Algorithms for Large-Scale Sparse Tensor Factorization

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Shaden Smith

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Dr. George Karypis, Advisor

April, 2019

Acknowledgements

No thesis is completed in isolation. This work is the culmination of tremendous mentorship and support from many people.

First and foremost, I would like to thank my advisor, George Karypis, for his guidance, expertise, and patience. George taught me to think critically, work hard, and to set the highest expectations for myself. I would not be the researcher that I am today without him.

I am grateful for Professors Arindam Banerjee, Daniel Boley, Abhishek Chandra, Yousef Saad, and Nikos Sidiropoulos for serving on my preliminary and final committees and for providing essential feedback throughout my degree progress.

I am deeply indebted to my wife, Sarah, for her unending support and countless sacrifices. Together, we celebrated every acceptance and mourned every rejection. Sarah is my strongest supporter, my greatest friend, and the joy of my life.

I am so grateful for my supportive and motivating family. I am a proud brother to amazing siblings Chyla, Bryland, and Noelle. In addition, Sarah's family has welcomed me as a son and brother.

I would like to thank the many members of Karypis Lab and the University of Minnesota that I am fortunate enough to call colleagues and friends: Agi, Ana, Ancy, Asmaa, David, Dom, Eva, Fan, Fateme, Haoji, Jake, Jeremy, Kejun, Kevin, Koorosh, Maria, Mohit, Rezwan, Santosh, Sara, Saurav, and Shalini. I am honored to have shared the past years alongside them and am a better person for it.

This work could not have been performed without access to the world-class staff and resources at the University of Minnesota. I would like to thank the Department of Computer Science and Engineering, Digital Technology Center, Minnesota Supercomputing Institute, and the Doctoral Dissertation Fellowship.

Lastly, I would like to thank our two wonderful cats: Tygra and Panthro. Research can be a lonely endeavor; I am thankful for the comfort, companionship, and critical feedback that they provided over many long nights.

Dedication

To my parents, for their unconditional love and support.

Abstract

Tensor factorization is a technique for analyzing data that features interactions of data along three or more axes, or *modes*. Many fields such as retail, health analytics, and cybersecurity utilize tensor factorization to gain useful insights and make better decisions. The tensors that arise in these domains are increasingly large, sparse, and high dimensional. Factoring these tensors is computationally expensive, if not infeasible. The ubiquity of multi-core processors and large-scale clusters motivates the development of scalable parallel algorithms to facilitate these computations. However, sparse tensor factorizations often achieve only a small fraction of potential performance due to challenges including data-dependent parallelism and memory accesses, high memory consumption, and frequent fine-grained synchronizations among compute cores.

This thesis presents a collection of algorithms for factoring sparse tensors on modern parallel architectures. This work is focused on developing algorithms that are scalable while being memory- and operation-efficient. We address a number of challenges across various forms of tensor factorizations and emphasize results on large, real-world datasets.

The canonical polyadic decomposition (CPD) is used extensively to analyze large and sparse tensors. We first propose operation-efficient algorithms for the sequence of tensor-matrix kernels that consume the majority of execution time. Topics such as parallelism, cache locality, and memory-efficiency are addressed through several optimizations and a data structure for sparse tensors. The preceding contributions are evaluated in both the serial and multi-core settings, and we demonstrate an average speedup over the state-of-the-art of over $5\times$.

High-performance computing systems are turning to many-core architectures that feature a large number of energy-efficient cores backed by high-bandwidth memory. These features are exemplified in Intel's Knights Landing many-core processor (KNL), which typically has 68 cores and 16GB of on-package multi-channel DRAM (MCDRAM). We investigate how the architectural features offered by KNL can be used in the context of computing the CPD. To achieve high performance, we (i) develop problem decompositions for the computations which are amenable to hundreds of concurrent threads while maintaining load balance and low synchronization costs; and (ii) explore

the utilization of architectural features such as MCDRAM. Using one KNL processor, our algorithm achieves up to 1.8× speedup over a dual socket Intel Xeon system with 44 cores.

We next address scalability challenges that arise on large-scale distributed-memory systems. Most methods have focused on distributing the tensor in a coarse-grained, one-dimensional fashion that prohibitively requires the factorization to be fully replicated on each node. Recent work overcomes this limitation by using a fine-grained decomposition of the tensor data, at the cost of computationally expensive pre-processing techniques and frequent small communications. To that effect, we introduce a medium-grained decomposition that avoids total replication and communication of the factorization, while eliminating the need for expensive pre-processing steps and sending fewer messages. We theoretically analyze the scalability of the decompositions and experimentally compare on up to 1024 compute cores. The medium-grained decomposition reduces communication volume by 36-90% compared to the coarse-grained decomposition and is 1.5-5.0× faster than the state-of-the-art decomposition with 1024 cores.

Imposing constraints on a factorization, such as non-negativity or sparsity, is a natural way of encoding prior knowledge of the multi-way data. While constrained factorizations are useful for practitioners, they can greatly increase factorization time due to slower convergence and computational overheads. We present a parallelization strategy and two approaches for accelerating constrained factorizations. By redefining the convergence criteria of the inner subproblems, we are able to split the data in a way that not only accelerates the per-iteration convergence, but also speeds up the execution of the computations due to efficient use of cache resources. Second, we develop a method of exploiting dynamic sparsity in the factors to speed up tensor-matrix kernels. These combined advancements achieve up to 8× speedup over the state-of-the-art on a variety of real-world sparse tensors.

In some applications, a tensor is sparse because of data that is *missing* instead of numerically zero. Tensor completion is a powerful tool used to estimate or recover missing values in multi-way data. We study three optimization algorithms that have been successfully applied to tensor completion: alternating least squares (ALS), stochastic gradient descent (SGD), and coordinate descent (CCD++). We explore opportunities for parallelism on shared- and distributed-memory systems and address challenges such

as memory- and operation-efficiency, load balance, cache locality, and communication. Furthermore, we show that introducing randomization during ALS and CCD++ can accelerate convergence. We evaluate our parallel formulations on a variety of real datasets on a modern supercomputer and demonstrate speedups through 16,384 cores. These improvements reduce time-to-solution from hours to seconds on real-world datasets.

