchange messages along dimension $D-i-1$, for $i = 0, \ldots, D-1$. The following theorem shows duality in the proposed EMB implementation.

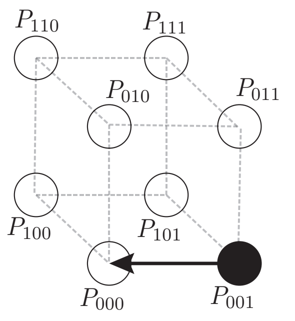
Theorem 1. *Utilizing $\mathfrak{R}(H)$ for embedding P2P sparse expand and $\mathfrak{R}^{-1}(H)$ for embedding a dual P2P sparse reduce (or vice versa) incurs dual embedded expand and reduce.*

Proof. In $\mathfrak{R}(H)$, each message $m(p_x, p_z)$ of the P2P expand of row r_i from $p_x = owner(r_i)$ to $p_z \in S_i$ routes through a certain path $\rho = p_x \rightarrow \cdots \rightarrow p_y \rightarrow \cdots \rightarrow p_z$. By the definition of $\mathfrak{R}(H)$ and $\mathfrak{R}^{-1}(H)$, a message $m(p_z, p_x)$ of the dual P2P reduce of row r_i follows the same path with reverse order $\rho^{-1} = p_z \rightarrow \cdots \rightarrow p_y \rightarrow \cdots \rightarrow p_x$. This means that for each expanded row in the message from p_y to its neighbor p_t in step s of $\mathfrak{R}(H)$, there is a dual reduced row in the message from p_t to p_y in step $D-s-1$ of $\mathfrak{R}^{-1}(H)$. Therefore, the constituent rows of the message from p_y to its neighbor p_t in step s of $\mathfrak{R}(H)$ are the same as those in the message from p_t to p_y in step $D-s-1$ of $\mathfrak{R}^{-1}(H)$. □

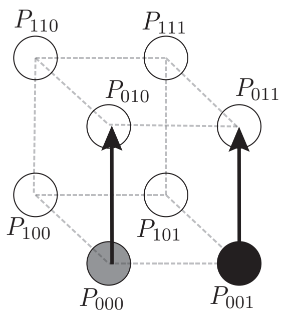
Duality in the EMB implementation, as well as in the P2P implementation, of expand and reduce is pivotal in reducing the problem size for intelligent partitioning models that encode decreasing communication cost metrics. Furthermore, the duality in EMB enables halving the storage overhead required for routing the data. That is, without the duality property there will be an explicit need for separate forward buffers during embedded expand and reduce operations.

4.5 Task-to-Processor Mapping

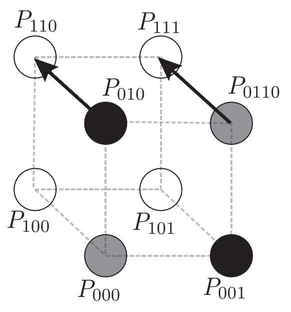
The objective of the proposed task partitioning and mapping is to minimize the communication volume overhead incurred by the embedding of the P2P communications into `ALL-REDUCE`. For this purpose, we define a communication cost metric which is set as the sum of the concurrent communication volume incurred



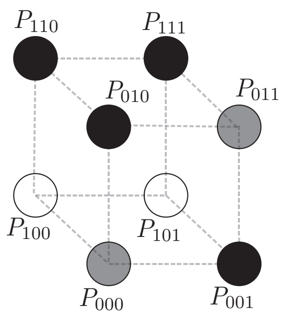
(a) Step 0: concurrent volume=1 total volume=1



(b) Step 1: concurrent volume=1 total volume=2



(c) Step 2: concurrent volume=1 total volume=2



(d) Expand done

Figure 4.1: A sample expand operation for a row r_i from the $owner(r_i) = p_1$ to $S_i = \{p_2, p_6, p_7\}$ in the embedded communication with E-cube routing.

by each shared factor-matrix row in EMB. In this concurrent communication cost metric, possibly multiple communications incurred by the same shared matrix row along the same dimension are counted as one. We preferred this communication cost metric in order to capture some form of volume concurrency involved in the expand and reduce operations associated with the shared factor-matrix rows during the `ALL-REDUCE` operations.

Figure 4.1 shows a sample expand incurred by a shared factor-matrix row r_i from $owner(r_i) = p_2$ to $S_i = \{p_2, p_6, p_7\}$ for E-cube routing on a 3-dimensional hypercube. The gray processors denote the intermediate processors which do not need r_i but involve in expanding r_i in EMB. In the figure, two communication operations along dimension two contributes only one to the concurrent communication volume. Then concurrent communication volume is three.

4.5.1 Hypergraph Model

Our proposed method utilizes a hypergraph model for nonzero-based partitioning. In this hypergraph model (will be denoted by $\mathcal{H}$), vertices represent atomic tasks, whereas nets represent factor-matrix rows. Here atomic tasks may refer to individual tensor nonzeros as well as disjoint nonzero clusters. The former case corresponds to the fine-grain [37, 45] tensor partitioning (see Section 3.2.2), whereas the latter case corresponds to the medium-grain [44] tensor partitioning. Each vertex is associated with a weight equal to the number of nonzeros it represents and each net is associated with a cost of R.

In $\mathcal{H}$, consider a net n_i^A representing factor matrix row $\mathbf{A}(i,:)$ along mode 1. Then, pins of this net represent the set of atomic tasks that contribute to the computation of $\mathbf{A}(i,:)$ during the MTTKRP operation along mode 1. During the MTTKRP operations along the two modes, the pins of this net represent the set of atomic tasks that need $\mathbf{A}(i,:)$ for their associated computations along that mode. Thus, n_i^A can be considered as encoding reduce type of communication along mode 1, whereas encoding expand type of communication along the two other modes. A similar discussion holds for nets n_j^B and n_k^C along modes 2 and 3, respectively.

A K-way partition on $\mathcal{H}$ with the objective of minimizing the "connectivity-1" metric given in (2.7) encodes minimizing the sum of the total communication volume along all MTTKRP operations. This does not encapsulate the communication of EMB but it does encapsulate the communication of P2P as follows: In a given partition $\Pi(\mathcal{H})$, if net n_i^A is internal to part $\mathcal{V}_k$ then row $\mathbf{A}(i,:)$ is local to part/processor $\mathcal{V}_k/p_k$ since all atomic tasks that contribute to and use that factor-matrix row are assigned to that part/processor. If net n_i^A is cut, then row $\mathbf{A}(i,:)$ becomes a shared row so that $\mathbf{A}(i,:)$ is local-shared for processor $owner(\mathbf{A}(i,:))$, whereas it is nonlocal for the processors in $S_i = \Lambda(n_i^A) \backslash owner(\mathbf{A}(i,:))$. If n_i^A is cut, its connectivity set $\Lambda(n_i^A) = \hat{S}_i$ denotes the set of processors that produce partial results for r_i^H during the MTTKRP operation along mode 1. $\hat{S}_i$ also denotes set of processors that need $\mathbf{A}(i,:)$ during MTTKRP operations along the two other modes. Thus, cut net n_i^A will incur reduce communication from the set of processors in S_i to the processor $owner(\mathbf{A}(i,:)$, whereas it will incur expand communication from the processor $owner(\mathbf{A}(i,:))$ to processors in S_i. A similar discussion holds for nets n_j^B and n_k^C along modes 2 and 3, respectively.

In the following subsection, we describe how $\mathcal{H}$ is utilized in an RB-based method that achieves many-to-one task mapping with the objective of reducing the concurrent communication volume incurred by the shared rows in EMB.

4.5.2 Recursive-Bipartitioning Scheme

In the proposed method, the RB levels are denoted as $d=0,\cdots,\log_2 K-1$, where $d=0$ denotes the root (bipartitioning of the original hypergraph) and $d=\log_2 K-1$ denotes the last internal level containing $K/2$ subhypergraphs. 2^d hypergraphs in the dth level are denoted by $\mathcal{H}_1^d,\ldots,\mathcal{H}_{2^d}^d$ from left to right for $0\leq d<\log_2 K$. Note that the RB tree is constructed utilizing the breadth-first bipartitioning order.

The RB steps are encoded as subtensor/subhypergraph-to-subcube mappings as follows: The root of the RB tree corresponds to hypergraph $\mathcal{H}_0^0$ representing the given tensor, which is initially mapped to whole hypercube $P_{\mathtt{X}\cdots\mathtt{X}}$. At level $d=0$, bipartitioning $\mathcal{H}_0^0$ into subhypergraphs $\mathcal{H}_0^1$ and $\mathcal{H}_1^1$ is encoded as mapping the subtensors represented by $\mathcal{H}_0^1$ and $\mathcal{H}_1^1$ respectively to the subcubes $P_{\mathtt{X}\cdots\mathtt{X0}}$ and $P_{\mathtt{X}\cdots\mathtt{X1}}$ of hypercube $P_{\mathtt{X}\cdots\mathtt{X}}$. At level $d=1$, bipartitioning $\mathcal{H}_0^1$ into $\mathcal{H}_0^2$ and $\mathcal{H}_1^2$ is encoded as mapping the subtensors represented by $\mathcal{H}_0^2$ and $\mathcal{H}_1^2$ respectively to the subcubes $P_{\mathtt{X}\cdots\mathtt{X00}}$ and $P_{\mathtt{X}\cdots\mathtt{X10}}$ of hypercube $P_{\mathtt{X}\cdots\mathtt{X0}}$; and bipartitioning $\mathcal{H}_1^1$ into $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$ is encoded as mapping the subtensors represented by $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$ respectively to the subcubes $P_{\mathtt{X}\cdots\mathtt{X01}}$ and $P_{\mathtt{X}\cdots\mathtt{X11}}$ of hypercube $P_{\mathtt{X}\cdots\mathtt{X1}}$. These two bipartitioning and mapping operations together corresponds to tearing hypercube along dimension $d=1$. That is, $P_{\mathtt{X}\cdots\mathtt{X00}}\cup P_{\mathtt{X}\cdots\mathtt{X01}}=P_{\mathtt{X}\cdots\mathtt{X0X}}$ and $P_{\mathtt{X}\cdots\mathtt{X10}}\cup P_{\mathtt{X}\cdots\mathtt{X11}}=P_{\mathtt{X}\cdots\mathtt{X1X}}$. This process is repeated at each level of the RB tree. Figure 4.2a shows simultaneous bipartitioning/mapping for a 3-dimensional hypercube. The RB-levels 0, 1 and 2 in the figure, respectively correspond to the tearing of the hypercube shown in Figure 4.1 along dimensions 0, 1 and 2.

In order to encode the objective of concurrent communication volume minimization mentioned earlier, we utilize a modified and enhanced version of the net splitting strategy, that is mentioned in Section 2.4 and used to encode the "connectivity-1" metric, in the above-mentioned recursive bipartitioning and mapping framework. The proposed enhancement is performed among the subnets of the same net within a same level, whereas conventional cut net splitting is continued to be applied across levels.

Consider the case where the subhypergraphs at a particular RB-level d contains multiple subnets (split nets) $n_i', n_i'', \cdots, n_i'^{\cdots\prime}$ of the same net n_i. Also consider the bipartitioning of the *first* level-d hypergraph $\mathcal{H}_x^d$ that contains the subnet n_i' of that net n_i. It is clear that there are three cases of net n_i' in the bipartition $\Pi_2(\mathcal{H}_x^d) = \{\mathcal{V}_0, \mathcal{V}_1\}$: n_i' is cut, n_i' is internal to left part $\mathcal{V}_0$ or right part $\mathcal{V}_1$.

1. n_i' is cut in $\Pi(\mathcal{H}_x^d)$: This means that shared-factor matrix row r_i is communicated along dimension d of the hypercube thus already encapsulating the concurrent communication volume metric along dimension d. Then we can safely remove its sibling nets $n_i'', \cdots, n_i'^{\cdots\prime}$ from the respective subhypergraph partitionings $\Pi(\mathcal{H}_{y>x}^d)$ to be performed later at this level. Although these sibling nets are not considered in the respective subhypergraph partitionings, the bipartitioning results of these subhypergraphs will be utilized to apply conventional cut net splitting on these sibling nets for including them into the subhypergraphs to be bipartitioned at the further RB levels $\ell > d$.

2. n_i' is internal to left part $\mathcal{V}_0$ in $\Pi(\mathcal{H}_x^d)$: This means that shared factor-matrix row r_i will incur concurrent communication volume only if at least one of its sibling nets $n_i'', \cdots, n_i'^{\cdots\prime}$ connect the right part $\mathcal{V}_1$ in a bipartition $\Pi(\mathcal{H}_{y>x}^d)$ to be obtained at the current level. This corresponds to the case where that sibling net is either cut or internal to right part $\mathcal{V}_1$ in that bipartition $\Pi(\mathcal{H}_{y>x}^d)$.

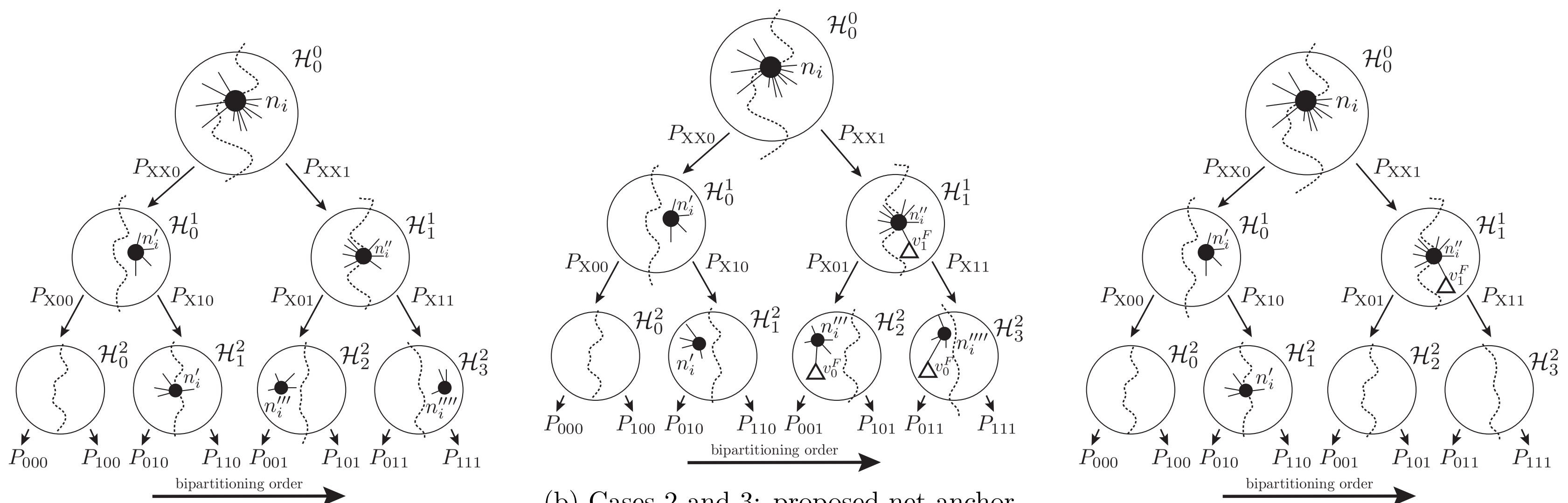
   Unfortunately current HP methods only adopt the cut net metric in two-way partitionings thus they cannot encode the increase in the cutsize for nets that are either cut or internal to a part. For this purpose, we introduce the net-anchoring scheme which is realized as follows: we introduce two vertices v_0^F and v_1^F which are fixed to left and right parts $\mathcal{V}_0$ and $\mathcal{V}_1$, respectively. Then a net is said to be anchored to the left part if it connects v_0^F, whereas it is said to be anchored to the right part if it connects v_1^F.

We utilize net-anchoring to encode the concurrent communication volume for such nets as follows: In each subhypergraph $\mathcal{H}_{y>x}^d$ that contains a sibling net n_i'' of n_i', we anchor n_i'' to the left part $\mathcal{V}_0$. In this way, we enforce n_i'' to connect left part in all bipartitions of those hypergraphs to be obtained at the current level. Thus, if n_i'' connects part $\mathcal{V}_1$ in any bipartition $\Pi_2(\mathcal{H}_{y>x}^d)$ then it will become cut and increasing the cutsize so that it will encode concurrent communication volume to be incurred correctly. After the first bipartition $\Pi_2(\mathcal{H}_{y>x}^d)$ in which n_i'' is cut at level d, all other further sibling nets $n_i''', \cdots, n_i'''^{\cdots\prime}$ will be removed from the respective subhypergraph $\mathcal{H}_{z>y}^d$ partitionings at level d in accordance with the Case 1.

3. n_i' is internal to right part $\mathcal{V}_1$ in $\Pi(\mathcal{H}_x^d)$: This case is handled in a dual manner with Case 2. That is, after the first bipartition $\Pi_2(\mathcal{H}_{y>x}^d)$ in which n_i' is internal to $\mathcal{V}_1$ at level d, in each subhypergraph $\mathcal{H}_{y>x}^d$ that contains a sibling net n_i'' of n_i', we anchor n_i'' to the right part $\mathcal{V}_1$.

Figure 4.2 illustrates the conventional cut net splitting technique (Figure 4.2a) as well as the proposed enhancements (Figure 4.2b and Figure 4.2c) for net n_i on 8-way partitioning with 3 RB levels. In all subfigures, at the root level bipartitioning, n_i is cut and thus split into its subnets n_i' and n_i''. In Figure 4.2a, at level-1, n_i' remains internal to $\mathcal{V}_1$ in $\Pi(\mathcal{H}_0^1)$, whereas it is cut in $\Pi(\mathcal{H}_1^1)$. At level-2, n_i' is cut in $\Pi(\mathcal{H}_1^2)$, whereas subnets n_i''' and n_i'''' of n_i'' remain internal to the left and right part in $\Pi(\mathcal{H}_2^2)$ and $\Pi(\mathcal{H}_2^2)$, respectively. Since n_i is cut three times, its $\lambda(n_i)-1$ value is three with the connectivity set $\Lambda(n_i) = \hat{S}_i = \{p_1, p_2, p_6, p_7\}$. Expanding this row is shown in Figure 4.1 for $owner(r_i) = p_1$.

Figure 4.2b shows Cases 2 and 3. At level-1 of the figure, since n_i' is internal to $\mathcal{V}_1$ in $\mathcal{H}_0^1$, n_i'' is anchored to the right part $\mathcal{V}_1$ in $\mathcal{H}_1^1$. Similarly, at level-2, n_i' is internal to $\mathcal{V}_0$ thus n_i''' and n_i'''' are anchored to the left part $\mathcal{V}_0$ in $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$, respectively. Figure 4.2c shows Case 1. At level-2 of the figure, n_i' is cut thus its sibling nets n_i''' and n_i'''', which are split from n_i'', are removed from $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$.



(a) Conventional cut net splitting

(b) Cases 2 and 3: proposed net anchoring

(c) Case 1: proposed net removal

Figure 4.2: (a) Conventional cut net splitting, (b) and (c) proposed enhancements for net n_i on eight-way partitioning with three levels of RB steps.

Algorithm 10 shows the steps of the proposed RB framework which realize the proposed enhancements. In the algorithm, $state(n)$ maintains if a net n becomes cut or internal to the left part (*L-internal*) or right part (*R-internal*) at the current level of the RB tree. $parent(n)$ denotes the parent net from which net n is obtained through splitting(s). That is, net n is effectively a subnet of $parent(n)$.

The outermost for loop in lines 4–32, performs the RB steps in breadth-first traversal order, whereas the inner for loop in lines 7–32 performs the bipartitionings at each level. The *state* information of the nets are initialized to NIL at the beginning of each level (lines 5–6). Lines 8 and 9 introduce the fixed vertices into $\mathcal{H}_k^d$ for enabling the realization of the net-anchoring. The inner for loop in line 10–16 applies proposed net-removal and net-anchoring techniques before bipartitioning the current hypergraph $\mathcal{H}_k^d$ according to current states of the subnets involved in $\mathcal{H}_k^d$. The inner for loop in lines 18-25 computes the *state* information for each net after bipartitioning. Lines 26–30 construct the left and right subhypergraphs $\mathcal{H}_{2k}^{d+1}$ and $\mathcal{H}_{2k+1}^{d+1}$ (to be bipartitioned at the next level $d+1$) from the current $\mathcal{H}_k^d$ using current bipartition $\Pi_2(\mathcal{H}_k^d)$ obtained in line 17 by utilizing the conventional cut net splitting. The for loop in lines 31–32 inherits the parent field of the cut nets to its split nets.

4.5.3 Factor-Matrix Row Assignment to Processors

The row-to-processor assignment problem corresponds to determining $owner(r_i)$ for each factor-matrix row r_i. For CPD utilizing P2P, the well known best-fit increasing heuristic used for solving the K-feasible bin-packing problem [75] is adopted [39, 44]. This method aims at balancing processors' volume loads without increasing the total communication volume. Here, we also adopt B-feasible bin-packing problem [75] for solving this assignment problem in EMB.

The main difference between the row-to-processor assignment problem encountered in P2P and EMB is that P2P involves a single communication step, whereas EMB involves loosely coupled $D = \log_2 K$ communication steps. So, in P2P, assignment of a row to a processor increases the volume load of only that processor,

Algorithm 10 RB-based task-to-processor assignment

```
Require: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $K$
 1: $\mathcal{H}_0^0 = \mathcal{H}$
 2: for each net $n \in \mathcal{N}_0^0$ do
 3:     $parent(n) = n$                    ▷ initialize parent of the net as itself
 4: for $d = 0$ to $\log_2 K - 1$ do
 5:     for each net $n \in \mathcal{N}_0^0$ do
 6:         $state(n) = NIL$                           ▷ initial value for each net
 7:     for $k = 0$ to $2^d - 1$ do
 8:         $\mathcal{V}_k^d = \mathcal{V}_k^d \cup \{v_0^F, v_1^F\}$
 9:         fix $v_0^F$ to $\mathcal{V}_0$, fix $v_1^F$ to $\mathcal{V}_1$
10:         for each net $n \in \mathcal{N}_k^d$ do
11:             if $state(parent(n))$ is $CUT$ then
12:                 $\mathcal{N}_k^d = \mathcal{N}_k^d \setminus \{n\}$
13:                          ▷ remove $n$ since it is already cut before at this level
14:             if $state(parent(n))$ is $L\text{-}internal$ then
15:                 $Pins(n) = Pins(n) \cup \{v_0^F\}$              ▷ anchor $n$ to left part
16:             if $state(parent(n))$ is $R\text{-}internal$ then
17:                 $Pins(n) = Pins(n) \cup \{v_1^F\}$             ▷ anchor $n$ to right part
18:         $\Pi_2 = \text{BIPARTITION}(\mathcal{H}_k^d = (\mathcal{V}_k^d, \mathcal{N}_k^d))$          ▷ $\Pi_2 = \{\mathcal{V}_0, \mathcal{V}_1\}$
19:         for each net $n \in \mathcal{N}_k^d$ do
20:             if $n$ is a cut net then
21:                 $state(parent(n)) = CUT$
22:             if $state(parent(n))$ is not $CUT$ then
23:                 if $n$ is internal to left part then
24:                     $state(parent(n)) = L\text{-}internal$
25:                 if $n$ is internal to right part then
26:                     $state(parent(n)) = R\text{-}internal$
27:         Form $\mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ induced by $\mathcal{V}_0$
28:         $\mathcal{N}_0 = \{n' : n \in \mathcal{N}, pins(n) \cap \mathcal{V}_0 \neq 0 \; \exists \; pins(n') = pins(n) \cap \mathcal{V}_0\}$          ▷
    conventional cut net splitting
29:         Form $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$ induced by $\mathcal{V}_1$
30:         $\mathcal{N}_1 = \{n'' : n \in \mathcal{N}, pins(n) \cap \mathcal{V}_1 \neq 0 \; \exists \; pins(n'') = pins(n) \cap \mathcal{V}_1\}$          ▷
    conventional cut net splitting
31:         $\mathcal{H}_{2k}^{d+1} = \mathcal{H}_0$, $\mathcal{H}_{2k+1}^{d+1} = \mathcal{H}_1$
32:         for each cut net $n \in \mathcal{N}_k^d$ split as $n'$ and $n''$ do
33:             $parent(n') = parent(n'') = parent(n)$
```

whereas in EMB it increases the volume loads of at most D processors in different communication steps. That is, if the distance between the owner and receiver processors is equal to the dimension D of the hypercube, there are $D-2$ intermediate processors which are only forwarding the factor-matrix row. So, each processor has a volume load at D different communication steps. This difference increases the number of bins from K in P2P to DK in EMB.

In EMB, the cost of a row-to-processor assignment instance is defined as the sum of the volume load of the maximally loaded processor in each dimension. So, for the best-fit criterion we define the sum of squares function as

$$\sum_{d=1}^{d=D}(\sum_{k=1}^{k=K} B_{dk}^2)^2. \tag{4.1}$$

In the proposed algorithm, for each mode, factor-matrix rows are considered in decreasing order of their $|\hat{S}_i|$ values for assignment. The best-fit criterion for the assignment is to select the processor that incurs the minimum increase in (4.1). After each assignment, we increase the loads of the bins which are involved in the communication in terms of both send and receive volumes. In this way, (4.1) captures processors' send plus receive volume loads during expand communication which is equal to the sum of processors' send volume loads during expand and reduce communications thanks to the duality described in Sec. 4.4.3.

For P2P, each row is assigned to one of the processors which contributes/needs that factor matrix row. This ensures total communication volume does not increase with the assignment. On the other hand, for EMB, we can relax this constraint. That is, consider the processors that participate in the communication of a shared row but do not possibly contribute/need that row. Such processors can also be considered as candidate owners. Since these processors are already communicating that row such assignments might not increase the volume load. Obviously this relaxation is expected to further decrease the function in (4.1) because of larger degree of freedom for each assignment. We should mention here that this relaxation in row-to-processor assignments does not affect the concurrent communication cost metric defined for individual shared factor-matrix rows and minimized by the scheme in Sec. 4.5.2.

4.6 Experiments

4.6.1 Setting

We performed experiments using three methods: P2P-mg, EMB-rand and EMB-hp. The term left to the hyphen denotes the parallel scheme used (P2P or EMB), whereas the right term denotes the nonzero partitioning method used. The mg in P2P-mg refers to partitioning the input tensor according to the state-of-the-art medium-grain HP model [44]. The rand in EMB-rand refers to partitioning the input tensor randomly in such a way that numbers of nonzeros assigned to processors differ by at most one. The hp in EMB-hp refers to partitioning and mapping the tensor nonzeros by using the method proposed in Sec. 4.5. For partitioning the hypergraph models in P2P-mg and EMB-hp, we use the tool PaToH [52, 76] with default parameters.

Parallel setup: The experiments are taken with up to 4096 processors on an Apollo 9000 HPC system. Each node in this system consists of two AMD EPYC 7742 processors, each with 64 cores, and 256 GB of memory. The nodes are connected with a Mellanox HDR Infiniband network. We use 16 cores per node in all our experiments.

Dataset: Our dataset consists of thirteen real-world sparse tensors with varying sizes. Table 4.1 shows the tensors and their properties. `Delicious`, `Enron`, `Flickr` and `NELL-1` are obtained from the FROSTT sparse tensors repository [77]. `1998DARPA` contains tuples that represent timestamps of connections made between source IP and destination IP. `Freebase-music` contains music-related (subject entity, object entity, relation) tuples from Freebase online database.`Gowalla` contains check-in data as (user, POI, check-in) tuples from the location-based social network Gowalla [78]. `Movies-amazon` contains user-movie-word tuples from the user reviews of movies in Amazon [79]. `Netflix` and `Yelp` are rating datasets that respectively contain (usr, business, rating) and (user, movie, rating) tuples.

The dataset also contains the largest three tensors from FROSTT, `Amazon-reviews`, `Patents` and `Reddit-2015`, each having more than 1B nonzeros. Since common HP tools such as PaToH and hMeTis [80] do not support 64-bit integers, these very large tensors are only used to evaluate the EMB framework (Sections 4.3 & 4.4) by comparing EMB-rand versus P2P-rand, whereas the rest of the tensors are used to evaluate all contributions.

Table 4.1: Properties of the Test Tensors

tensor	mode sizes				nnz	density
	I_1	I_2	I_3	I_4		
`1998DARPA`	23.8M	22.5K	22.5K		28.4M	2.37E-09
`Delicious`	533K	17.3M	2.47M	1.44K	140M	4.27E-15
`Enron`	6.07K	5.70K	244K	1.18K	54.2M	5.46E-09
`Flickr`	320K	28.2M	1.61M	731	113M	1.07E-14
`Freebase-music`	23.3M	0.17K	23.3M		100M	1.10E-09
`Gowalla`	1.3M	0.60K	107K		6.26M	7.65E-08
`Movies-amazon`	227K	4.40K	87.8K		15.0M	1.72E-07
`NELL-1`	25.5M	2.14M	2.90M		144M	9.05E-13
`Netflix`	480K	2.18K	17.8K		100M	5.40E-06
`Yelp`	773K	85.5K	687K		186M	4.09E-09
Very Large Tensors						
`Amazon-reviews`	4.82M	1.77M	1.80M		1.74B	1.13E-10
`Patents`	239K	46	239K		3.60B	1.37E-03
`Reddit-2015`	8.21M	176K	8.12M		4.69B	3.97E-10

4.6.2 Performance Results

Latency hiding: Table 4.2 displays the amount of latency hidden by EMB in terms of number of messages whose latency overheads are totally avoided. In the table, P2P columns show the number of messages only for the sparse expand and reduce operations during a CPD-ALS iteration. That is, latency overhead of `ALL-REDUCE` is not included in the P2P columns. The table also displays the latency overhead of `ALL-REDUCE` during a CPD-ALS iteration which is the only latency overhead in EMB. In the table, “max” and “avg” respectively refer to the maximum and average number of messages handled by processors during a CPD-ALS iteration. The maximum and average number of messages under the P2P columns are the sums of maximum and average number of messages required to perform the sparse expand and reduce operations in P2P for each tensor mode. The number of messages under the EMB column are the sum of messages during the two `ALL-REDUCE` operations for each tensor mode. Note that maximum and average values in EMB are equal due to the regularity of communication.

Table 4.2: Max/Avg number of messages in a CPD-ALS iteration on $K = 4096$

tensor	P2P		EMB
	max	avg	max(=avg)
`1998DARPA`	6,578	145	72
`Delicious`	25,413	13,332	96
`Enron`	19,755	2,149	96
`Flickr`	15,382	4,709	96
`Freebase-music`	14,739	868	72
`Gowalla`	6,245	907	72
`Movies-amazon`	9,590	1,786	72
`NELL-1`	22,543	15,434	72
`Netflix`	14,824	3,391	72
`Yelp`	20,375	6,112	72
average	**14,076**	**2,480**	**78**

Table 4.2 shows that sparse expand/reduce incur significantly large number of messages in P2P, thus rendering the parallel CPD-ALS as latency bound with increasing K. This is because the number of messages in P2P usually increases linearly with increasing K. On the other hand, the number of messages in EMB is significantly smaller and increase logarithmically with increasing K. The 72 and 96 values under EMB refer to the number of messages handled by a processor in 3-mode ($3 \times 2 \times \log_2 K$) and 4-mode ($4 \times 2 \times \log_2 K$) tensors, respectively.

As seen in Table 4.2, there is a significant imbalance between maximum and average number of messages in P2P. This disturbs the scaling performance since

Table 4.3: Improvement of Expand/Reduce-aware EMB against Naive EMB

method	K = 128	256	512	1024	2048	4096
EMB-rand	0.78	0.79	0.78	0.76	0.72	0.73
EMB-hp	0.90	0.86	0.84	0.81	0.82	0.76

Values are EMB runtimes normalized w.r.t. those by naive EMB.

usually the maximum metric defines the runtime since there are global synchronizations (due to `ALL-REDUCE`) before/after the sparse P2P communication steps. On the other hand, this problem does not arise in EMB since the regular communication pattern of EMB naturally attains equal number of maximum and average messages. This is a clear advantage in favor of EMB since there is no need to consider reducing/balancing the number of messages when designing intelligent partitioning models allowing them to focus on reducing/balancing volume.

The expand-and-reduce-aware embedding: Table 4.3 shows the benefit of using expand-and-reduce-aware EMB (Sec. 4.4.2) over naive EMB (Sec. 4.4.1) on both EMB-rand and EMB-hp. The values given in the table are CPD-ALS iteration times of the ten tensors taken with expand-and-reduce-aware EMB normalized with respect to those taken with naive EMB. The runtimes of all tensors are then averaged per K value for each method. As seen in the table, utilizing expand-and-reduce-aware EMB for sparse expand and reduce decreases the parallel CPD-ALS runtime, on average, by up to 21% − 28% when EMB-rand is used and by 10% − 24% when EMB-hp is used. Furthermore, the relative percent improvement of expand-and-reduce-aware EMB over naive EMB for both EMB-rand and EMB-hp generally increases with increasing K.

HP-based mapping: Table 4.4 shows the performance improvement attained by the HP-based mapping algorithm discussed in Sec. 4.5 against EMB-rand on $K = 4096$. The performance comparison is given in terms of maximum and concurrent volume metrics as well as parallel runtimes for $R = \{8, 32\}$. For each tensor, the first line displays actual values for EMB-hp, whereas the second line displays normalized values with respect to those of EMB-rand.