Lastly, we turn to the Tucker decomposition, which is a higher-order analogue of the singular value decomposition. Computing the Tucker decomposition is more demanding than the CPD in terms of memory and computational resources. State-of-the-art algorithms accelerate the underlying computations by trading off memory to store intermediate results and reuse them across iterations. We present an algorithm based on a compressed data structure for sparse tensors and show that many computational redundancies during the computations can be identified and pruned without the memory overheads of competing approaches. In addition, the presented algorithm can further reduce the number of operations by exploiting an additional amount of user-specified memory. We evaluate our algorithm on a collection of real-world and synthetic datasets and demonstrate up to 20.7× speedup while using 28.5× less memory than the state-of-the-art parallel algorithm.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many domains rely on data with variables interacting in multiple ways. An electronic health record is an interaction between variables such as a patient, symptoms, diagnoses, medical procedures, and outcome. Similarly, how much a listener will like a song is an interaction between the listener, the song, and the setting (e.g., whether the listener is relaxing, studying, or dancing). These relationships can be modeled as a *tensor*, which is the generalization of a matrix to higher order. Tensor *factorization* is a technique similar to its matrix analogs that produces a lower-dimensional representation of the data. This form most often has lower rank than the original data, thus reducing complexity and noise in the original data while exposing relationships that can lead to insights. Numerous fields such as recommender systems [1], health analytics [2], cybersecurity [3], and social network analysis [4] utilize tensor factorization to gain useful insight and make better decisions.

Common traits among all of these applications are the high dimensionality and extreme level of sparsity of the data. Exploiting tensor sparsity is essential for developing efficient factorization algorithms. However, sparsity makes the underlying data unstructured, which requires a significant rethinking on how to perform the underlying computations. Unstructured applications often achieve only a small fraction of potential performance due to challenges including data-dependent parallelism and memory accesses, high memory consumption, and frequent fine-grained synchronizations among cores. Existing tools can take days or even weeks to factor large sparse tensors and can require memory that far exceeds what is available in today's computers.

The last decade has seen a shift in computer hardware from releasing faster, more powerful cores to placing a large number of low-powered cores in a single machine. Laptops with four or even eight cores are now ubiquitous and the latest generation of servers have over fifty cores. If we want to process increasingly large datasets, algorithm developers must embrace modern hardware and exploit highly parallel machines.

Thus, it is imperative to develop algorithms and software that utilize parallel systems to accelerate and enable the processing of massive sparse tensors. By developing novel methods and releasing them to the community, domain experts and fellow researchers can analyze large multi-way data and gain insights that were previously infeasible to obtain without high-performance computing (HPC).

1.1 Contributions

The contributions of this thesis are efficient and parallel algorithms for several forms of tensor factorization in the context of large, sparse datasets. Our advancements include scaling up the popular canonical polyadic decomposition (CPD) on shared- and distributed-memory systems, efficiently enforcing constraints on the CPD, large-scale tensor completion, and an asymptotic improvement to the Tucker decomposition. Common themes among these advancements are compressed data structures, operation- and memory-efficient algorithms, and increased parallelism.

The research presented in this thesis is culminated in SPLATT, an open source toolkit for large-scale sparse tensor factorization [5]. SPLATT is now actively used by researchers in academia, industry, and government.

1.1.1 Accelerating the CPD on Multi-Core Systems

The most expensive step when factoring large, sparse tensors is performing a sequence of tensor-matrix kernels that comprise each iteration of the optimization procedure. The kernels have several key challenges: (i) unstructured, data-dependent memory accesses, (ii) high computational cost, and (iii) intermediate results that can consume orders of magnitude more memory than the input or output of the kernel. Efficient approaches must simultaneously minimize the number of performed operations and memory accesses while constraining the amount of memory consumed.

In this thesis (Chapter 4), we present several optimizations for performing large-scale factorizations on multi-core systems. These optimizations include a data structure for sparse tensors that facilitates performing the key kernels in both a memory- and operation-efficient manner, methods of increasing the degree of available parallelism, and a method of cache-friendly reorderings and tilings. These optimizations achieve an average of almost 5× speedup over the state-of-the-art parallel approach on a 16-core system.

1.1.2 Accelerating the CPD on Many-Core Systems

The growing demands of data intensive applications have driven the adoption of *many-core processors*. These are highly parallel processors that feature several tens of cores with wide vector instructions. The large processing capabilities of many-core processors places increased pressure on memory bandwidth that DDR memory is unable to satisfy. A key recent trend of many-core processors is the inclusion of high-bandwidth memory, which has several times higher bandwidth than traditional DDR memory. This trend is exemplified in recent architectures such as NVIDIA Pascal, AMD Fiji, and Intel Knights Landing (KNL).

In this thesis (Chapter 5), we present an exploration of performance optimizations for sparse tensor factorization on many-core processors. We study the CPD and use KNL as a vehicle for exploration. We identify and address challenges that involve both the algorithms used for the CPD and the utilization of KNL's architectural features. Specifically, we address issues associated with extracting concurrency at different levels of granularity from unstructured and sparse data, load balancing computations in the presence of highly skewed data distributions, and understanding the different trade-offs between fine-grained synchronization and storage overheads.

1.1.3 Accelerating the CPD on Distributed-Memory Systems

Some applications require the lowest possible time to solution, or use datasets that are too large to process on a single machine. In order to solve such problems we turn to *supercomputers*, which are distributed systems that support millions of concurrent threads of execution and communicate over a high-speed network. The tradeoff for

increased computational and memory resources is a larger need for scalable parallel algorithms. Scalability can only be achieved if work is balanced among the machines and network communications do not outweigh the benefits of additional parallelism.

In this thesis (Chapter 6), we present for computing the CPD on distributed-memory systems. Our contributions include a way of distributing the input and output data that that co-optimizes bandwidth and latency by trading a small amount of additional communication volume for fewer exchanged messages. This *medium-grained decomposition* achieves 1.5× to 5× speedup over the state-of-the-art distributed approach on a 1024-core system.

1.1.4 Sparse Tensor Factorization with Constraints

Oftentimes, a domain expert wishes to encode some prior knowledge of the data in order to obtain a more interpretable factorization. Prior knowledge is typically incorporated by either forcing the solution to take some form (i.e., imposing a *constraint*), or penalizing unwanted solutions (i.e., adding a *regularization*). For example, imposing a non-negativity constraint on a factorization allows one to better model data whose values are additive. Similarly, adding a regularization term which encourages sparsity can help model data whose interactions are sparse. While valuable to practitioners, constrained and regularized factorizations change the underlying computations and can significantly increase the computational cost of factorization.

In this thesis (Chapter 7), we present a parallelization strategy and high performance implementation for computing the CPD with constraints. Our algorithm features two optimizations: (i) a blockwise reformulation of underlying optimization algorithm to improve convergence rate, parallelism, and cache efficiency; and (ii) a method of exploiting the sparsity which dynamically evolves in the factorization.

1.1.5 Large-Scale Tensor Completion

In many cases, a tensor is sparse due to values which are *missing* or *unknown* instead of numerically zero. Tensor completion is the problem of recovering or predicting the values in a sparse tensor and can be approached with tensor factorization. The resulting optimization formulation differs from traditional factorization problems and is addressed

with a variety of algorithms and underlying computations.

In this thesis (Chapter 8), we explore high performance tensor completion with three popular optimization algorithms. We address issues on shared- and distributed-memory systems such as memory and operation-efficient algorithms, cache locality, load balance, and communication. The resulting algorithms are demonstrated to scale to thousands of cores and outperform the state-of-the-art parallel methods by up to 150×.

1.1.6 Accelerating the Tucker Decomposition

The Tucker decomposition is another form of tensor factorization and can be viewed as the higher-order analogue of the singular value decomposition. Computing the Tucker decomposition of a sparse tensor is more demanding than the CPD in terms of both memory and computational resources. Existing approaches either have long runtimes or require larger amounts of intermediate memory than the combined problem input and output.

In this thesis (Chapter 9), we restructure the underlying computations in order to remove two forms of redundant computations that occur during the factorization procedure. We present an algorithm for performing the computations with a sparse tensor that is as computationally efficient as state-of-the-art algorithms, while requiring less additional intermediate memory. Additionally, this approach enables users to optionally trade off additional memory for computational savings.

1.2 Outline

The rest of this thesis is organized as follows. Chapter 2 establishes notation and provides a introduction to tensors and their factorizations. Chapter 3 details related work in the field of large-scale sparse tensor factorization. Chapter 4 focuses on shared-memory parallel systems and presents our work on restructuring and accelerating a kernel that is ubiquitous in many tensor factorizations. Chapter 5 explores performance optimizations for sparse tensor computations on emerging many-core processors. Chapter 6 presents our work on computing tensor factorizations using distributed-memory parallel systems. Chapter 7 focuses on efficiently incorporate constraints via a restructuring of the underlying optimization problem and data structures. Chapter 8 details

HPC formulations of several optimization algorithms for tensor completion. Chapter 9 presents our operation-efficient algorithms for the Tucker decomposition. Finally, we offer concluding remarks in Chapter 10.

1.3 Related Publications

The work presented in this thesis has been published in several leading conferences and journals in parallel computing. The related publications are as follows:

- **Shaden Smith**, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2015.

- **Shaden Smith** and George Karypis. Tensor-matrix products with a compressed sparse tensor. *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2015.

- **Shaden Smith**, Jongsoo Park, and George Karypis. Sparse tensor factorization on many-core processors with high-bandwidth memory. *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.

- **Shaden Smith** and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

- **Shaden Smith**, Alec Beri, and George Karypis. Constrained tensor factorization with accelerated AO-ADMM. *46th International Conference on Parallel Processing (ICPP)*, 2017.

- **Shaden Smith**, Jongsoo Park, and George Karypis. HPC formulations of optimization algorithms for tensor completion. *Parallel Computing*, (74), 2018.

- **Shaden Smith** and George Karypis. Accelerating the Tucker decomposition with compressed sparse tensors. *European Conference on Parallel Processing (Euro-Par)*, 2017.

Chapter 2

Background & Notation

We first provide a background on tensors, the notations used in this document, and tensor factorization. For further background on tensors and tensor factorization, we direct the reader to several essential surveys [6, 7].

2.1 Notation

Tensors are the generalization of matrices to higher dimensions, or *modes*. We denote vectors using bold lowercase letters ($\boldsymbol{\lambda}$), matrices using bold capital letters ($\mathbf{A}$), and tensors using bold capital calligraphic letters ($\boldsymbol{\mathcal{X}}$).

We will limit discussion to three-mode tensors when possible for clarity and notational convenience. When discussing tensors of general order, we denote the number of modes as N. A third-order tensor is assumed to have dimensions $I \times J \times K$, and an nth-order tensor has dimensions $I_1 \times I_2 \times \ldots \times I_N$. We use $\text{nnz}(\boldsymbol{\mathcal{X}})$ to denote the number of non-zero elements in $\boldsymbol{\mathcal{X}}$. A tensor entry with coordinate (i, j, k) is denoted $\boldsymbol{\mathcal{X}}(i, j, k)$, and matrix entries are similarly denoted $\mathbf{A}(i, j)$. A colon in the place of an index represents all entries of that mode. For example, $\mathbf{A}(:, f)$ is the fth column of the matrix $\mathbf{A}$. *Fibers* are the generalization of matrix rows and columns and are the result of holding two indices constant (e.g., $\boldsymbol{\mathcal{X}}(i, j, :)$ or $\boldsymbol{\mathcal{X}}(i, :, k)$). A *slice* of a tensor is the result of holding one index constant and the result is a matrix (e.g., $\boldsymbol{\mathcal{X}}(i, :, :)$).

A tensor can be unfolded, or *matricized*, into a matrix along any of its modes. In the mode-n matricization, the mode-n fibers form the columns of the resulting matrix.

$$\boldsymbol{\mathcal{X}}(:,:,1) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \boldsymbol{\mathcal{X}}(:,:,2) = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(1)} = \begin{bmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(2)} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

$$\mathbf{X}_{(3)} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \\ 7 & 10 & 8 & 11 & 9 & 12 \end{bmatrix}$$

Figure 2.1: The matricizations of an (2×3×2) tensor.

The mode-n unfolding of $\boldsymbol{\mathcal{X}}$ is denoted as $\mathbf{X}_{(n)}$. If $\boldsymbol{\mathcal{X}}$ has dimension $I \times J \times K$, then $\mathbf{X}_{(1)}$ is $I \times JK$. Figure 2.1 illustrates the unfoldings of a third-order tensor.