Table 4.4: Performance of EMB-hp against EMB-rand on $K = 4096$

tensor	volume $\times R$		runtime (ms)	
	max	concurrent	$R = 8$	$R = 32$
`1998DARPA`	9,155	94,639	28.609	35.794
	0.359	0.004	0.929	0.702
`Delicious`	178,695	15,014,082	79.236	201.793
	0.477	0.124	0.512	0.311
`Enron`	23,964	443,498	4.984	16.829
	0.525	0.145	0.321	0.219
`Flickr`	51,391	5,472,870	12.802	77.683
	0.139	0.025	0.091	0.142
`Freebase-music`	19,421	5,440,976	64.492	424.432
	0.053	0.016	0.444	0.507
`Gowalla`	7,152	1,056,561	1.966	7.306
	0.314	0.126	0.254	0.219
`Movies-amazon`	22,730	1,152,274	4.473	15.737
	0.766	0.504	0.526	0.343
`NELL-1`	230,823	27,511,288	50.343	262.416
	0.651	0.176	0.369	0.479
`Netflix`	60,170	3,092,374	11.275	45.305
	0.440	0.519	0.151	0.137
`Yelp`	180,303	6,374,191	35.990	172.814
	0.591	0.525	0.261	0.265
average	**41,708**	**2,566,274**	**16.688**	**62.751**
	0.349	**0.098**	**0.322**	**0.292**

For each tensor, the first line displays actual values for EMB-hp, whereas the second line displays the normalized values w.r.t. those of EMB-rand.

As seen in Table 4.4, EMB-hp achieves significant decrease in concurrent communication volume metric (90% on average) compared to EMB-rand. EMB-hp achieves also significant decrease in maximum communication volume handled by a processor (65% on average) compared to EMB-rand. These improvements in concurrent and maximum communication volume metrics lead to an approximately 68% and 71% improvement in CPD-ALS iteration time respectively for $R = 8$ and 32. Note that improvement in the maximum communication volume closely correlates with the improvement in the parallel runtime on average.

Factor-matrix row assignment: Table 4.5 shows the performance improvement of the proposed bin-backing based factor-matrix row assignment method (Sec. 4.5.3) against random assignment on $K = \{128, \cdots, 4096\}$. The performance comparison is given in terms of the maximum communication volume handled by processors obtained by bin-packing-based-assignment algorithm normalized with respect to those by random assignment. As seen in the table, the bin-backing algorithm attains considerable performance improvement (15% on average) against random assignment.

Table 4.5: Performance of Proposed Row-to-Processor Assignment

tensor	maximum volume					
	K = 128	256	512	1024	2048	4096
`1998DARPA`	1.07	1.08	1.04	0.90	1.00	0.99
`Delicious`	0.82	0.86	0.84	0.85	0.86	0.86
`Enron`	0.91	0.93	0.90	0.86	0.89	0.86
`Flickr`	0.87	0.87	0.89	0.86	0.86	0.84
`Freebase-music`	0.69	0.63	0.70	0.63	0.60	0.56
`Gowalla`	0.81	0.79	0.83	0.82	0.84	0.81
`Movies-amazon`	0.85	0.88	0.84	0.81	0.87	0.83
`NELL-1`	0.80	0.84	0.83	0.87	0.88	0.88
`Netflix`	0.82	0.82	0.85	0.88	0.91	0.93
`Yelp`	0.85	0.84	0.84	0.84	0.89	0.84
average	**0.84**	**0.85**	**0.85**	**0.83**	**0.85**	**0.83**

Values are normalized w.r.t. those of random assignment.

Strong scaling: Figures 4.3 and 4.4 show2 the strong scaling curves of the three methods on $K = \{128, \cdots, 4096\}$ processors with two different R values. As seen in Figures 4.3 and 4.4, P2P-mg does not scale after $K = 1024$ for the tensors `Delicious`, `Flickr` and `NELL-1`, whereas it does not scale after $K = 512$ for rest of the tensors. Both EMB schemes scale much better than P2P for each tensor and for both R values.

As seen in Figures 4.3 and 4.4, EMB-hp runs much faster than EMB-rand in all instances thus showing the validity of the task-to-processor mapping method proposed in Sec. 4.5. Furthermore, EMB-hp runs much faster than the state-of-the-art P2P-mg for all tensors and all R values on $K > 1024$.

Figure 4.5 shows the strong scaling curves for the three very large tensors. As seen in the figure, for each large tensor, P2P-rand fails to scale after 1024 processors, whereas EMB-rand continues to scale up to 4096 processors.

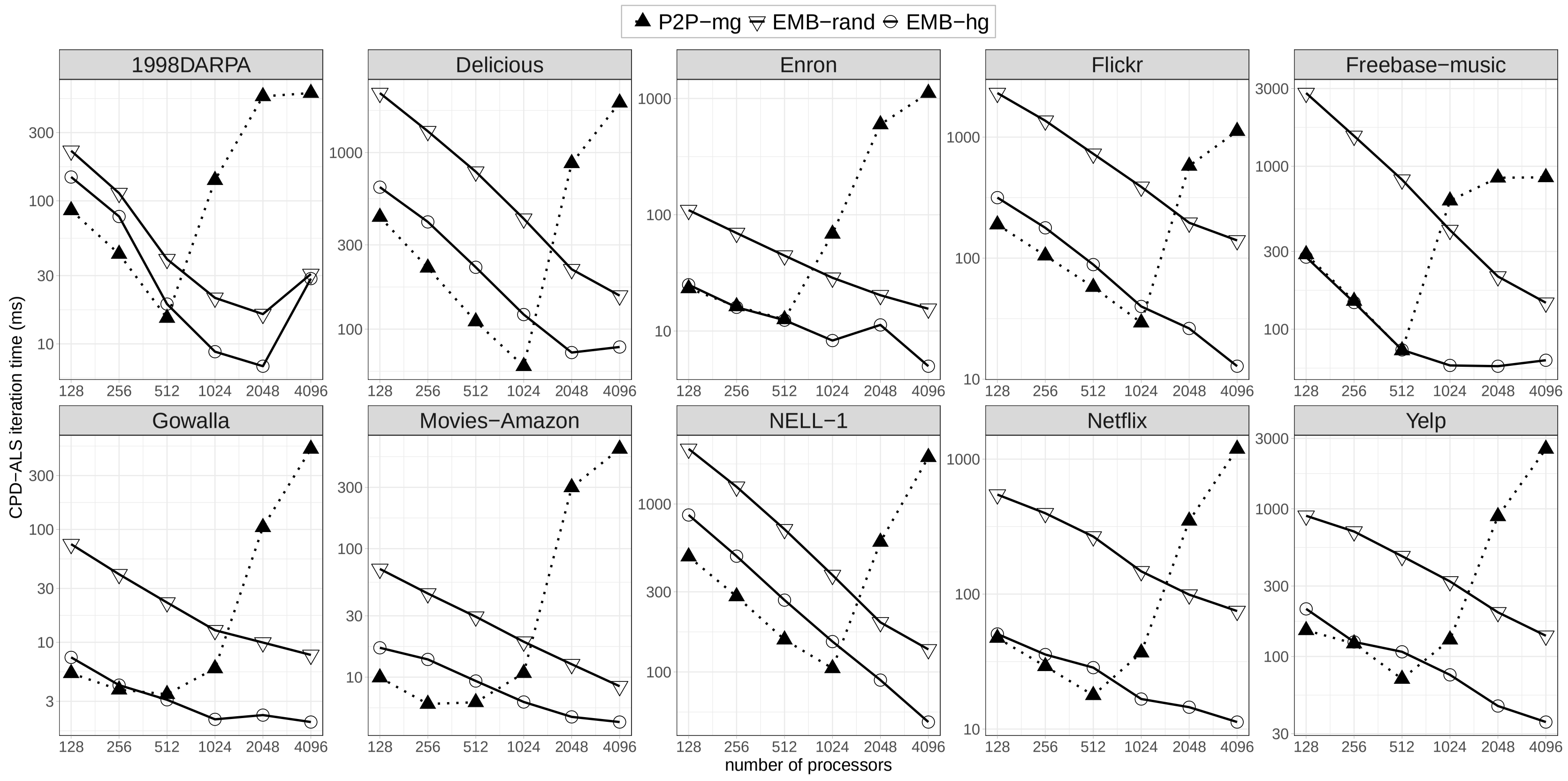


Figure 4.3: Comparing Strong Scaling curves of P2P-mg, EMB-rand and EMB-hp with decomposition rank $R = 8$.

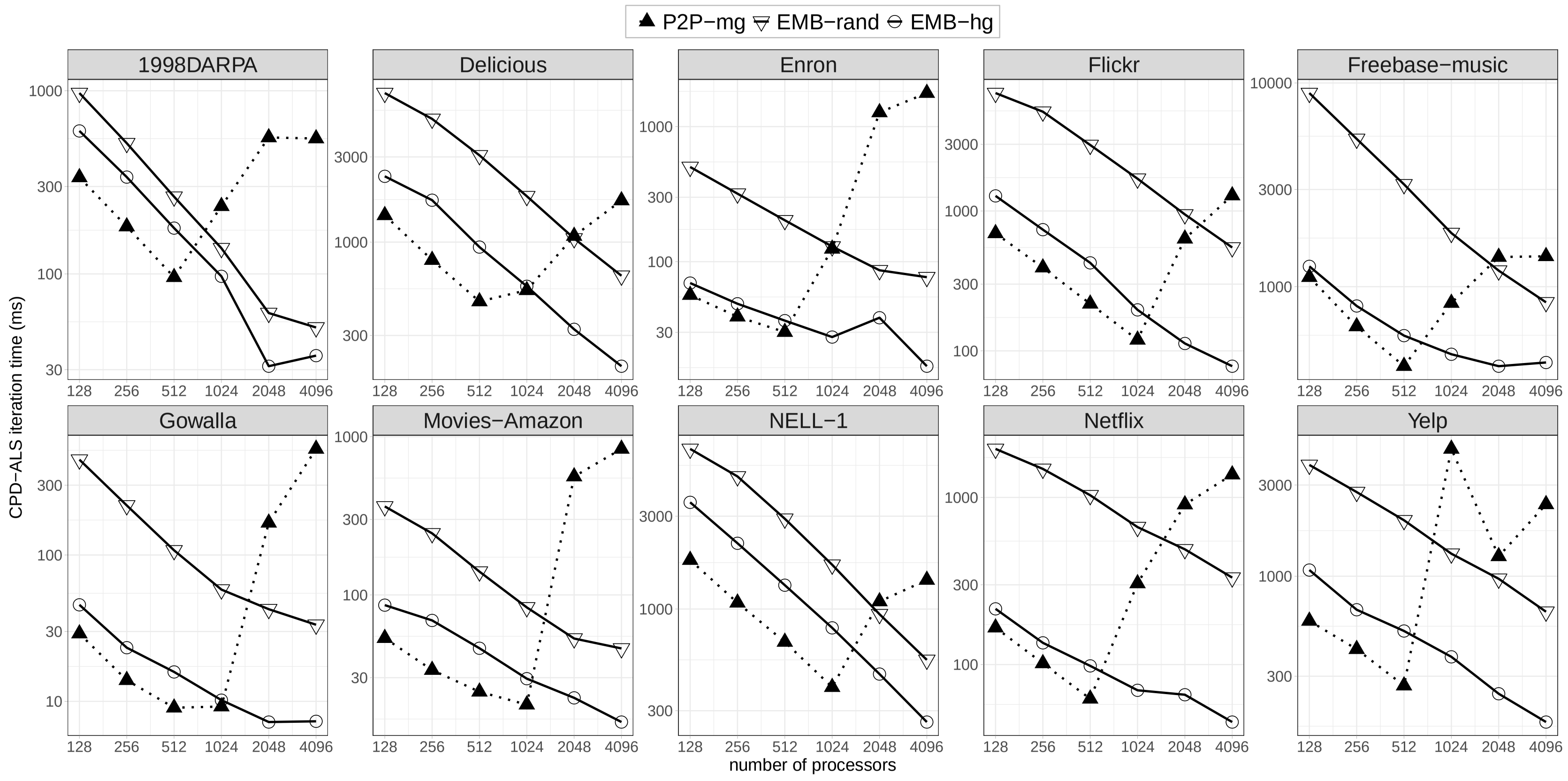
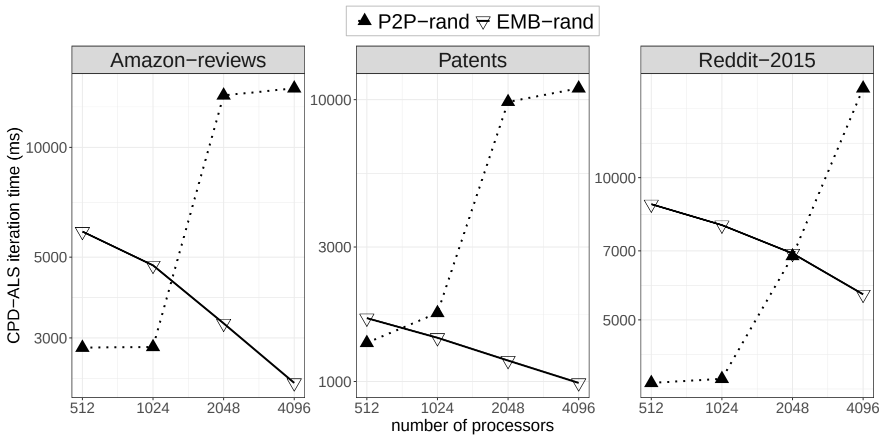
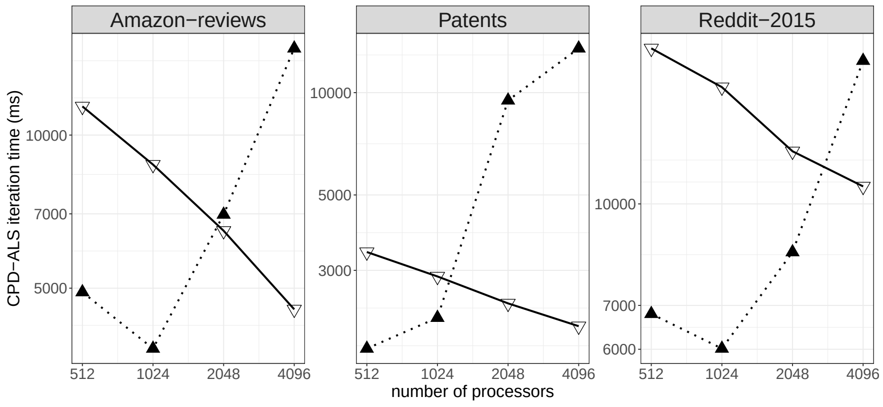


Figure 4.4: Comparing Strong Scaling curves of P2P-mg, EMB-rand and EMB-hp with decomposition rank $R = 32$.



(a) $R = 8$



(b) $R = 32$

Figure 4.5: Strong Scaling curves of EMB-rand versus P2P-rand on very large tensors with decomposition ranks $R = 8$ and $R = 32$.

4.7 Related Work

In the literature, there exists many shared- and distributed-memory parallel CPD-ALS algorithms [34, 36–45, 81–83]. Here we briefly mention about distributed-memory parallel CPD-ALS algorithms.

Several works on scaling distributed-memory parallel CPD-ALS target at enhancing the MTTKRP operation and/or reducing the bandwidth overhead of the P2P sparse reduce and expand operations through intelligent combinatorial models or through multidimensional division methods. For instance, among combinatorial models for enhancing MTTKRP, [40, 43] and [45] are proposed. Among combinatorial models for reducing communication overhead, HP is utilized [37, 39, 44, 45]. However, these HP models focus on reducing the bandwidth component of the communication. Multidimensional Cartesian partitioning is utilized with a nice property of bringing upper bounds on both bandwidth and latency components costs [38, 39]. The HP model in [39] targets at reducing the bandwidth requirement of Cartesian partitioning. [41] also considers partitioning factor matrices column-wise at the expense of tensor replication, whereas all other methods as well as our method involve row-wise partitioning of the factor matrices. There also exists toolkits for shared- and distributed-memory parallel systems [32, 36, 38, 42, 83].

Latency reduction and hiding is well-known in parallel iterative solvers, such as Conjugate Gradient and GMRES, through communication/computation overlapping [84, 85], pipelining [86], and embedding [87]. The embedding scheme proposed in [87] exploits the fact that each SpMV is followed by an inner product which involves the input and output vectors. They propose to embed sparse expand operations on the output vector entries to the following inner product realized with `ALL-REDUCE` by utilizing row-parallel SpMV. Our work differs from [87] in the following aspects: The rearrangements which enable the embedding are different because of the nature of the applications (CG vs. CPD-ALS); [87] embeds only sparse expand whereas we embed both sparse expand and reduce; [87] uses naive embedding so that each message in the `ALL-REDUCE` may contain multiple copies of same output-vector entries, whereas we avoid this with the proposed expand-and-reduce-aware embedding; [87] uses conventional HP followed by a KL-based one-to-one mapping, whereas we propose a simultaneous partitioning/mapping algorithm. To our knowledge, our work is the first to use latency hiding in parallel tensor decomposition.

4.8 Conclusion

We proposed a framework for hiding the latency of P2P sparse expand and reduce operations during parallel CPD-ALS through embedding them into dense collective `ALL-REDUCE` operations which already exist in the CPD-ALS. The framework consists of a computation/communication rearrangement of the CPD-ALS which enables the embedding as well as an intelligent embedding scheme that helps reducing the increase in communication due to embedding. The recursive-bipartitioning-based hypergraph partitioning method proposed for subtensor-to-processor mapping as well as the bin-backing-based method proposed for factor-matrix row to processor mapping are found to be quite effective in reducing the bandwidth overhead in the embedded-`ALL-REDUCE`. We have obtained very good scaling results on up to 4096 processors for ten real-word tensors, whereas a state-of-the-art P2P implementation does not scale after 1024 processors due to large the latency overhead especially for small decomposition ranks. The proposed latency-hiding framework paves the way for scalable sparse tensor decomposition on exa-scale systems.

Chapter 5

Communication-Efficient Stratified Stochastic Gradient Descent for Distributed Matrix Completion

5.1 Overview

This chapter presents a framework for communication-efficient distributed SSGD. The framework enables SSGD-based matrix factorization to run on very large-scale parallel systems through significantly reducing the bandwidth overhead and controlling the latency overhead through putting upper limits on the number of exchanged messages.

The rest of the chapter is organized as follows: Section 5.2 gives the essentials of using and parallelizing SGD for matrix completion. In Section 5.3, the communication requirement in parallel SSGD is studied in detail. In Section 5.4, the proposed framework for scaling P2P SSGD, including the hold-and-combine scheme, is presented. In Section 5.5, the proposed hypergraph partitioning (HP)

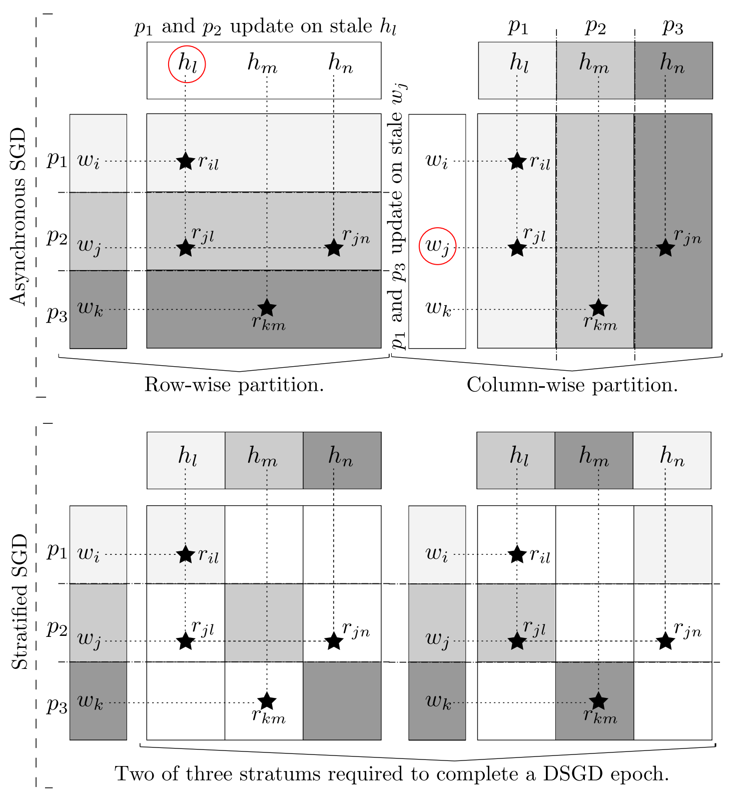


Figure 5.1: Stale updates in simple row- or column-wise partitions (up) versus stale-free DSGD (bottom). In the row-wise partition of $\mathbf{R}$, the rows of $\mathbf{W}$ are partitioned conformably and thus each $\mathbf{W}$-matrix row is accessed by one processor. However, this is not the case for $\mathbf{H}$-matrix rows. For instance, ratings r_{il} and r_{jl} are respectively distributed to p_1 and p_2 and both used to update $\mathbf{h}_l$ possibly at the same time thus either p_1 or p_2 will update on a stale $\mathbf{h}_l$. Similar discussion holds for column-wise partition in a dual manner regarding r_{jl}, r_{jn} and $\mathbf{w}_j$.

method is presented. Section 5.6 contains the experiments conducted on an HPC system along with the results and discussions. Related works are discussed in Section 5.7 and the chapter is concluded in Section 5.8.

5.2 Preliminaries: Parallel SGD

SGD is sequential in nature, thus parallelizing it requires communicating up-to-date $\mathbf{W}$- and $\mathbf{H}$-matrix rows. There are two main approaches to parallel SGD: asynchronous and stratified. *Asynchronous* methods allow the updates given in (2.3) and (2.4) to use and to be performed on stale versions of $\mathbf{w}_i$ and/or $\mathbf{h}_j$. Asynchronous methods are usually non-serializable. Simple parallelizations of the SGD-based matrix completion, such as row-wise or column-wise partitioning of the rating matrix, are examples of asynchronous SGD (see Figure 5.1).

In order to mitigate the staleness problem, Gemulla et al. [27] proposed the *stratified* SGD (SSGD). In SSGD, the rating matrix is divided into K^2 2D blocks using K-way mutually exclusive and exhaustive partitions on the rows $\Pi^R = \{R_1, \ldots, R_K\}$ and columns $\Pi^C = \{C_1, \ldots, C_K\}$ of $\mathbf{R}$. The rows of dense matrices $\mathbf{W}$ and $\mathbf{H}$ are partitioned conformably with Π^R and Π^C, respectively. We denote the row blocks of $\mathbf{W}$ and $\mathbf{H}$ that respectively conform with R_α and C_β as $\mathbf{W}_\alpha$ and $\mathbf{H}_\beta$. We denote a block of $\mathbf{R}$ with rows in R_α and columns in C_β as $\mathbf{R}_{\alpha\beta}$.

A set of K 2D non-overlapping sub-matrix blocks are called a *stratum* (denoted by $\mathcal{S}$ hereafter). Two 2D sub-matrix blocks are said to be non-overlapping if they do not share any row or column. A set of K stratums $S = \{\mathcal{S}_1, \ldots, \mathcal{S}_K\}$ that exhausts all of the K^2 sub-matrix blocks is called correct strata. Figure 5.2 shows the strata $S = \langle \mathcal{S}_1 = \{\mathbf{R}_{1,1}, \mathbf{R}_{2,2}, \ldots, \mathbf{R}_{8,8}\}, \mathcal{S}_2 = \{\mathbf{R}_{1,2}, \mathbf{R}_{2,3}, \ldots, \mathbf{R}_{8,1}\}, \ldots, \mathcal{S}_8 = \{\mathbf{R}_{1,8}, \mathbf{R}_{2,1}, \ldots, \mathbf{R}_{8,7}\}\rangle$. The distinguishing property of SSGD is that no ratings in different blocks of a stratum can update the same row of the factor matrices $\mathbf{W}$ and $\mathbf{H}$. Therefore, if the ratings constituting a stratum $\mathcal{S}_i$ are used to update the relevant $\mathbf{H}$-matrix rows in a mini SGD epoch (called sub-epoch) then each of the K 2D sub-matrix blocks can be handled by a separate processor; thus eliminating the staleness problem.

There are several ways to generate a correct strata that covers the whole dataset and schedule the strata to sub-epochs. For simplicity, we consider a simple form of scheduling as follows: at sub-epoch 1, processor p_x, for $x = 1, 2, \ldots, K$, processes the ratings in $\mathbf{R}_{xx}$ to update the rows in $\mathbf{W}_x$ and $\mathbf{H}_x$; at sub-epoch k, processor p_x processes the ratings in $\mathbf{R}_{x\beta}$ to update the rows in $\mathbf{W}_x$ and $\mathbf{H}_\beta$, where $\beta = (1 + (x + k - 2) \mod K)$. We refer to this scheduling as "ring scheduling" or "ring strata" hereafter. A general form of the ring scheduling consists of a *seed*, where $1 \leq seed \leq K$. At sub-epoch k, the processor p_{seed} processes the ratings in $\mathbf{R}_{seed,k}$ to update the rows in $\mathbf{W}_{seed}$ and $\mathbf{H}_k$. At sub-epoch k, processor p_x

processes the ratings in $\mathbf{R}_{x\beta}$ to update the rows in $\mathbf{W}_x$ and $\mathbf{H}_\beta$, where

$$\beta = 1 + (k + (seed + x - 1 \mod K) - 2 \mod K). \tag{5.1}$$

In [27], the parallel algorithm that utilizes SSGD is converted to a Distributed Stochastic Gradient Descent (DSGD) algorithm for large-scale matrix completion. In DSGD, each stratum is executed in parallel in one sub-epoch, where the **W**- and **H**-matrix rows are updated with the ratings in the stratum according to (2.3) and (2.4). Then, inter-processor communications are performed to synchronize all updated rows of factor matrices. If a row-parallel execution is chosen, that is the **R** matrix is partitioned row-wise such that each row block is executed by a single processor, then communication is restricted to the **H**-matrix rows. Row-parallel execution is usually preferred because the number of items is generally much less than the number of users which means the amount of data to be communicated (**H**) is small compared to **W**. In row-parallel execution, we abuse the stratum notation $\mathcal{S}$ to also be viewed as a mapping function $\mathcal{S} : [K] \rightarrow [K]$ (where $[K]$ is used to denote the set $\{1, \dots, K\}$ hereafter) from a processor p_k to the index β of a column block C_β. For instance, $\mathcal{S}_2(p_4) = 5$ means that during sub-epoch 2, processor p_4 will exclusively update the rows of the $\mathbf{H}_5$ sub-matrix. We also use $\mathcal{S}_\beta^{-1}(p_x)$ to retrieve the sub-epoch at which p_x updates $\mathbf{H}_\beta$.

5.3 Communication in Distributed SSGD

Given strata S where each stratum is to be processed in a sub-epoch in row-parallel execution. For an **H**-matrix row block $\mathbf{H}_\beta$, we define the sequence

$$\Upsilon_\beta = \langle p_{i_1}, p_{i_2}, \dots, p_{i_K} \rangle$$

of processors that compute the gradient using ratings in the column block C_β according to S. That is, p_{i_1} updates the rows of $\mathbf{H}_\beta$ in the first sub-epoch, p_{i_2} in the second sub-epoch and so forth. Furthermore, we define a distance metric d_β^{xy} between two processors p_x and p_y updating $\mathbf{H}_\beta$ as

$$d_\beta^{xy} = \mathcal{S}_\beta^{-1}(p_x) - \mathcal{S}_\beta^{-1}(p_y). \tag{5.2}$$

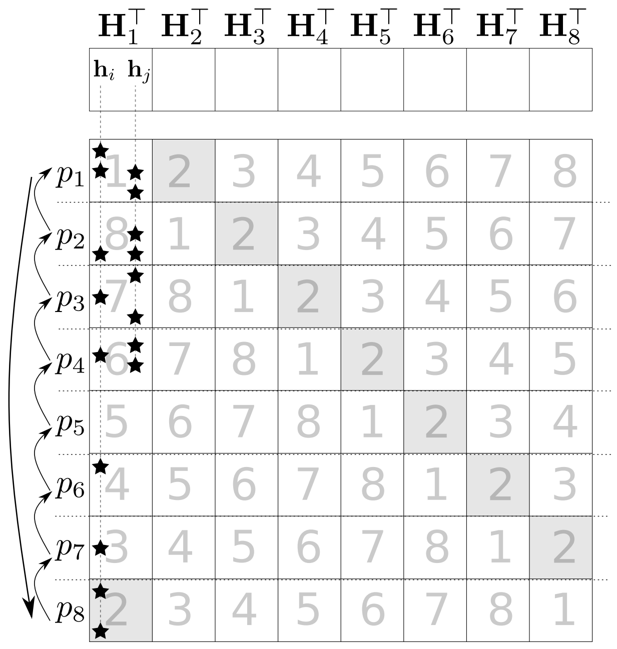


Figure 5.2: The numbers identify the sub-matrix blocks that constitute a stratum in a ring strata with *seed* = 1. Stratum $\mathcal{S}_2$ is highlighted. Side arrows show the processor update order of $\mathbf{h}_i$ and $\mathbf{h}_j$.

This distance quantifies the number of sub-epochs elapsed after p_x updates rows in $\mathbf{H}_\beta$ and before p_y does so.

5.3.1 Communication in DSGD

In the original DSGD algorithm [27], after processor p_x updates row block $\mathbf{H}_\beta$ in sub-epoch k, it sends the rows in $\mathbf{H}_\beta$ to the the processor that will update $\mathbf{H}_\beta$ in sub-epoch $k+1$. Therefore, at each sub-epoch, each processor sends a whole row block of $\mathbf{H}$ to exactly one processor. For instance, assuming the DSGD is executed according to the ring strata given in Figure 5.2, after sub-epoch 1 is completed p_1 sends $\mathbf{H}_1$ to p_8, p_8 sends $\mathbf{H}_8$ to p_7 and so forth.

In Figure 5.2, the update sequence for row block $\mathbf{H}_1$ is $\Upsilon_1 = \langle p_1, p_8, p_7, p_6, p_5, p_4, p_3, p_2 \rangle$. The communication of $\mathbf{h}_i \in \mathbf{H}_1$ through the subsequence/subchain $p_1 \rightarrow p_8 \rightarrow p_7 \rightarrow p_6$ does not incur any extra volume since each of these processors update $\mathbf{h}_i$. However, p_5 does not update $\mathbf{h}_i$ hence p_5 needs to receive the up-to-date $\mathbf{h}_i$ from p_6 and forward it to p_4 in the next sub-epoch. In this case, $\mathbf{h}_i$ incurs F words of forwarding overhead. In the case of $\mathbf{h}_j \in \mathbf{H}_1$, the first processor to update it after p_1 is p_4. Therefore, four forwarding communications, each of size F, are incurred due to $\mathbf{h}_j$ in $p_1 \to p_8 \to p_7 \to p_6 \to p_5$. In the worst case, an $\mathbf{H}$-matrix row can be forwarded $K-1$ times incurring $F(K-1)$ extra words of communication volume. This indicates that the volume due to forwarding has a loose upper bound of $\mathcal{O}(MF(K-1))$ words. Let $\lambda(\mathbf{h}_j)$ denote the number of processors that update $\mathbf{h}_j \in \mathbf{H}_\beta$, then the amount of forwarding overhead of $\mathbf{h}_j$ is $K - \lambda(\mathbf{h}_j)$.

The communication scheme of the DSGD method has the nice property of very low latency overhead since it restricts the number of messages sent by any processor at any sub-epoch to one. However, this scheme suffers from increasing the bandwidth overhead (communication volume) due to forwarding the $\mathbf{H}$-matrix rows. For each epoch, the communication volume sent by each processor is equal to $F \times M \times K$ as each processor sends approximately M/K dense $\mathbf{H}$-matrix rows each of size F during each of the K sub-epochs. Especially for highly sparse rating matrices, it is clear that the volume of communication performed is much more than the required, and the increased bandwidth overhead due to forwarding can be prohibitive as K increases.

5.3.2 Essential Communication for distributed SSGD

The communication of $\mathbf{H}$-matrix rows required for correctly executing SSGD in a distributed fashion is described according to the following definition:

Definition 1 (*d*-gap rows)**.** Consider two nonadjacent processors in Υ_β: p_{i_x} and p_{i_y} such that $x < y$. During DSGD, a row $\mathbf{h}_j \in \mathbf{H}_\beta$ is called a zero-gap row in terms of p_{i_x} and p_{i_y} if it is updated by both p_{i_x} and p_{i_x+1}. $\mathbf{h}_j$ is called one-gap row if it is updated by both p_{i_x} and p_{i_x+2} but not p_{i_x+1}. $\mathbf{h}_j$ is called d-gap row if it is updated by both p_{i_x} and p_{i_y} but not any of the $d = d_\beta^{xy}$ processors in-between (that is, in $\langle p_{i_{x+1}}, \ldots, p_{i_{y-1}} \rangle$). The set of all such d-gap rows between p_{i_x} and p_{i_y}

in $\mathbf{H}_\beta$ is given by

$$\mathbf{H}_\beta^{i_x i_y} = \{\mathbf{h}_j \mid (\exists r_{ij})[r_{ij} \in \mathbf{R}_{x\beta} \cap \mathbf{R}_{y\beta} \wedge r_{ij} \notin \mathbf{R}_{(x+1)\beta} \cup \cdots \cup \mathbf{R}_{(y-1)\beta}]\}. \quad (5.3)$$

Communicating $\mathbf{H}_\beta^{i_x i_y}$ from p_{i_x} to p_{i_y} after p_{i_x} processes the ratings in 2D block $\mathbf{R}_{i_x\beta}$ and before p_{i_y} starts processing the ratings in $\mathbf{R}_{i_y\beta}$ guarantees a correct distributed row-parallel SSGD execution.

The DSGD scheme discussed in Sec. 5.3.1 guarantees the correctness of the SSGD since up-to-date $\mathbf{h}_j \in \mathbf{H}_\beta^{i_x i_y}$ will eventually reach p_{i_y} from p_{i_x}, assuming $x < y$ in Υ_β, via forwarding through $\langle p_{i_{x+1}}, \ldots, p_{i_{y-1}} \rangle$.