2.2 Common Matrix and Tensor Operations

Three essential matrix operations are the *Hadamard product*, the *Kronecker product*, and the *Khatri-Rao product*. The Hadamard product, denoted $\mathbf{A} * \mathbf{B}$, is the element-wise multiplication of $\mathbf{A}$ and $\mathbf{B}$. The element (m, n) of $\mathbf{A} * \mathbf{B}$ is $\mathbf{A}(m, n)\mathbf{B}(m, n)$. $\mathbf{A}$ and $\mathbf{B}$ must match in dimension for the Hadamard product to exist. The Kronecker product, denoted $\mathbf{A} \otimes \mathbf{B}$, is a generalization of the vector outer product to matrices and is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}(1,1)\mathbf{B} & \dots & \mathbf{A}(1,F)\mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}(I,1)\mathbf{B} & \dots & \mathbf{A}(I,F)\mathbf{B} \end{bmatrix}.$$

If $\mathbf{A}$ is $I \times F$ and $\mathbf{B}$ is $J \times F$, then $\mathbf{A} \otimes \mathbf{B}$ is $IJ \times F^2$. The Khatri-Rao product, denoted $\mathbf{A} \odot \mathbf{B}$, is defined in terms of the Kronecker product

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

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

$$\mathbf{C} \odot \mathbf{B} = \begin{bmatrix} c_{11}b_{11} & c_{12}b_{12} \\ c_{11}b_{21} & c_{12}b_{22} \\ c_{21}b_{11} & c_{22}b_{12} \\ c_{21}b_{21} & c_{22}b_{22} \\ c_{31}b_{11} & c_{32}b_{12} \\ c_{31}b_{21} & c_{32}b_{22} \end{bmatrix}$$

Figure 2.2: The Khatri-Rao product of two matrices.

$\mathbf{A}$ and $\mathbf{B}$ must have matching column dimension for their Khatri-Rao product to be defined. If $\mathbf{A}$ is $I \times F$ and $\mathbf{B}$ is $J \times F$, then $\mathbf{A} \odot \mathbf{B}$ is $IJ \times F$. Figure 2.2 illustrates the Khatri-Rao product.

Matricized Tensor Times Khatri-Rao Product A common kernel in tensor factorizations is the *matricized tensor times Khatri-Rao Product* (MTTKRP), denoted $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. If $\mathbf{B}$ and $\mathbf{C}$ each have F columns, then the result of MTTKRP is an $I \times F$ matrix that is typically dense. When $\boldsymbol{\mathcal{X}}$ is large and sparse, the Khatri-Rao product $(\mathbf{C} \odot \mathbf{B})$ cannot be constructed in practice due to it being a dense $JK \times F$ matrix, which can be orders of magnitude larger than $\boldsymbol{\mathcal{X}}$. Numerous approaches have been developed to avoid this *intermediate data explosion* and are discussed in Section 3.1.1.

Tensor-Matrix Product. The tensor-matrix product, or *n-mode product* [6], multiplies a tensor by a matrix along the nth mode. Suppose $\mathbf{M}$ is an $F \times I_n$ matrix. The tensor-matrix product for the nth mode, denoted $\boldsymbol{\mathcal{X}} \times_n \mathbf{M}$, emits a tensor with dimensions $I_1 \times \ldots \times I_{n-1} \times F \times I_{n+1} \times \ldots \times I_N$. Elementwise,

$$(\boldsymbol{\mathcal{X}} \times_n \mathbf{M})(i_1, \ldots, i_{n-1}, f, i_{n+1}, \ldots, i_N) = \sum_{i_n=1}^{I_n} \boldsymbol{\mathcal{X}}(i_1, \ldots, i_N)\mathbf{M}(f, i_n).$$

Note that the resulting mode-n fibers are generally dense regardless of the sparsity pattern of $\boldsymbol{\mathcal{X}}$.

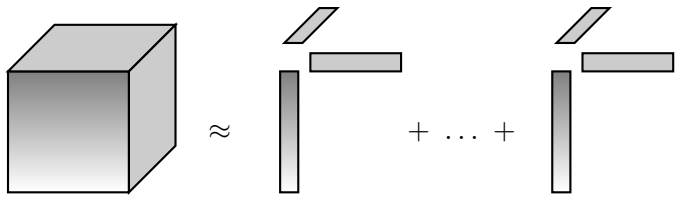


Figure 2.3: The CPD as the summation of outer products.

2.3 Canonical Polyadic Decomposition

The canonical polyadic decomposition (CPD) is the most widely used tensor factorization, especially in the case of large, sparse tensors [6]. The rank-F CPD decomposes a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ into factors $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$. In our applications of interest, we are most often interested in a *low-rank* factorization, in which F is a small constant on the order of 10 or 100.

Shown in Figure 2.3, the CPD models a tensor as the summation of F outer products:

$$\boldsymbol{\mathcal{X}} \approx \sum_{f=1}^{F} \mathbf{A}(i,:) \circ \mathbf{B}(j,:) \circ \mathbf{C}(k,:).$$

The CPD can also be formulated elementwise using multi-way inner products:

$$\boldsymbol{\mathcal{X}}(i,j,k) \approx \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f). \tag{2.1}$$

The CPD is formulated as the following optimization problem:

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \quad \mathcal{L}(\boldsymbol{\mathcal{X}}; \{\mathbf{A}, \mathbf{B}, \mathbf{C}\})$$

where $\mathcal{L}(\cdot)$ is some suitable loss function. Two common loss functions are the traditional sum-of-squares loss:

$$\mathcal{L}(\boldsymbol{\mathcal{X}}; \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}) = \frac{1}{2} \left\| \boldsymbol{\mathcal{X}} - \sum_{f=1}^{F} \mathbf{A}(:,f) \circ \mathbf{B}(:,f) \circ \mathbf{C}(:,f) \right\|_F^2, \tag{2.2}$$

and one that treats sparsity as *missing* instead of numerical zeros:

$$\mathcal{L}(\boldsymbol{\mathcal{X}}; \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}) = \frac{1}{2} \sum_{(i,j,k) \in \Omega} \left(\boldsymbol{\mathcal{X}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \right)^2, \tag{2.3}$$

where Ω is the set of *observed* entries in $\boldsymbol{\mathcal{X}}$. Note that Equation (2.3) is only defined over the observed entries of $\boldsymbol{\mathcal{X}}$ and is derived from the element-wise formulation of the CPD in Equation (2.1). As a result, it can be used to predict or recover the missing values in the tensor as opposed to modeling numerical zeros, as Equation (2.2) is designed to do. Optimizing with loss function Equation (2.3) is a task often referred to as *tensor completion*.