5.4 A Framework for Scaling SSGD

5.4.1 Communicating d-gap rows through P2P messages

We propose to avoid the forwarding overhead by sending an updated **H**-matrix row to the processor that updates it next directly through P2P communications. At the beginning of sub-epoch k, processor p_x sends P2P messages to a set of processors $SendSet^k(p_x)$ and receives from $RecvSet^k(p_x)$. These two sets can be respectively constructed as

$$SendSet^k(p_x) = \{p_y \mid \mathbf{H}^{xy}_{\mathcal{S}_{k-1}(p_x)} \neq \emptyset\}, \quad (5.4)$$

$$RecvSet^k(p_x) = \{p_y \mid \mathbf{H}^{yx}_{\mathcal{S}_k(p_x)} \neq \emptyset\}. \quad (5.5)$$

For example, in Figure 5.2, at the beginning of the second sub-epoch p_1 sends $\mathbf{h}_i$ to p_8 and $\mathbf{h}_j$ to p_4.

Algorithm 11 presents the P2P-based parallel SSGD algorithm for processor p_x. At line 3, processor p_1 picks strata S and broadcasts to all other processors. At line 4, p_x determines the communication requirement according to (5.3) and constructs the send/receive information of the P2P messages according to (5.4)

Algorithm 11 Point-to-Point parallel SSGD on processor p_x

```
Require: Rating matrix $\mathbf{R}$, Processor count $K$
 1: Initialize local factor matrices $\mathbf{W}$ and $\mathbf{H}$ randomly
 2: repeat
 3:     Receive strata $S$ from $p_1$ through Bcast.
 4:     Construct P2P communication according to $S$
 5:     for $k = 1$ to $K$ do                                   ▷ For each sub-epoch
 6:         $\beta_{\text{prev}} \leftarrow \mathcal{S}_{k-1}(p_x)$
 7:         $\beta_{\text{curr}} \leftarrow \mathcal{S}_k(p_x)$;
 8:         for each $p_y \in SendSet^k(p_x)$ do
 9:             Send $\mathbf{H}^{xy}_{\beta_{\text{prev}}}$ to $p_y$
10:         for each $p_z \in RecvSet^k(p_x)$ do
11:             Receive $\mathbf{H}^{zx}_{\beta_{\text{curr}}}$ from $p_z$
12:         for each $r_{ij} \in \mathbf{R}_{x\beta_{\text{curr}}}$ do
13:             $\mathbf{w}_i = \mathbf{w}_i - \epsilon[(r_{ij} - \hat{r}_{ij})\mathbf{h}_j + \gamma\mathbf{w}_i]$
14:             $\mathbf{h}_j = \mathbf{h}_j - \epsilon[(r_{ij} - \hat{r}_{ij})\mathbf{w}_i + \gamma\mathbf{h}_j]$
15: until convergence or max. number of epochs reached
```

and (5.5). Then, the up-to-date rows required in the current sub-epoch are communicated at lines 8-13 through P2P messages. The SGD updates are performed at lines 14 and 15 respectively according to (2.3) and (2.4).

5.4.2 Efficiently constructing d-gap row sets

Computing d-gap $\mathbf{H}$-matrix rows using (5.3) has recurring computations for different instances. For example, computing $\mathbf{H}^{i_x i_y}_{\beta}$ and $\mathbf{H}^{i_x i_y+1}_{\beta}$ would require computing the same $\mathbf{R}_{(x+1)\beta} \cup \cdots \cup \mathbf{R}_{(y-1)\beta}$ term twice. For an efficient computation, we devise an algorithm that utilizes a dynamic programming formulation leveraging efficient bulk bit-wise operations.

Consider a binary string $B^{i_x}_{\beta} \in \{0,1\}^{n_\beta}$ of length $n_\beta = |\mathbf{H}_\beta|$, such that the bth entry of $B^{i_x}_{\beta}$ is set to '1' if p_{i_x} updates the bth row in $\mathbf{H}_\beta$, and set to '0' otherwise. Then, the indices of the rows to be communicated between p_{i_x} and p_{i_y} are the indices of the 1-bits in

$$\Psi^{i_x,i_y}_{\beta} = B^{i_x}_{\beta} \wedge B^{i_y}_{\beta} \oplus (B^{i_x}_{\beta} \wedge B^{i_y}_{\beta} \wedge (B^{i_x+1}_{\beta} \vee \cdots \vee B^{i_y-1}_{\beta})), \tag{5.6}$$

where $\oplus$, $\wedge$ and $\vee$ respectively denote logical exclusive OR (XOR), logical AND and logical OR operations. The term $(B_\beta^{i_x+1} \vee \cdots \vee B_\beta^{i_y-1})$ in (5.6) can be computed incrementally thanks to the associativity property of the $\vee$ operation.

Algorithm 12 Find d-gap $\mathbf{H}$-matrix rows on processor p_x

```
Require: Rating matrix $\mathbf{R}$, Processor count $K$, Strata $S$
 1: for each $\mathbf{H}_\beta \in \mathbf{H}$ do
 2:     Compute $\Upsilon_\beta^{p_x}$
 3:     $mask \leftarrow B_\beta^{\Upsilon_\beta^{p_x}[2]}$
 4:     for $k = 2$ to $K$ do
 5:         $p_y \leftarrow \Upsilon_\beta^{p_x}[k]$
 6:         $\Psi_\beta^{p_x,p_y} \leftarrow B_\beta^x \wedge B_\beta^y \oplus (B_\beta^x \wedge B_\beta^y \wedge mask)$
 7:         $\mathbf{H}_\beta^{xy} \leftarrow \{\mathbf{h}_i \in \mathbf{H}_\beta \mid \Psi_\beta^{p_x,p_y}[i] = 1\}$
 8:         $mask \leftarrow mask \vee B_\beta^y$
```

Given $\mathbf{H}_\beta$ and Υ_β, we define $\Upsilon_\beta^{p_x}$ as the sequence of processors updating $\mathbf{H}_\beta$ starting from p_x. $\Upsilon_\beta^{p_x}$ can be obtained from Υ_β by left-rotating the sequence until p_x is at the first index. Algorithm 12 presents the efficient dynamic-programming-based computation of the d-gap $\mathbf{H}$-matrix rows between p_x and the other $K-1$ processors. For each $\mathbf{H}_\beta$, the order of processors updating $\mathbf{H}_\beta$ starting from p_x according to strata S is maintained in $\Upsilon_\beta^{p_x}$ (line 3). Then, p_x constructs the d-gap rows one by one according to this order leveraging the bottom-up construction of the term $(B_\beta^{i_x+1} \vee \cdots \vee B_\beta^{i_y-1})$, lines 4-9.

5.4.3 Hold & Combine strategy for reducing latency

Using P2P messages to communicate the updated rows without forwarding is indispensable for reducing the bandwidth overhead of the communication. However, it has a high potential of increasing the latency overhead via increasing the number of messages performed per epoch compared to DSGD. In DSGD, a processor sends K messages per epoch (one message to one processor at each sub-epoch), whereas using the P2P requires sending at most $K \times (K-1)$ messages per epoch (up to $K-1$ messages from each of the K processors at each sub-epoch). We propose the hold and combine (H&C) strategy to reduce the upper-bound on the number of messages sent per epoch to $\mathcal{O}(K \lg K)$.

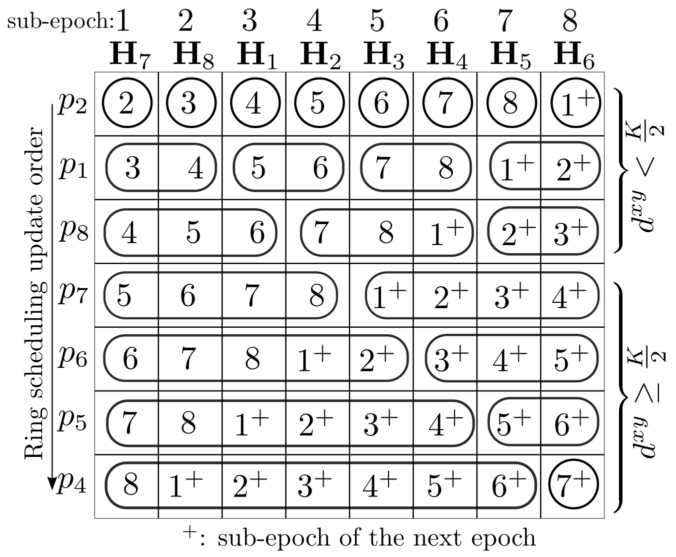

sub-epoch: 1 2 3 4 5 6 7 8
$\mathbf{H}_7$ $\mathbf{H}_8$ $\mathbf{H}_1$ $\mathbf{H}_2$ $\mathbf{H}_3$ $\mathbf{H}_4$ $\mathbf{H}_5$ $\mathbf{H}_6$
p_2: 2 3 4 5 6 7 8 1^+
p_1: 3 4 5 6 7 8 1^+ 2^+
p_8: 4 5 6 7 8 1^+ 2^+ 3^+
p_7: 5 6 7 8 1^+ 2^+ 3^+ 4^+
p_6: 6 7 8 1^+ 2^+ 3^+ 4^+ 5^+
p_5: 7 8 1^+ 2^+ 3^+ 4^+ 5^+ 6^+
p_4: 8 1^+ 2^+ 3^+ 4^+ 5^+ 6^+ 7^+
Ring scheduling update order
$d^{xy} < \frac{K}{2}$
$d^{xy} \geq \frac{K}{2}$
$^+$: sub-epoch of the next epoch

Figure 5.3: An example TSMS for p_3. The rows are the processors that p_3 communicates with sorted according to their distance from p_3. The columns represent both the sub-epochs and the **H**-matrix blocks to be updated at each sub-epoch. An entry (p_y, $\mathbf{H}_\beta$) gives the sub-epoch at which p_y updates $\mathbf{H}_\beta$ after p_3 does (note that this sub-epoch might be in the next epoch). The circles show the messages that can be combined.

Definition 2. Fixed-distance strata is any strata that satisfies

$$d^{xy}_{\alpha} = d^{xy}_{\beta} \quad \text{for any pair of } \mathbf{H}\text{-matrix row } \alpha \text{ and } \beta. \tag{5.7}$$

That is, the fixed-distance strata have the property of constant distance between any two processors regardless of the **H**-matrix row block they are updating. We refer to the distance between two processors p_x and p_y in a fixed-distance strata as d^{xy}. Any ring strata scheduled with (5.1) is a fixed-distance strata.

During an SGD epoch, the communication of $\mathbf{H}^{xy}_{\beta}$ should be performed after p_x updates $\mathbf{H}_\beta$ in sub-epoch k and before p_y starts updating $\mathbf{H}_\beta$. This means that $\mathbf{H}^{xy}_{\beta}$ can be sent at the beginning of any sub-epoch between $k+1$ and $k+d^{xy}_{\beta}$. Now consider the communication of $\mathbf{H}^{xy}_{\beta}$ at sub-epoch k in a fixed-distance strata. Observe that when sub-epoch $k+d^{xy}$ is reached, all the rows in $\mathbf{H}^{xy}_{\mathcal{S}_{k+1}(p_x)}, \mathbf{H}^{xy}_{\mathcal{S}_{k+2}(p_x)}, \ldots, \mathbf{H}^{xy}_{\mathcal{S}_{k+d^{xy}-1}(p_x)}$ are already updated by p_x and ready to be sent to p_y. So, these rows can be held by p_x and sent all at once in one message to p_y in sub-epoch $k+d^{xy}$ along with $\mathbf{H}^{xy}_{\beta}$.

Utilizing fixed-distance strata, we propose to hold P2P messages and combine them as follows: If $d^{xy} \geq K/2$, then the messages between p_x and p_y in an epoch can be combined into two or more P2P messages. This is because if $d^{xy} = K-1$ then one message is needed for $K-1$ **H**-matrix row blocks and another message needed for the last block. Otherwise if $d^{xy} < K/2$, then the messages between p_x and p_y can be combined in $\lceil K/d^{xy} \rceil$ P2P messages. Therefore, the number of messages sent per processor per epoch can be computed by

$$\sum_{i=1}^{\frac{K}{2}} 2 + \sum_{i=1}^{\frac{K-1}{2}} \frac{K}{i}.$$

The second summation is a harmonic series which can be approximated by $\ln(K-1)/2 + 1$, thus

$$\sum_{i=1}^{\frac{K}{2}} 2 + \sum_{i=1}^{\frac{K-1}{2}} \frac{K}{i} \approx K + K \ln(K-1)/2 \leq K \lg K \tag{5.8}$$

is the upper bound on the number of messages sent per processor per epoch.

To facilitate the presentation of the H&C strategy, we assume that each processor constructs a tabular-shaped message schedule (TSMS). In the TSMS of p_x, rows are the $K-1$ processors that p_x communicates with during an epoch, and columns represent sub-epochs as well as the corresponding **H**-matrix row blocks updated by p_x. Each table entry TSMS(p_y, $\mathbf{H}_\beta$) represents the sub-epoch $\mathcal{S}_\beta^{-1}(p_y)$.

Figure 5.3 shows a TSMS for p_3 using strata with *seed* = 5. In the figure, the circled TSMS entries denote the messages (**H**-matrix row blocks) that can be combined. For instance, the communication requirement between p_3 and p_7 during an SGD epoch can be done with two messages. The first message, required at the beginning of sub-epoch 5, consists of $\mathbf{H}_7^{3,7} \cup \mathbf{H}_8^{3,7} \cup \mathbf{H}_1^{3,7} \cup \mathbf{H}_2^{3,7}$. The second message, required at the beginning of sub-epoch 1 of the next SGD epoch, consists of $\mathbf{H}_3^{3,7} \cup \mathbf{H}_4^{3,7} \cup \mathbf{H}_5^{3,7} \cup \mathbf{H}_6^{3,7}$. Observe that the sub-epoch at which the combined message should be sent is decided by the first **H**-matrix block of the combined message. For instance, the first message to p_7 must arrive before p_7 starts updating rows in $\mathbf{H}_7$ which is sub-epoch 5.

Algorithm 13 Message combining strategy on processor p_x

```
Require: $SendSet^k(p_x)$, $\mathbf{H}^{xy}\ \forall k, y \in [K] \wedge y \neq x$, $S$
1: for $k = 1$ to $K$ do
2:     for each $p_y \in SendSet^k(p_x)$ do
3:         $m^{\text{id}} \leftarrow \lceil \frac{k}{d^{xy}} \rceil$                    ▷ get the message ID
   //get the $\mathbf{H}$-matrix block that $p_x$ updates at SE $k$
4:         $\beta \leftarrow \mathcal{S}_k(p_x)$
   //add the $d$-gap rows $\mathbf{H}^{xy}_{\beta}$ to Msg $m^{\text{id}}$
5:         $M^{xy}_{m^{\text{id}}} \leftarrow M^{xy}_{m^{\text{id}}} \cup \mathbf{H}^{xy}_{\beta}$
   // When does $p_x$ update the first block of $m^{\text{id}}$ ?
6:         $t \leftarrow (m^{\text{id}} - 1) * d^{xy} + 1$
   // get the $\mathbf{H}$-matrix block that $p_x$ updates at SE $t$
7:         $\eta \leftarrow \mathcal{S}_t(p_x)$
   // get the sub-epoch $s$ at which $p_y$ updates $\mathbf{H}_{\eta}$
8:         $s \leftarrow \mathcal{S}^{-1}_{\eta}(p_y)$
9:         $cSendSet^s(p_x) \leftarrow cSendSet^s(p_x) \cup p_y$
```

Algorithm 13 shows the procedure to construct combined messages from P2P messages at p_x. Given the $SendSet^k(p_x)$ $\forall k \in \{1, \ldots, K\}$ and the d-gap rows between p_x and $\{p_y \mid y \neq x\}$, the combined messages are constructed as follows: There are $\lceil K/d^{xy} \rceil$ possible messages to p_y each of which is identified by m^{id}. For each $p_y \in SendSet^k(p_x)$ the rows in $\mathbf{H}^{xy}_{\beta}$ are assigned to a combined message $M^{xy}_{m^{\text{id}}}$ (lines 3 and 4). Then, p_y is added to the new send set of the sub-epoch at which message m^{id} is sent (lines 5-8).

Algorithm 11 can be modified to accommodate the H&C strategy as follows: After constructing the P2P communication (line 4), Algorithm 13 is used to combine the messages. Then, lines 8-13 can be replaced with the sending/receiving of combined messages; for each p_y in $cSendSet^k(p_x)$ a combined message is identified using $m^{\text{id}} = \lceil k/d^{xy} \rceil$ and sent to p_y, and similarly so for receiving from each p_z in $cRecvSet^k(p_x)$.

It is important to make sure that the $K \lg K$ messages sent per epoch are uniformly distributed over K sub-epochs. Otherwise, some sub-epochs will constitute a performance bottleneck due to high number of messages. We show that utilizing Algorithm 13 for combining the messages has the nice property of limiting the expected number of messages sent by each processor at each sub-epoch to $\mathcal{O}(\lg K)$.

Theorem 2. *Using the H&C strategy, the expected number of messages sent by each processor at each sub-epoch is* $\mathcal{O}(\lg K)$.

Proof. Consider a set $\mathcal{F}^{xy}$ that consists of all sub-epochs wherein a message is sent from p_x to p_y. For each sub-epoch k, the function

$$\sigma(k, p_x, p_y) = \begin{cases} 1 & k \in \mathcal{F}^{xy} \\ 0 & \text{otherwise} \end{cases}$$

defines if there is a message to be sent from p_x to p_y in k.

We can prove that $\mathcal{O}(\lg K)$ messages are sent at each sub-epoch as follows. The number of messages per sub-epoch is equal to the number of occurrences of that sub-epoch in $\bigcup_{y \in [K] \wedge y \neq x} \mathcal{F}^{xy}$. For each processor p_y with distance d^{xy}, the probability that k is one of the sub-epochs in which a message is sent to p_y is equal to $1/d^{xy}$. In other words, given K sub-epochs, the probability that sub-epoch k will be used to send one of the $\lceil K/d^{xy} \rceil$ messages is $1/d^{xy}$. Then, the expected number of messages from p_x to p_y at sub-epoch k is

$$\mathbf{E}[\sigma(k, p_x, p_y)] = \mathbf{Pr}(\sigma(k, p_x, p_y) = 1) \times 1 + \mathbf{Pr}(\sigma(k, p_x, p_y) = 0) \times 0 = \frac{1}{d^{xy}}. \quad (5.9)$$

Using linearity of expectation, the expected total number of messages sent by p_x at sub-epoch k is

$$\mathbf{E}[\sum_{p_y \neq p_x} \sigma(k, p_x, p_y)] = \sum_{i=1}^{K} \frac{1}{i} \approx \ln K + 1 \leq \lg K + 1. \quad (5.10)$$

□

5.5 HP Model for Reducing Bandwidth Cost

There exists two hypergraph models for 1D partitioning of sparse matrices for SpMV-like kernels; namely the column-net model for rowwise partitioning and the row-net model for columnwise partitioning [52]. In these models, the "connectivity-1" metric [52] is utilized for partitioning objective of reducing the communication volume in SpMV-like kernels, whereas the partitioning constraint is maintaining computational balance among processors. As mentioned earlier, rating matrices usually have larger number of rows than columns, hence we mainly focus on rowwise partitioning of rating matrix $\mathbf{R}$. The hypergraph model discussed here is topologically similar to the column-net model, however the cutsize metric utilized in the partitioning objective is different.

In the hypergraph model $\mathcal{H}_{\mathbf{R}} = (\mathcal{V}, \mathcal{N})$, there exists a vertex $v_i \in \mathcal{V}$ for each row $\mathbf{r}_i$ of $\mathbf{R}$ and a net (hyperedge) $n_j \in \mathcal{N}$ for each column $\mathbf{c}_j$ of $\mathbf{R}$. Each net n_j connects the vertices corresponding to the $\mathbf{R}$-matrix rows that contain nonzeros in column $\mathbf{c}_j$. That is, $Pins(n_j) = \{v_i \in \mathcal{V} \mid r_{ij} \neq 0\}$. Each vertex v_i is associated with a weight equals to the number of nonzeros in row $\mathbf{r}_i$. Each net is associated with a cost F.

A K-way partition $\Pi(\mathcal{H}_{\mathbf{R}}) = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is decoded as a K-way rowwise partition of $\mathbf{R}$, where the rows corresponding to the vertices in part $\mathcal{V}_\alpha$ constitute the row block $\mathbf{R}_\alpha$, for $\alpha = 1, 2, \ldots, K$. Without loss of generality, row block $\mathbf{R}_\alpha$ is assigned to processor p_α for $\alpha = 1, 2, \ldots, K$. The $\mathbf{W}$-matrix rows are partitioned conformably with the $\mathbf{R}$-matrix row partition. That is, $\mathbf{W}$-matrix rows in $\mathbf{W}_\alpha$ correspond to the $\mathbf{R}$-matrix rows in $\mathbf{R}_\alpha$.

In partition $\Pi(\mathcal{H}_{\mathbf{R}})$, the weight of each part is equal to the sum of the weights of the vertices in that part. Hence, the partitioning constraint of maintaining balance on the part weights encodes maintaining balance on the nonzero counts of the R-matrix row blocks. This in turn corresponds to maintaining balance on the computational loads of the processors.

In this model, $\Lambda(n_j)$ also represents the set of $\mathbf{R}$-matrix row blocks that has at least one nonzero in column $\mathbf{c}_j$ of $\mathbf{R}$. Hence, the connectivity set of net n_j denotes the set of processors that update the $\mathbf{H}$-matrix row $\mathbf{h}_j$. Consider the $\mathbf{H}$-matrix row $\mathbf{h}_j$ corresponding to a cut net n_j in the P2P communication scheme. Also consider $\mathbf{h}_j$ update sequence defined using the connectivity set and strata.

For each epoch, each processor except the last processor in the sequence should send its updated $\mathbf{h}_j$ value once to the next processor in the sequence. The last processor sends its updated $\mathbf{h}_j$ value to the first processor for the next iteration. Hence, each cut net n_j incurs a communication volume of $F\lambda(n_j)$. On the other hand, uncut nets incurs no communication. Therefore, cutsize which encapsulates the total communication volume during an SGD epoch can be computed as

$$\sum_{n_j \ni \lambda(n_j)>1} F\lambda(n_j). \tag{5.11}$$

Among the various cutsize metrics in the literature, cutsize (5.11) is called as the sum of external degrees (SOED) [80].

There exists several successful hypergraph partitioning tools that utilize multilevel recursive bipartitioning (RB) algorithms. Among these partitioning tools, to our knowledge, only *hMETIS* [80] supports the SOED metric via direct multiway partitioning [88]. In fact Karypis and Kumar [88] clearly indicates that RB framework does not allow directly optimizing the SOED metric. Here, we propose an RB-based algorithm that encodes the minimization of the SOED metric correctly.

In order to encode the SOED metric (5.11), we propose the following strategy during the RB paradigm. We assign a cost of $2F$ to each net of the initial hypergraph. Then, after each RB step, internal nets inherit their cost, whereas splitted nets are assigned a cost of F. That is, a net holds its cost of $2F$ until it becomes cut for the first time, then a cost of F is assigned to each of split subnets and they inherit their cost of F through the further RB steps until the end of the partitioning. Hence, when a net becomes cut for the first time it incurs $2F$ to the cutsize, then whenever its subnets become cut they incur F to the cutsize. In this way, the sum of all cut net costs encountered during the overall RB algorithm becomes equal to the SOED metric (5.11).

5.6 Experimental Evaluations

5.6.1 Experimental Framework

We evaluate the contributions proposed in this chapter through comparing three methods implementing parallel SSGD using six real-world rating matrices. The first method, DSGD, is the algorithm proposed in the original work of Gemulla et al. [27]. DSGD performs block-wise communication of **H**-matrix row blocks in each sub-epoch. The second method, P2P, uses P2P messages as in Algorithm 11. The third, H&C, uses combined P2P messages (Algorithm 13) for communication.

In all three methods, column-to-stratum assignments are done randomly in such a way that the number of columns per stratum differs by at most one. Row-to-processor assignments are obtained either randomly in a way similar to that of column-to-stratum assignments, or using the HP method discussed in Section 2.4. Whenever the former is used, the method will be prefixed by RAND, whereas if the latter is used the method will be prefixed by HP. The HP method is implemented according to the RB framework described in Section 2.4 to encapsulate the SOED metric. In order to obtain two-way partitions on the (sub)hypergraphs at each RB level, we use the HP tool PaToH [52] with default parameters in `SPEED` mode.

We implemented the parallel SSGD code that includes DSGD, P2P and H&C in C and used MPI for inter-process communications. We perform our experiments on an HPC system with AMD EPYC 7742 processors and a high-speed HDR InfiniBand network with 200Gb/s bandwidth.

We compare the three methods in terms of communication cost metrics as well as SGD iteration time. The communication cost metrics consist of bandwidth-oriented metrics: *sum-max vol* and *tot vol*, and latency-oriented metrics: *sum-max msgs* and *tot msgs*. *sum-max msgs* is calculated as follows: at each sub-epoch, the number of messages sent by the bottleneck processor (the processor that sends highest number of messages) is obtained. Then, the summation is taken over all K sub-epochs. That is,

$$\textit{sum-max msgs} = \sum_{k=1}^{K} \max_{x\in[K]}(|SendSet^k(p_x)|).$$

In a similar way, *sum-max vol* is computed as

$$\textit{sum-max vol} = \sum_{k=1}^{K} \max_{x\in[K]}(SendVol^k(p_x)).$$

tot msgs and *tot vol* are respectively computed as

$$\textit{tot msgs} = \sum_{k=1}^{K} \sum_{x\in[K]} (|SendSet^k(p_x)|),$$

$$\textit{tot vol} = \sum_{k=1}^{K} \sum_{x\in[K]} (SendVol^k(p_x)).$$

Here, $SendVol^k(p_x) = |\mathbf{H}_{\mathcal{S}_{k-1}(p_x)}|$ if DSGD is used, and $SendVol^k(p_x) = \sum_{p_y} |\mathbf{H}^{xy}_{\mathcal{S}_{k-1}(p_x)}|$ if P2P or H&C are used. Whenever the values for the volume of communication are presented, these values are normalized with respect to F. This uncoupling of F from the volume values helps evaluate the proposed methods and model with any F value.

Table 5.1: Properties of matrices in the dataset

Matrix	#rows	#cols	#nnz	density
`Amz Items`	21.177M	9.874M	82.677M	3.95E-07
`Amz Books`	8.026M	2.330M	22.50M	1.20E-06
`Amz Clothing & Jewelry`	3.117M	1.136M	5.75M	1.62E-06
`Goodreads Reviews`	0.465M	2.080M	15.740M	1.63E-05
`Google Reviews`	5.055M	3.117M	11.454M	7.27E-07
`Twitch`	15.524M	6.162M	474.677M	4.96E-06

Table 5.1 shows the real-world matrices used to evaluate the proposed methods and their properties. `Amz Items` contains product reviews from Amazon between May 1996 - July 2014 [89] with aggressive duplicate removal. The other two amazon datasets, Books and Clothing, are category-based subsets of the original comprehensive reviews. `Goodread Reviews` contains user ratings of books from the Goodreads website [90]. `Google Reviews` contains user ratings/reviews of local businesses from the Google Maps website [91, 92]. `Twitch` contains ratings relative to how much time a user spent on a stream in the Twitch streaming

website [93]. The original data does not contain any explicit ratings. We modified the dataset to represent (*user, stream, rating*) such that the rating value is proportional to the amount of time the user spent in the specific stream.

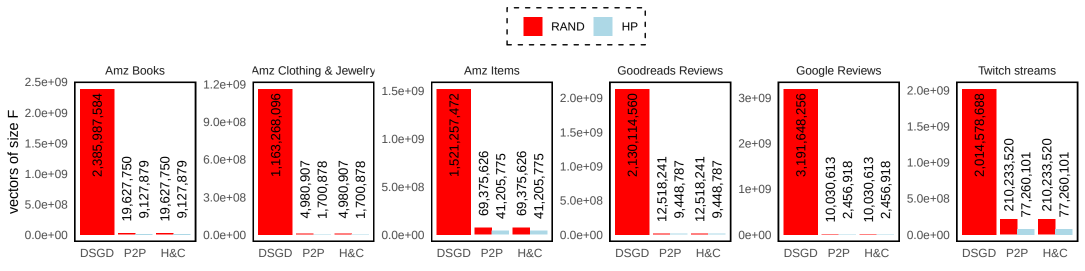
5.6.2 Evaluations with Communication Cost Metrics

Figs. 5.4a, 5.4b and 5.4c compare DSGD, P2P and H&C in terms of communication cost metrics *tot vol*, *sum-max vol* and *tot msgs* on $K = 1024$ processors. In the figures, the red bars denote RAND-based methods whereas light blue bars denote HP-based methods. HP does not affect DSGD's communication which is why HP is not applicable for DSGD and hence DSGD has only red bars. Comparison in terms of *sum-max msgs* will be discussed in Figure 5.5.

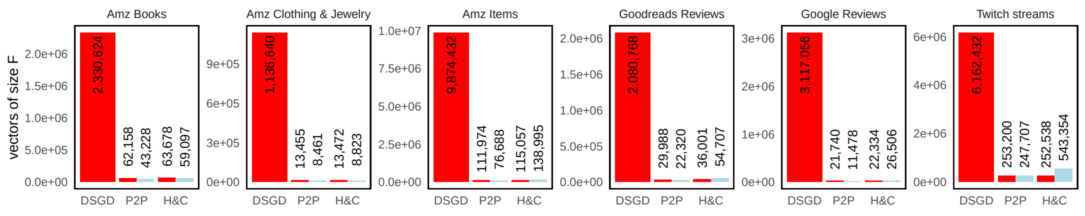
5.6.2.1 Bandwidth-oriented Communication Cost Metrics

As seen in Figure 5.4a, both P2P and H&C incur the essential amount of communication volume as defined in (5.6), without any forwarding overhead. Compared to DSGD, both RAND- and HP-based P2P and H&C methods incur significantly reduced amount of communication volume per epoch (more than 10x). Compared to RAND, the HP-based P2P and H&C methods incur significantly reduced volume (between 1.4x and 5x).

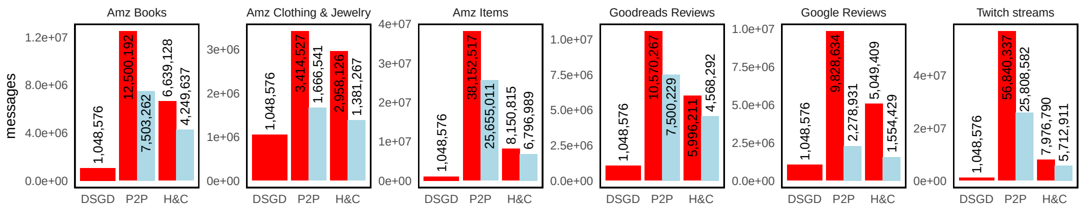
Figure 5.4b shows that in all matrix instances P2P and H&C have a significantly reduced *sum-max vol* compared to DSGD (more than 10x). H&C has slightly higher *sum-max vol* compared to P2P. This is because combining the messages disturbs the random volume balancing of P2P. As expected, HP-based P2P incurs less *sum-max vol* compared to RAND-based P2P. HP-based H&C shows a decrease in *sum-max vol* on two matrices (`Amz Books` and `Amz Clothing & Jewelry`), and an increase in other four matrices. This is because the HP method, when used for H&C, does not encapsulate reducing the *sum-max vol* metric.



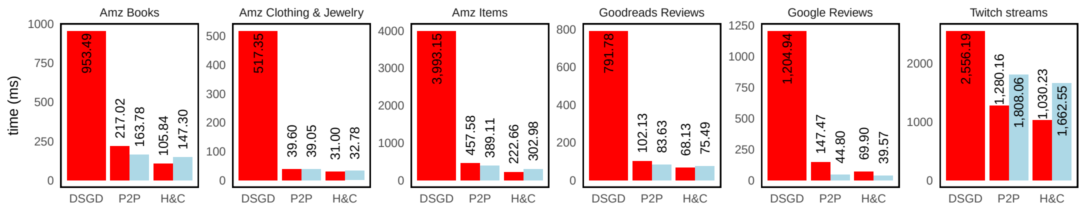
(a) *tot vol* in an SGD epoch



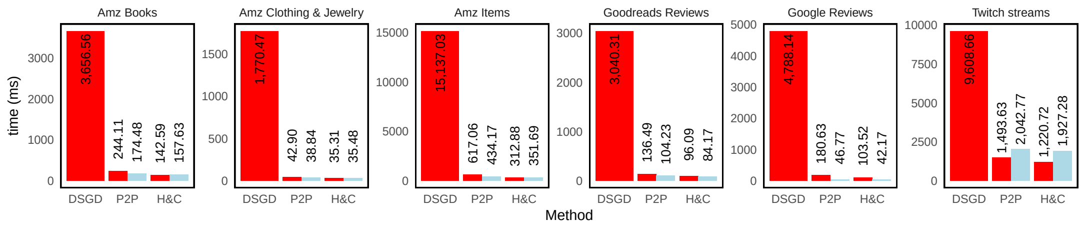
(b) *sum-max vol* in an SGD epoch



(c) *tot msgs* in an SGD epoch



(d) SGD iteration time when $F = 16$



(e) SGD iteration time when $F = 64$

Figure 5.4: Comparing RAND- and HP-based P2P and H&C methods against RAND-based DSGD using communication cost metrics (a to c) and SGD iteration time (d and e) using all dataset matrices on $K = 1024$ processors.