2.3.1 Computing the CPD with Alternating Least Squares

Computing the least-squares CPD is a non-convex optimization problem most often approximated with alternating optimization (AO) approaches. AO approaches simplify the computation by holding all but one factor matrix constant to reach a convex formulation. The factor matrices are cyclicly updated in this manner. The most common AO approach is *alternating least squares* (CPD-ALS) [6]. During each iteration, $\mathbf{B}$ and $\mathbf{C}$ are fixed and we solve the unconstrained quadratic optimization problem

$$\underset{\mathbf{A}}{\text{minimize}} \quad \frac{1}{2} \left\| \mathbf{X}_{(1)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\intercal \right\|_F^2$$

with solution

$$\mathbf{A}^\intercal = (\mathbf{C}^\intercal \mathbf{C} * \mathbf{B}^\intercal \mathbf{B})^{-1} \left(\mathbf{X}_{(1)} (\mathbf{C} \odot \mathbf{B}) \right)^\intercal .$$

An important consequence of using ALS is that computations are *mode-centric*, meaning each of the kernels will update only one mode of the factorization. Therefore, when discussing algorithmic details we will sometimes only refer to the first mode and imply that the other modes proceed similarly by appealing to symmetry.

Algorithm 1 details the CPD-ALS algorithm. There are three primary computations required to update a factor matrix: (i) construction of the $F \times F$ normal equations, (ii) MTTKRP, and (iii) solving for the new factor matrix. Constructing the normal equations involves forming $N-1$ Gram matrices of size $F \times F$. While each Gram computation requires $\mathcal{O}(IF^2)$ operations, they can be cached and only require recomputation after the factor is updated. MTTKRP can be performed in $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ time and is the most computationally expensive step of CPD-ALS. The output of MTTKRP is a matrix matching the dimensionality of the factor matrix being updated (e.g., $I \times F$). Lastly, solving for the new factor matrix is implemented with a Cholesky factorization

Algorithm 1 CPD-ALS

1: **while** not converged **do**
2: $\quad \mathbf{A}^\intercal \leftarrow (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})^{-1} \left(\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})\right)^\intercal$
3: $\quad \mathbf{B}^\intercal \leftarrow (\mathbf{C}^\intercal\mathbf{C} * \mathbf{A}^\intercal\mathbf{A})^{-1} \left(\mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})\right)^\intercal$
4: $\quad \mathbf{C}^\intercal \leftarrow (\mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A})^{-1} \left(\mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})\right)^\intercal$
5: **end while**

in $\mathcal{O}(F^3)$ time and forward/backward substitutions taking $\mathcal{O}(IF^2)$ time. Steps (i) and (iii) are inexpensive relative to MTTKRP and also can be efficiently implemented using well-studied dense linear algebra packages such as LAPACK [8]. Thus, most research on large-scale sparse tensor factorization focuses on the MTTKRP computation.

CPD-ALS iterates until convergence. The residual of a tensor $\boldsymbol{\mathcal{X}}$ and its CPD approximation $\boldsymbol{\mathcal{Z}}$ is

$$\sqrt{\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}} \rangle + \langle \boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Z}} \rangle - 2\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle},$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product. The expression $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}} \rangle$ is equivalent to $||\boldsymbol{\mathcal{X}}||_F^2$, and is a direct extension of the matrix Frobenius norm, i.e., the sum-of-squares of all non-zero elements. $\boldsymbol{\mathcal{X}}$ is also a constant input and thus its norm can be pre-computed. The norm of a factored tensor is

$$||\boldsymbol{\mathcal{Z}}||_F^2 = \mathbf{1}^\intercal \left(\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B} * \mathbf{A}^\intercal\mathbf{A}\right) \mathbf{1}.$$

Fortunately, each $\mathbf{A}^\intercal\mathbf{A}$ product is computed during the CPD-ALS iteration and the results can be cached and reused in just $\mathcal{O}(F^2)$ space. The complexity of computing the residual is dominated by the inner product $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle$ which is given by

$$\sum_{f=1}^{F} \left(\sum_{(i,j,k)\in\Omega} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \right). \tag{2.4}$$

The cost of Equation (2.4) is $4F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ floating-point operations (FLOPs), which is as expensive than an MTTKRP operation. In Section 6.1.3 we present a method of reusing MTTKRP operation results to reduce the cost to $2FI$.

2.3.2 Incorporating Constraints with AO-ADMM

Domain-specific knowledge is often used to guide a tensor factorization in order to arrive at a more interpretable or insightful result. For example, when working with physical phenomena (e.g., weather simulations or medical diagnoses), negative values do not have real-world meanings. Similarly, a doctor may already know that some procedures and diagnoses are related, and by guiding the tensor factorization to include that prior knowledge we can discover new relationships that complement those that were previously known. Domain-specific knowledge is encoded in a tensor factorization in the form of *constraints* on the solution. For example, a non-negativity constraint forces the solution to only include non-negative values.

We can modify Equation (2.2) to encode constraints:

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \quad \frac{1}{2}\left\|\boldsymbol{\mathcal{X}} - \sum_{f=1}^{F} \mathbf{A}(:,f) \circ \mathbf{B}(:,f) \circ \mathbf{C}(:,f)\right\|_F^2 + r(\mathbf{A}) + r(\mathbf{B}) + r(\mathbf{C}), \tag{2.5}$$

where $r(\cdot)$ is a penalty function. Constraints can be implemented by having $r(\cdot)$ take the value of infinity when the constraint is violated, and regularizations use finite values to penalize unwanted (but valid) solutions. For example, a non-negativity constraint uses the indicator function of $\mathbb{R}_+$ and sparsity-inducing regularization uses $||\cdot||_1$. Due to the similar nature of constraints and regularizations, we will use the terms interchangeably.

When no constraints are enforced, Equation (2.5) can simply be solved with CPD-ALS. AO-ADMM [9, 10] combines the AO framework with the alternating direction method of multipliers (ADMM), which is a popular framework for constrained optimization problems [11]. AO-ADMM inherits positive qualities from each: a monotonically decreasing objective function from AO and the ability to flexibly incorporate constraints from ADMM.

AO-ADMM proceeds with a sequence of *outer* and *inner* iterations. Each step of an outer iteration optimizes one of the matrix factors by means of ADMM. Internally, ADMM executes a sequence of inner iterations to enforce constraints. Since the ADMM algorithm is the same across all of the tensor modes, we will simplify the discussion and only consider the computations associated with the first mode.

Algorithm 2 ADMM to solve Equation (2.6)

1: **Input:** $\mathbf{H}$, $\mathbf{U}$, $\mathbf{K}$, $\mathbf{G}$
2: **Output:** $\mathbf{H}$, $\mathbf{U}$
3: $\rho \leftarrow \text{trace}(\mathbf{G})/F$
4: $\mathbf{L} \leftarrow \text{Cholesky}(\mathbf{G} + \rho\mathbf{I})$
5: **repeat** ▷ Inner iterations
6: $\quad \widetilde{\mathbf{H}} \leftarrow \mathbf{L}^{-T}\mathbf{L}^{-1}\left(\mathbf{K} + \rho(\mathbf{H} + \mathbf{U})\right)^T$
7: $\quad \mathbf{H}_0 \leftarrow \mathbf{H}$
8: $\quad \mathbf{H} \leftarrow \text{argmin}_{\mathbf{H}}\, r(\mathbf{H}) + \frac{\rho}{2}||\mathbf{H} - \widetilde{\mathbf{H}}^T + \mathbf{U}||_F^2$
9: $\quad \mathbf{U} \leftarrow \mathbf{U} + \mathbf{H} - \widetilde{\mathbf{H}}^T$
10: $\quad r \leftarrow ||\mathbf{H} - \widetilde{\mathbf{H}}^T||_F^2 / ||\mathbf{H}||_F^2$
11: $\quad s \leftarrow ||\mathbf{H} - \mathbf{H}_0||_F^2 / ||\mathbf{U}||_F^2$
12: **until** $r < \epsilon$ and $s < \epsilon$

When discussing ADMM, we refer to the primal and dual variables as $\mathbf{H} \in \mathbb{R}^{I \times F}$ and $\mathbf{U} \in \mathbb{R}^{I \times F}$, respectively. An auxiliary variable $\widetilde{\mathbf{H}} \in \mathbb{R}^{F \times I}$ is introduced to arrive at a constrained optimization problem in the form of ADMM:

$$\begin{aligned} \underset{\mathbf{H}, \widetilde{\mathbf{H}}}{\text{minimize}} \quad & \frac{1}{2}\left\|\mathbf{X}_{(1)} - \widetilde{\mathbf{H}}^T(\mathbf{C} \odot \mathbf{B})^T\right\|_F^2 + r(\mathbf{H}) \\ \text{subject to} \quad & \mathbf{H} = \widetilde{\mathbf{H}}^T. \end{aligned} \tag{2.6}$$

Algorithm 2 details the resulting ADMM algorithm. It accepts as input the primal and dual variables and two additional matrices: (i) $\mathbf{K} \in \mathbb{R}^{I \times F}$, the output of the MTTKRP operation; and (ii) $\mathbf{G} \in \mathbb{R}^{F \times F}$, the Gram matrix which is formed via $\mathbf{G} = (\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})$. Line 6 executes forward- and backward-substitution on an $F \times I$ matrix in $\mathcal{O}(F^2 I)$ time. Line 8 is the *proximity operator* and varies based on $r(\cdot)$. For example, non-negativity constraints project to the non-negative orthant (i.e., re-assign negative entries to be zero). Line 9 updates the dual variable and lastly, lines 10 and 11 compute the relative primal and dual residuals.

Finally, the complete AO-ADMM framework is detailed in Algorithm 3. The factors are cyclically updated using Algorithm 2. Each MTTKRP operation requires $\mathcal{O}(F\,\text{nnz}(\boldsymbol{\mathcal{X}}))$ operations and is often the most expensive step of the AO-ADMM framework.

Algorithm 3 AO-ADMM

Initialize primal variables $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ randomly.
Initialize dual variables $\hat{\mathbf{A}}$, $\hat{\mathbf{B}}$, and $\hat{\mathbf{C}}$ with $\mathbf{0}$.
while not converged **do** ▷ Outer iterations
    $\mathbf{G} \leftarrow \mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C}$
    $\mathbf{K} \leftarrow \mathbf{X}_{(1)} (\mathbf{C} \odot \mathbf{B})$ ▷ MTTKRP
    $\mathbf{A}, \hat{\mathbf{A}} \leftarrow \text{ADMM}(\mathbf{A}, \hat{\mathbf{A}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 2

    $\mathbf{G} \leftarrow \mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C}$
    $\mathbf{K} \leftarrow \mathbf{X}_{(2)} (\mathbf{C} \odot \mathbf{A})$ ▷ MTTKRP
    $\mathbf{B}, \hat{\mathbf{B}} \leftarrow \text{ADMM}(\mathbf{B}, \hat{\mathbf{B}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 2

    $\mathbf{G} \leftarrow \mathbf{A}^T\mathbf{A} * \mathbf{B}^T\mathbf{B}$
    $\mathbf{K} \leftarrow \mathbf{X}_{(3)} (\mathbf{B} \odot \mathbf{A})$ ▷ MTTKRP
    $\mathbf{C}, \hat{\mathbf{C}} \leftarrow \text{ADMM}(\mathbf{C}, \hat{\mathbf{C}}, \mathbf{K}, \mathbf{G})$ ▷ Algorithm 2
end while

2.3.3 Optimization Algorithms for Tensor Completion

In the case of tensor completion, Equation (2.3) is most commonly augmented with a regularization on the factor matrices to prevent overfitting. We arrive at the following objective function:

$$\begin{aligned} \underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \quad & \frac{1}{2} \sum_{(i,j,k)\in\Omega} \left(\boldsymbol{\mathcal{X}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \right)^2 \\ & + \frac{\lambda}{2} \left(||\mathbf{A}||_F^2 + ||\mathbf{B}||_F^2 + ||\mathbf{C}||_F^2 \right), \end{aligned} \tag{2.7}$$

where λ is a parameter to scale regularization. The non-convexity of Equation (2.7) has inspired a substantial amount of research on optimization algorithms that effectively minimize the objective while being operation- and memory-efficient enough to be used in practice. While the traditional CPD is overwhelmingly computed using CPD-ALS, many optimization algorithms are actively used for tensor completion. Three optimization algorithms have seen particular success due to their efficiency, opportunities for

Table 2.1: Summary of optimization algorithms for tensor completion.