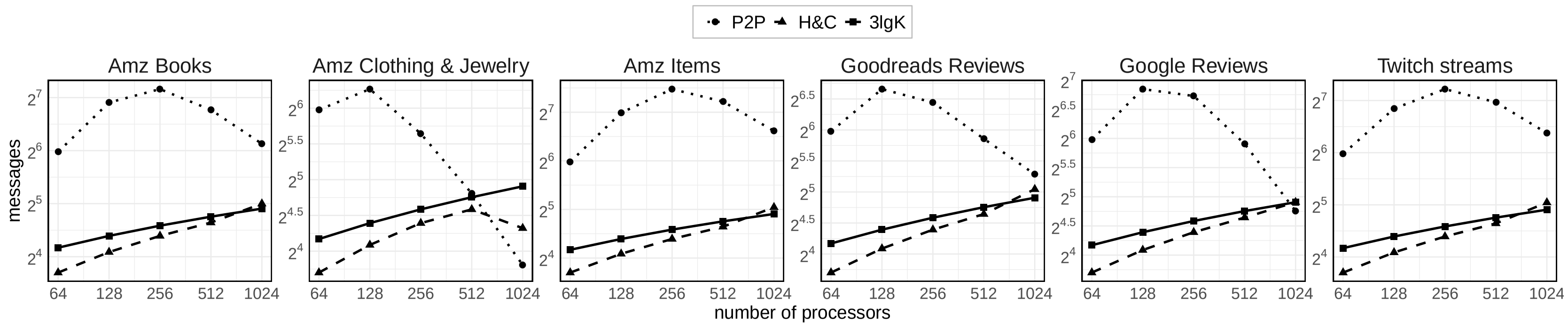
5.6.2.2 Latency-oriented Communication Cost Metrics

Figure 5.4c shows that the H&C method significantly reduces *tot msgs* on all dataset matrices. DSGD always incur a constant number of messages for each K value, thus *tot msgs* is always equal to $K^2 = (1024)^2 = 1048576$. *tot msgs* of P2P can go up to $K^2 \times (K-1)$. On the other hand, H&C keeps *tot msgs* limited to $\mathcal{O}(K^2 \lg K)$. Depending on the sparsity pattern of the matrix, *tot msgs* of P2P can be very high (e.g., `Amz Books`, `Amz Items` and `Twitch`) or relatively close to the lower bound (e.g., `Amz Clothing & Jewelry`). The H&C method successfully controls the fluctuation in the number of P2P messages thanks to the $\lg K$ factor. The significant reduction in *tot vol* of HP-based P2P and H&C methods compared to those of RAND-based is expected to reflect on the total number of messages, which is the case as shown in the figure.

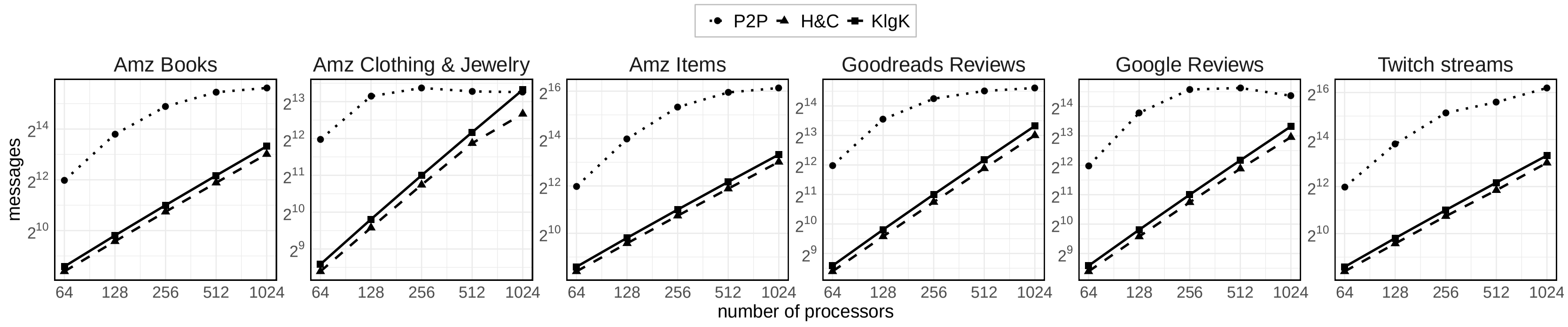
Figure 5.5 showcases the H&C method's regularization of messages sent per epoch over K sub-epochs. In order to experimentally verify the $\mathcal{O}(\lg K)$ bound given in Theorem 2, we introduce the *max-max msgs* metric as the maximum number of messages sent per sub-epoch among all sub-epochs. That is,

$$max\text{-}max\ msgs = max\{\max_{x\in[K]}(|SendSet^k(p_x)|) \mid k \in [K]\}.$$

As seen in Figure 5.5a, using H&C, *max-max msgs* is empirically found to be $\approx 3\times \lg K$, which is very close to the expected $\lg K$ bound on the number of messages per sub-epoch given in (5.10). The figure shows that P2P incurs high *max-max msgs* on $K=256$, and then the number starts to decrease as K increases. We believe this is attributed to the ability of random partitioning to balance P2P message counts and volume. In Figure 5.5b, the *sum-max msgs* metric is shown for all matrices in the dataset using P2P and H&C on $K = 64, \ldots, 1024$ processors. The figure shows the success of H&C in keeping the number of messages under the $K \lg K$ theoretical bound. Since the P2P sum-max messages do not decrease as K increases, this means maximum number of messages per sub-epoch are almost equal among all sub-epochs, especially when $K \geq 512$. On the other hand, although the H&C's *max-max msgs* come very close to those of P2P on some instances such as `Goodreads Reviews` and `Google Reviews`, *sum-max msgs* stay significantly less than those of P2P. This means that although the maximum number of messages sent per sub-epoch can reach $3\lg K$ in very few sub-epochs, it is still equal to or less than the expected $\lg K$ messages.



(a) *max-max msgs* in an SGD epoch



(b) *sum-max msgs* in an SGD epoch

Figure 5.5: Showcasing the upper bound of the max-max messages and sum-max messages sent per sub-epoch using the H&C method compared to P2P on $K = \{64, \dots, 1024\}$ processors.

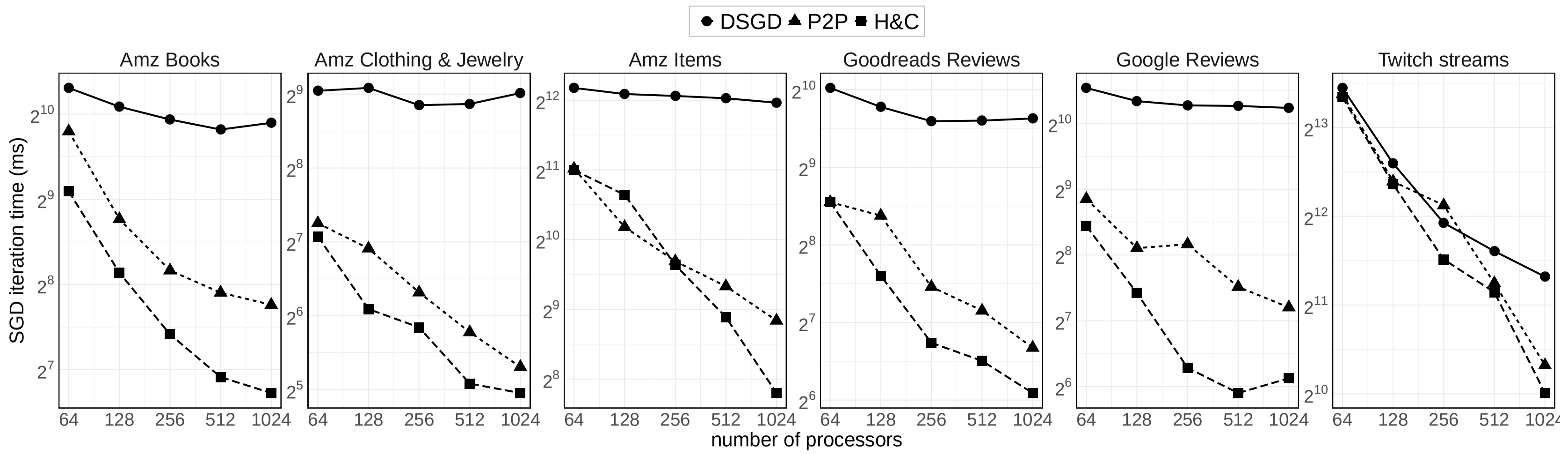
5.6.3 Evaluations with SGD Iteration Time

Figs. 5.4d and 5.4e compare the methods in terms of SGD iteration time on $K = 1024$ processors respectively using $F = 16$ and $F = 64$ values. The figure shows that the P2P improvement over DSGD is significant (more than 4x on all matrices, except for `Twitch` which is 1.4x) when $F = 16$. The improvement grows further as F increases to 64. It becomes more than 15x on all matrices except `Twitch`, and on `Twitch` the improvement becomes at least 4.7x.

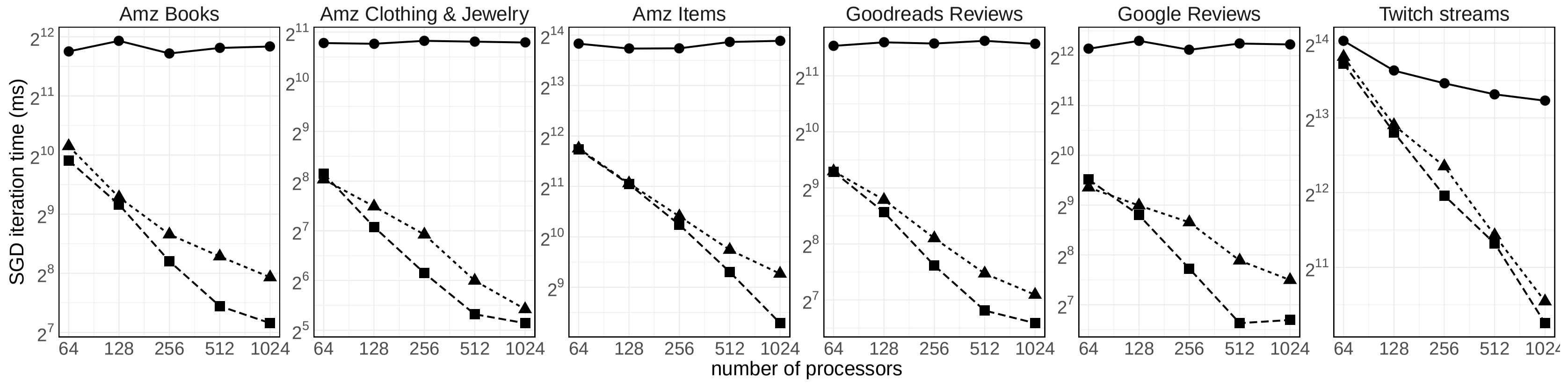
Using RAND, the H&C improvement over P2P is also significant. When $F = 16$, H&C improves the iteration runtime over P2P by 2x, 1.2x, 2x, 1.5x, 2.15x, and 1.25x respectively on `Amz Books`, `Amz Clothing & Jewelry`, `Amz Items`, `Goodreads Reviews`, `Google Reviews` and `Twitch`. When $F = 64$, the respective values become 1.7x, 1.2x, 2x, 1.4x, 1.74x, and 1.22x. The slight reduction in improvement is expected since increasing the value of F renders the application bandwidth-bound. Therefore, the effect of the H&C method with higher F values, although crucial, slightly diminishes.

Using HP improves the P2P runtime by 1.3x, 1.17x, 1.22x and 3.35x on `Amz Books`, `Amz Items`, `Goodreads Reviews` and `Google Reviews` when $F = 16$. On `Amz Clothing & Jewelry` there is no significant improvement and on `Twitch` there is deterioration by 1.4x. When $F = 64$, HP improves the P2P runtime by 1.4x, 1.42x, 1.3x and 3.9x respectively on `Amz Books`, `Amz Items`, `Goodreads Reviews` and `Google Reviews`. The increase in the gap between HP and RAND in terms of P2P runtime when F grows from 16 to 64 is expected since the HP method aims at reducing the total volume, effect of which is seen more with higher F values.

Figure 5.6 shows the strong scaling curves of RAND-based DSGD, P2P and H&C using two different F values on $K = \{64, 128, 256, 512, 1024\}$ processors. As seen in the figure, P2P and H&C show superior scaling compared to DSGD. H&C performs significantly better than P2P, especially with smaller F values. The difference in performance between P2P and H&C reduces with increasing F value since the communication in SGD becomes more bandwidth-bound.



(a) $F = 16$



(b) $F = 64$

Figure 5.6: Strong scaling curves of DSGD, P2P and H&C on $K = \{64, 128, 256, 512, 1024\}$ processors using all dataset matrices with two F values.

5.6.4 Evaluations with Loss Values

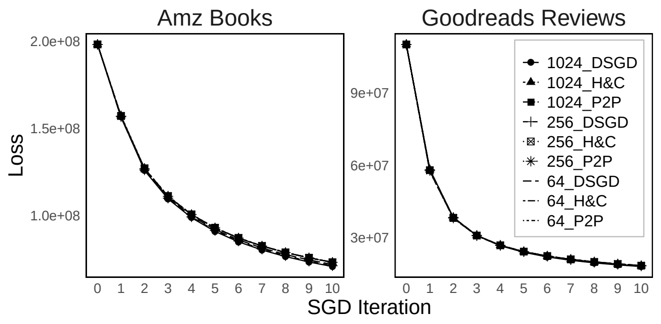
Since all the methods discussed in this chapter follow the stratified SGD algorithm, their loss values per iteration is expected to be very similar regardless of the communication strategy used or number of processors. We demonstrate this using Figure 5.7a. The figure shows the loss value (y-axis) following each SGD iteration (x-axis) of `Amz Books` and `Goodreads Reviews` using the RAND-based DSGD, P2P, H&C methods on $K = \{64, 256, 1024\}$ processors. The loss values are very close as expected thus the curves appear to be on top of each other.

Figure 5.7b shows the amount of time (x-axis) required to reach a certain loss value (y-axis) of `Amz Items` and `Google Reviews` using the RAND-based DSGD, P2P, H&C methods on 1024 processors. The figure shows that DSGD requires significantly more time to reach a certain loss value compared to P2P and H&C.

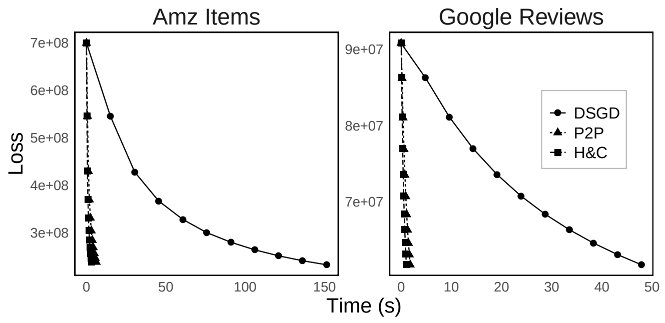
Figure 5.7c shows the scaling behavior of the RAND-based H&C method with `Amz Clothes & Jewelry` and `Twitch` in terms of loss value as the time increases.

5.7 Related Work

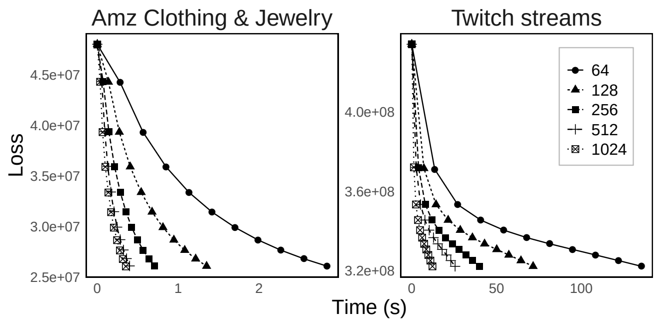
There exist several works in the literature that adopt the SSGD for parallel matrix completion. The work of Gemulla et at. [27] proposed the SSGD approach as well as the parallel DSGD algorithm discussed in Sections 5.2 and 5.3.1. Teflioudi et al. [46] proposed DSGD++, an improved DSGD framework for better performance. They use computation and communication overlaying through dividing the input matrix into $K \times 2K$ blocks, and in each of the K sub-epochs DSGD++ performs computation on K blocks while simultaneously communicating the other K blocks. They report up to 2.3x improvement over DSGD in terms of runtime. Yun et al. [47] extend the idea of DSGD++ in their framework, NOMAD, and divide the input matrix into $K \times M$ blocks. Each of the K processors dedicates ℓ threads to update ℓ **H**-matrix rows, and $M - \ell$ other threads for communication. Once processor p_x updates an **H**-matrix row, or a set of rows, it sends it/them to



(a) SGD iteration vs. loss for all methods on different K values.



(b) Time vs. loss for all methods on K=1024 processors.



(c) Time vs. loss for H&C on different K values.