Algorithm	**Complexity**	**Storage**	**Traversals**	**Ref.**
ALS	$\mathcal{O}(N(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{X}}) + IF^3))$	$\mathcal{O}(F^2)$	N	[12, 13]
SGD	$\mathcal{O}(NF \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$	$\mathcal{O}(F)$	1	[12, 14]
CCD++	$\mathcal{O}(NF \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$	$\mathcal{O}(I)$	NF	[14, 13]

Complexity is the number of floating-point operations performed in one epoch. **Storage** is the amount of memory required to perform the factorization, excluding matrix and tensor storage. **Traversals** is the number of times the sparsity structure must be traversed in one epoch, excluding checks for convergence. **Ref.** provides references for the *tensor* variant of the algorithm. F is the rank of the factorization. N is the number of modes in the tensor. I is the length of the longest mode.

parallelism, and fast convergence. These methods are summarized in Table 2.1 and described below.

We follow the convention of the matrix completion community and refer to an *epoch* as the work performed to update $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ one time using the training data. We avoid the term *iteration* in order to emphasize the varying amounts of work performed and progress made.

For convenience, we define the loss function $\mathcal{L}'(i,j,k)$ to be the loss local to a given entry:

$$\mathcal{L}'(i,j,k) = \boldsymbol{\mathcal{X}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f).$$

Alternating Least Squares. Like the least-squares CPD, ALS can also used in the tensor completion setting. However, instead of each factor matrix having an update, each *row* is updated independently in the tensor completion setting. Illustrated in Figure 2.4, computing $\mathbf{A}(i,:)$ accesses all entries in $\boldsymbol{\mathcal{X}}(i,:,:)$ and also the rows $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ for each entry $\boldsymbol{\mathcal{X}}(i,j,k)$. The rows of $\mathbf{B}$ and $\mathbf{C}$ are used to compute $\mathbf{H}_i$, a $|\boldsymbol{\mathcal{X}}(i,:,:)| \times F$ matrix. If the lth entry in $\boldsymbol{\mathcal{X}}(i,:,:)$ has coordinate (i,j,k), then $\mathbf{H}_i(l,:) = [\mathbf{B}(j,:) * \mathbf{C}(k,:)]$. Given $\mathbf{H}_i$, we can compute $\mathbf{A}(i,:)$ via

$$\mathbf{A}(i,:) \leftarrow \left(\mathbf{H}_i^T\mathbf{H}_i + \lambda\mathbf{I}\right)^{-1} \mathbf{H}_i^T \operatorname{vec}(\boldsymbol{\mathcal{X}}(i,:,:)), \tag{2.8}$$

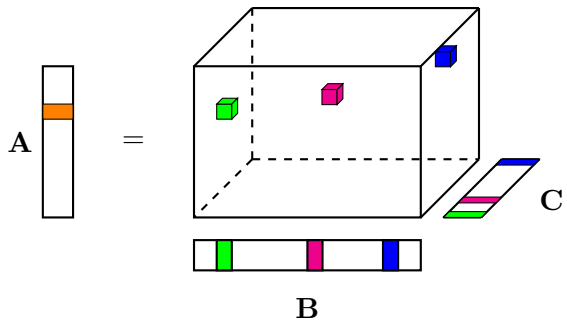


Figure 2.4: Memory accesses during ALS for tensor completion. Computing $\mathbf{A}(i,:)$ (in orange) requires $\boldsymbol{\mathcal{X}}(i,:,:)$ and the corresponding rows of $\mathbf{B}$ and $\mathbf{C}$.

where vec$(\cdot)$ rearranges the entries of its argument into a dense vector. The matrix $\left(\mathbf{H}_i^T\mathbf{H}_i + \lambda\mathbf{I}\right)$ is symmetric positive-definite and so the inversion is accomplished via a Cholesky factorization and forward/backward substitutions. ALS requires $\mathcal{O}(F^2\,\text{nnz}(\boldsymbol{\mathcal{X}}))$ operations to form all of the $\mathbf{H}_i$, and $\mathcal{O}(F^3)$ operations per row for the matrix inversions. In total, $\mathcal{O}(F^2\,\text{nnz}(\boldsymbol{\mathcal{X}}) + IF^3)$ operations are performed to update a factor. After computing all rows of $\mathbf{A}$, the other factors are computed in the same manner.

Stochastic Gradient Descent. The strategy of stochastic gradient descent (SGD) is to take many small steps per epoch, each based on the gradient at a single observation. At each step, SGD selects one entry at random and updates the factorization based on the gradient at $\boldsymbol{\mathcal{X}}(i,j,k)$. Updates are of the form

$$\begin{aligned}
\mathbf{A}(i,:) &\leftarrow \mathbf{A}(i,:) + \eta\left[\mathcal{L}'(i,j,k)\left(\mathbf{B}(j,:) * \mathbf{C}(k,:)\right) - \lambda\mathbf{A}(i,:)\right], \\
\mathbf{B}(j,:) &\leftarrow \mathbf{B}(j,:) + \eta\left[\mathcal{L}'(i,j,k)\left(\mathbf{A}(i,:) * \mathbf{C}(k,:)\right) - \lambda\mathbf{B}(j,:)\right], \\
\mathbf{C}(k,:) &\leftarrow \mathbf{C}(k,:) + \eta\left[\mathcal{L}'(i,j,k)\left(\mathbf{A}(i,:) * \mathbf{B}(j,:)\right) - \lambda\mathbf{C}(k,:)\right],
\end{aligned}$$

where η is a step size parameter. Each update requires $\mathcal{O}(F)$ operations, resulting in a complexity of $\mathcal{O}(F\,\text{nnz}(\boldsymbol{\mathcal{X}}))$ per epoch.