Figure 5.7: Loss versus time and iteration counts for different methods and K values.

another processor p_y that has idle computation threads. DSGD, DSGD++ and NOMAD has the same total communication volume during an SGD epoch per processor which is equals to $F \times M \times K$ as discussed in Section 5.3.1. The number of messages sent per processor during an epoch of DSGD or DSGD++ has an upper bound of $\mathcal{O}(K)$, whereas NOMAD may send up to $\mathcal{O}(M)$ messages. Guo et al. [94] proposed a novel framework, BaPa, for improving the nonzero load balance of DSGD through a novel algorithm for balancing per-processor and per-epoch ratings. Their BaPa-based DSGD shows a significant runtime improvement on small number of processors (< 16). However, their results show that both the original DSGD as well as the BaPa-based DSGD stop scaling after 256 processors.

There are several asynchronous-SGD-based parallel matrix completion algorithms in the literature. ASGD [46] (shown in the upper part of Figure 5.1) is the simplest example of such algorithm. During ASGD, it is possible that several processors update the same $\mathbf{H}$-matrix row $\mathbf{h}_j$ at the same time (i.e., stale updates). This results in each processor having a different copy of $\mathbf{h}_j$. These copies are coordinated by sending them to a processor responsible for $\mathbf{h}_j$. This processor takes their average and then sends the up-to-date version of $\mathbf{h}_j$ back to the same set of processors. This type of coordination is done once or more during an SGD epoch [46, 95]. GASGD [95] extends ASGD by utilizing intelligent partitioning for balancing computational loads, reducing communication between processors, and reducing staleness. The authors utilize a bipartite graph model and propose a partitioning method based on the *balanced K-way vertex-cut* problem [96] to achieve the partitioning goals. Luo et al. [97] proposed a different strategy to facilitate asynchronously computing SGD in parallel which is called alternating SGD. In alternating SGD, each epoch is divided into two sub-epochs where in each sub-epochs one factor matrix is fixed and the other is updated. This approach enables limiting the feature vector updates that use stale data to one of the two factor matrices during a sub-epoch. Recently, Shi et al. [98] proposed a distributed algorithm based on alternating SGD with data-aware partitioning.

5.8 Conclusion

We proposed a framework for scaling stratified SGD through significantly reducing the communication overhead. The framework targets at reducing the bandwidth overhead by efficiently finding the required communication during an SGD epoch, using P2P messages to perform it, and an HP-based method to further reduce the P2P communication volume. The framework targets at reducing the increase in latency overhead through the novel H&C strategy to limit the number of messages sent by a processor per epoch to $\mathcal{O}(K \lg K)$. Our proposed framework achieves scalable distributed SGD, on up to $K = 1024$ processors, without any compromise on convergence rate or any update on stale factors. The proposed framework achieves up to 15x runtime improvement over the state of the art DSGD method, on 1024 processors, using six real-world rating matrices.

Chapter 6

Conclusions

In this dissertation, we proposed several algorithms and partitioning models to enable the scalability of CPD-ALS for tensor decomposition as well as distributed SSGD for matrix factorization. The identified load balancing problems of MTTKRP detailed in **Chapter 3** paved the way for a better understanding of the dimensions of the load balancing problem for nonzero-based CPD-ALS. Furthermore, the proposed solutions, applied to a fine-grain HP model, can be extended to other nonzero-based partitioning models and methods that follow the CSF-oriented MTTKRP. The proposed embedding framework in **Chapter 4** enables the CPD-ALS to scale beyond 1K processors though decreasing the number of messages via the embedding scheme. The HP-based method proposed for the embedding algorithm correctly encapsulates a concurrent cost metric that mimics the communication behavior of the algorithm and leads to faster runtimes due to decreased volume compared to random partitioning. The algorithms proposed in **Chapter 5** for reducing the bandwidth overhead of SSGD enabled the algorithm to scale on large number of processors compared to state-of-the-art DSGD method. The H&C algorithm further enables the scalability through putting a nice $\mathcal{O}(K \log K)$ upper bound on the number of messages sent per processor. The experimental results detailed in each chapter on up to 4K processors on HPC systems proved the validity and importance of the proposed algorithms and models for the scalability of CPD-ALS and SSGD.

Bibliography

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

[2] Y.-X. Wang and Y.-J. Zhang, "Nonnegative matrix factorization: A comprehensive review," *IEEE Transactions on knowledge and data engineering*, vol. 25, no. 6, pp. 1336–1353, 2012.

[3] T. D. Nguyen, T. Tran, D. Phung, and S. Venkatesh, "Tensor-variate restricted boltzmann machines," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[4] E. Acar and B. Yener, "Unsupervised multiway data analysis: A literature survey," *IEEE transactions on knowledge and data engineering*, vol. 21, no. 1, pp. 6–20, 2008.

[5] S. Hosseinimotlagh and E. E. Papalexakis, "Unsupervised content-based identification of fake news articles with tensor decomposition ensembles," in *Proceedings of the Workshop on Misinformation and Misbehavior Mining on the Web (MIS2)*, 2018.

[6] S. Rabanser, O. Shchur, and S. Günnemann, "Introduction to tensor decompositions and their applications in machine learning," *arXiv preprint arXiv:1711.10781*, 2017.

[7] E. Acar, C. Aykut-Bingol, H. Bingol, R. Bro, and B. Yener, "Multiway analysis of epilepsy tensors," *Bioinformatics*, vol. 23, pp. i10–i18, 07 2007.

[8] I. Davidson, S. Gilpin, O. Carmichael, and P. Walker, "Network discovery via constrained tensor analysis of fMRI data," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, (New York, NY, USA), pp. 194–202, 2013.

[9] E. Acar and B. Yener, "Unsupervised multiway data analysis: A literature survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 1, pp. 6–20, 2009.

[10] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[11] K. R. Murphy, C. A. Stedmon, D. Graeber, and R. Bro, "Fluorescence spectroscopy and multi-way techniques. PARAFAC," *Analytical Methods*, vol. 5, no. 23, pp. 6557–6566, 2013.

[12] K. Maruhashi, F. Guo, and C. Faloutsos, "MultiAspectForensics: Pattern mining on large-scale heterogeneous networks with tensor analysis," in *2011 International Conference on Advances in Social Networks Analysis and Mining*, pp. 203–210, 2011.

[13] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, "Tensor decomposition for signal processing and machine learning," *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, 2017.

[14] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal link prediction using matrix and tensor factorizations," *ACM Trans. Knowl. Discov. Data*, vol. 5, Feb. 2011.

[15] H. Zhou, L. Li, and H. Zhu, "Tensor regression with applications in neuroimaging data analysis," *Journal of the American Statistical Association*, vol. 108, no. 502, pp. 540–552, 2013. PMID: 24791032.

[16] K. Makantasis, A. D. Doulamis, N. D. Doulamis, and A. Nikitakis, “Tensor-based classification models for hyperspectral data analysis,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 12, pp. 6884–6898, 2018.

[17] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, “Speeding-up convolutional neural networks using fine-tuned cp-decomposition,” *arXiv preprint arXiv:1412.6553*, 2014.

[18] Y. Wang, W. G. Guo, and X. Yue, “Tensor decomposition to compress convolutional layers in deep learning,” *IISE Transactions*, p. 1–60, Apr 2021.

[19] D. Song, P. Zhang, and F. Li, “Speeding up deep convolutional neural networks based on tucker-cp decomposition,” in *Proceedings of the 2020 5th International Conference on Machine Learning Technologies*, ICMLT 2020, (New York, NY, USA), p. 56–61, Association for Computing Machinery, 2020.

[20] Y. Ji, Q. Wang, X. Li, and J. Liu, “A survey on tensor techniques and applications in machine learning,” *IEEE Access*, vol. 7, pp. 162950–162990, 2019.

[21] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[22] 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,” WWW ’10, (New York, NY, USA), p. 681–690, Association for Computing Machinery, 2010.

[23] D. D. Lee and H. S. Seung, “Learning the parts of objects by non-negative matrix factorization,” *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[24] T. Hofmann, “Probabilistic latent semantic indexing,” in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 50–57, 1999.

[25] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[26] 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,” *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[27] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, “Large-scale matrix factorization with distributed stochastic gradient descent,” in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 69–77, 2011.

[28] O. Kaya and B. Uçar, “Scalable sparse tensor decompositions in distributed memory systems,” in *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2015.

[29] O. Kaya and B. Uçar, “Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees,” *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99–C130, 2018.

[30] S. Acer, T. Torun, and C. Aykanat, “Improving medium-grain partitioning for scalable sparse tensor decomposition,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 2814–2825, Dec 2018.

[31] M. O. Karsavuran, S. Acer, and C. Aykanat, “Partitioning models for general medium-grain parallel sparse tensor decomposition,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 147–159, 2021.

[32] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, “SPLATT: Efficient and parallel sparse tensor-matrix multiplication,” in *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 61–70, May 2015.

[33] S. Smith and G. Karypis, “Tensor-matrix products with a compressed sparse

tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, (New York, NY, USA), pp. 5:1–5:7, ACM, 2015.

[34] J. Li, J. Sun, and R. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 238–252, 2018.

[35] R. O. Selvitopi, M. M. Ozdal, and C. Aykanat, "A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 632–645, 2015.

[36] J. H. Choi and S. Vishwanathan, "DFacTo: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, pp. 1296–1304, 2014.

[37] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2015.

[38] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 902–911, May 2016.

[39] S. Acer, T. Torun, and C. Aykanat, "Improving medium-grain partitioning for scalable sparse tensor decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 2814–2825, Dec 2018.

[40] O. Kaya and B. Uçar, "Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99–C130, 2018.

[41] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 568–577, May 2018.

[42] M. Baskaran, T. Henretty, and J. Ezick, “Fast and scalable distributed tensor decompositions,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.

[43] L. Ma and E. Solomonik, “Efficient parallel cp decomposition with pairwise perturbation and multi-sweep dimension tree,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 412–421, 2021.

[44] M. O. Karsavuran, S. Acer, and C. Aykanat, “Partitioning models for general medium-grain parallel sparse tensor decomposition,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 147–159, 2021.

[45] N. Abubaker, S. Acer, and C. Aykanat, “True load balancing for matricized tensor times khatri-rao product,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1974–1986, 2021.

[46] C. Teflioudi, F. Makari, and R. Gemulla, “Distributed matrix completion,” in *2012 IEEE 12th International Conference on Data Mining*, pp. 655–664, 2012.

[47] 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*, vol. 7, no. 11, 2014.

[48] O. Fortmeier, H. M. BüCker, B. O. Fagginger Auer, and R. H. Bisseling, “A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications,” *Parallel Comput.*, vol. 39, p. 319–335, aug 2013.

[49] S. Acer, E. Kayaaslan, and C. Aykanat, “A recursive bipartitioning algorithm for permuting sparse square matrices into block diagonal form with overlap,” *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C99–C121, 2013.

[50] O. Selvitopi, S. Acer, and C. Aykanat, “A recursive hypergraph bipartitioning framework for reducing bandwidth and latency costs simultaneously,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 345–358, 2016.

[51] S. Acer, O. Selvitopi, and C. Aykanat, "Optimizing nonzero-based sparse matrix partitioning models via reducing latency," *Journal of Parallel and Distributed Computing*, vol. 122, pp. 145–158, 2018.

[52] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 673–693, July 1999.

[53] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 902–911, IEEE, 2016.

[54] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker, "PASTA: a parallel sparse tensor algorithm benchmark suite," *CCF Transactions on High Performance Computing*, vol. 1, no. 2, pp. 111–130, 2019.

[55] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[56] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM review*, vol. 47, no. 1, pp. 67–95, 2005.

[57] J. Shetty and J. Adibi, "The enron email dataset database schema and brief statistical report," *Information sciences institute technical report, University of Southern California*, vol. 4, 2004.

[58] S. Fernandes, H. Fanaee-T, and J. Gama, "Tensor decomposition for analysing time-evolving social networks: an overview," *Artificial Intelligence Review*, pp. 1–26, 2020.

[59] O. Görlitz, S. Sizov, and S. Staab, "PINTS: peer-to-peer infrastructure for tagging systems.," in *IPTPS*, p. 19, 2008.

[60] P. Bhargava, T. Phan, J. Zhou, and J. Lee, "Who, what, when, and where: Multi-dimensional collaborative recommendations using tensor factorization

on sparse user-generated data," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, (Republic and Canton of Geneva, CHE), p. 130?140, International World Wide Web Conferences Steering Committee, 2015.

[61] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 165–172, 2013.

[62] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning.," in *AAAI*, vol. 5, p. 3, 2010.

[63] 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*, pp. 316–324, ACM, 2012.

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

[65] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019.

[66] J. Li, Y. Ma, and R. Vuduc, "ParTI! : A parallel tensor infrastructure for multicore CPUs and GPUs," Oct 2018.

[67] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[68] U. Catalyurek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pp. 28–28, 2001.

[69] Ü. V. Çatalyürek and C. Aykanat, “A fine-grain hypergraph model for 2D decomposition of sparse matrices.,” in *IPDPS*, vol. 1, p. 118, 2001.

[70] B. Uçar and C. Aykanat, “Minimizing communication cost in fine-grain partitioning of sparse matrices,” in *International Symposium on Computer and Information Sciences*, pp. 926–933, Springer, 2003.

[71] D. M. Pelt and R. H. Bisseling, “A medium-grain method for fast 2D bipartitioning of sparse matrices,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 529–539, IEEE, 2014.

[72] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, “Collective communication: theory, practice, and experience,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[73] Z. Chi, H. Yan, and T. Pham, *Fuzzy algorithms: with applications to image processing and pattern recognition*, vol. 10. World Scientific, 1996.

[74] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, “Iterative algorithms for solution of large sparse systems of linear equations on hypercubes,” *IEEE Transactions on computers*, vol. 37, no. 12, pp. 1554–1568, 1988.

[75] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD, USA: Computer Science Press, 1978.

[76] Ü. V. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*, pp. 1479–1487. Boston, MA: Springer US, 2011.

[77] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, “FROSTT: The formidable repository of open sparse tensors and tools,” 2017.

[78] E. Cho, S. A. Myers, and J. Leskovec, “Friendship and mobility: user movement in location-based social networks,” in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1082–1090, 2011.

[79] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 165–172, 2013.

[80] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 69–79, March 1999.

[81] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), pp. 227–237, ACM, 2019.

[82] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, (New York, NY, USA), pp. 5:1–5:7, ACM, 2015.

[83] K. D. Devine and G. Ballard, "GentenMPI: Distributed memory sparse tensor decomposition.," August 2020.

[84] E. De Sturler and H. A. van der Vorst, "Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers," *Applied Numerical Mathematics*, vol. 18, no. 4, pp. 441–459, 1995.

[85] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a conjugate gradient solver with non-blocking collective operations," *Parallel Computing*, vol. 33, no. 9, pp. 624–633, 2007.

[86] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014.

[87] R. O. Selvitopi, M. M. Ozdal, and C. Aykanat, "A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 632–645, 2015.

[88] G. Karypis and V. Kumar, “Multilevel k-way hypergraph partitioning,” *VLSI design*, vol. 11, no. 3, pp. 285–300, 2000.

[89] R. He and J. McAuley, “Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering,” in *proceedings of the 25th international conference on world wide web*, pp. 507–517, 2016.

[90] M. Wan and J. McAuley, “Item recommendation on monotonic behavior chains,” in *Proceedings of the 12th ACM conference on recommender systems*, pp. 86–94, 2018.

[91] R. He, W.-C. Kang, and J. McAuley, “Translation-based recommendation,” in *Proceedings of the eleventh ACM conference on recommender systems*, pp. 161–169, 2017.

[92] R. Pasricha and J. McAuley, “Translation-based factorization machines for sequential recommendation,” in *Proceedings of the 12th ACM Conference on Recommender Systems*, pp. 63–71, 2018.

[93] J. Rappaz, J. McAuley, and K. Aberer, “Recommendation on live-streaming platforms: Dynamic availability and repeat consumption,” in *Fifteenth ACM Conference on Recommender Systems*, pp. 390–399, 2021.

[94] R. Guo, F. Zhang, L. Wang, W. Zhang, X. Lei, R. Ranjan, and A. Y. Zomaya, “Bapa: A novel approach of improving load balance in parallel matrix factorization for recommender systems,” *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 789–802, 2021.

[95] F. Petroni and L. Querzoni, “GASGD: Stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning.,” in *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys ’14, (New York, NY, USA), p. 241–248, Association for Computing Machinery, 2014.

[96] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, (USA), p. 17–30, USENIX Association, 2012.

[97] X. Luo, H. Liu, G. Gou, Y. Xia, and Q. Zhu, “A parallel matrix factorization based recommender by alternating stochastic gradient decent,” *Engineering Applications of Artificial Intelligence*, vol. 25, no. 7, pp. 1403–1412, 2012. Advanced issues in Artificial Intelligence and Pattern Recognition for Intelligent Surveillance System in Smart Home Environment.

[98] X. Shi, Q. He, X. Luo, Y. Bai, and M. Shang, “Large-scale and scalable latent factor analysis via distributed alternative stochastic gradient descent for recommender systems,” *IEEE Transactions on Big Data*, pp. 1–1, 2020.