Coordinate Descent. In contrast to ALS and SGD which update entire factor rows at a time, coordinate descent methods optimize only one variable at a time. CCD++ is a coordinate descent method originally developed for matrix factorization [15] and later

extended to tensors [14, 13]. CCD++ updates columns of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in sequence, in effect optimizing the rank-one components of the factorization. Updates take the form:

$$\mathbf{A}(i,f) \leftarrow \frac{\alpha_i}{\lambda + \beta_i}, \tag{2.9}$$

where

$$\alpha_i = \sum_{\boldsymbol{\mathcal{X}}(i,:,:)} \mathcal{L}'(i,j,k)\mathbf{B}(j,f)\mathbf{C}(k,f),$$

and

$$\beta_i = \sum_{\boldsymbol{\mathcal{X}}(i,:,:)} \left(\mathbf{B}(j,f)\mathbf{C}(k,f)\right)^2.$$

After updating $\mathbf{A}(:,f)$, the columns $\mathbf{B}(:,f)$ and $\mathbf{C}(:,f)$ are updated similarly. An important optimization for CCD++ is to compute $\mathcal{L}(\cdot)$ only once each epoch and reuse it for each of the F columns [15]. This can be accomplished without additional storage by directly updating $\boldsymbol{\mathcal{X}}$ each iteration with the current residual. The resulting complexity is $\mathcal{O}(F(\text{nnz}(\boldsymbol{\mathcal{X}})+I+J+K))$, which for most datasets is $\mathcal{O}(F\,\text{nnz}(\boldsymbol{\mathcal{X}}))$, matching SGD.

2.4 Tucker Decomposition

The Tucker decomposition is another form of tensor factorization used in applications such as anomaly detection [3], healthcare [2], recommender systems [1], and web search [16]. Illustrated in Figure 2.5, the objective of the Tucker decomposition is to model a tensor $\boldsymbol{\mathcal{X}}$ with a set of factor orthonormal matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ that respectively have F_1, F_2, and F_3 columns, and a core tensor, $\boldsymbol{\mathcal{G}}$, of size $F_1 \times F_2 \times F_3$.

2.4.1 Higher-Order Orthogonal Iterations

Like the CPD, computing the Tucker decomposition is a non-convex optimization problem:

$$\begin{aligned} \underset{\mathbf{A},\mathbf{B},\mathbf{C},\boldsymbol{\mathcal{G}}}{\text{minimize}} \quad & \frac{1}{2}\left\|\mathbf{X}_{(1)} - \mathbf{A}\mathbf{G}_{(1)}\left(\mathbf{C}\otimes\mathbf{B}\right)^\intercal\right\|_F^2, \\ \text{subject to} \quad & \mathbf{A}^\intercal\mathbf{A} = \mathbf{I},\ \mathbf{B}^\intercal\mathbf{B} = \mathbf{I},\ \mathbf{C}^\intercal\mathbf{C} = \mathbf{I}. \end{aligned}$$

Several optimization algorithms have been developed to compute the Tucker decomposition, including the higher-order SVD (HOSVD) [17] and higher-order orthogonal iterations (HOOI) [18]. HOSVD is popular for decomposing *dense* tensors and efficient

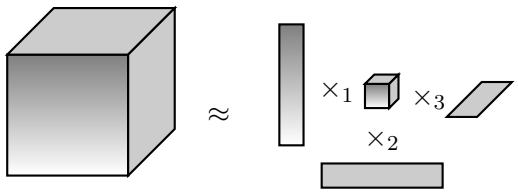


Figure 2.5: A rank-$\{F_1, F_2, F_3\}$ Tucker factorization of an $I \times J \times K$ tensor.

Algorithm 4 Tucker Decomposition with HOOI

1: **while** $\mathcal{G}$ not converged **do**
2: $\quad \mathcal{Y} \leftarrow \mathcal{X} \times_2 \mathbf{B}^\intercal \times_3 \mathbf{C}^\intercal$
3: $\quad \mathbf{A} \leftarrow F_1$ leading left singular vectors of $\mathbf{Y}_{(1)}$
4: $\quad \mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^\intercal \times_3 \mathbf{C}^\intercal$
5: $\quad \mathbf{B} \leftarrow F_2$ leading left singular vectors of $\mathbf{Y}_{(2)}$
6: $\quad \mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^\intercal \times_2 \mathbf{B}^\intercal$
7: $\quad \mathbf{C} \leftarrow F_3$ leading left singular vectors of $\mathbf{Y}_{(3)}$
8:
9: $\quad \mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{A}^\intercal \times_2 \mathbf{B}^\intercal \times_3 \mathbf{C}^\intercal$
10: **end while**

parallel algorithms have been developed [19, 20]. However, the computation becomes progressively more dense during HOSVD and it is not often applied to sparse computations. Thus, HOOI is the most popular algorithm for sparse tensors.

HOOI follows an alternating approach and cyclically updates each factor matrix until convergence. Algorithm 4 details the steps in computing the factor matrices and core tensor using HOOI. At its core is a tensor-times-matrix chain (TTMc), which performs a sequence of tensor-matrix products along all but one of the tensor modes (see Section 2.2). The result is flattened to a matrix and the leading left singular vectors are used to form the next factor matrix.

Chapter 3

Related Work

In this chapter, we overview work in the area of computing sparse tensor factorizations. We focus on parallel algorithms for the various kernels that arise while computing the CPD and Tucker models.

3.1 Computing the CPD

The key kernel to accelerate when computing the CPD is the MTTKRP operation, which can consume over 95% of the total execution time. The remaining computations consist of dense linear algebra, which can be efficiently handled using libraries such as BLAS.

3.1.1 CPD on Shared-Memory Systems

Over the years, a number of approaches have been developed for performing the MTTKRP. The most efficient of these methods operate in $\mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{X}}))$ time, but differ in memory consumption, cache utilization, and opportunities for parallelism.

Elementwise Formulation

The MTTKRP can be computed in place by observing the tensor in its non-matricized form. When updating the first factor matrix, each non-zero contributes:

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k)\left[\mathbf{B}(j,:) * \mathbf{C}(k,:)\right], \tag{3.1}$$

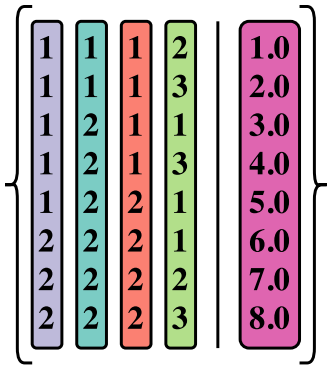


Figure 3.1: A sparse tensor represented in coordinate format. The tensor has four modes and eight non-zeros.

for a total of $3F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ operations.

Elementwise approaches are natural when $\boldsymbol{\mathcal{X}}$ is represented in *coordinate format*, the most popular storage format for sparse tensors. Shown in Figure 3.1, the coordinate format for sparse tensors represents the tensor as an $\mathrm{nnz}(\boldsymbol{\mathcal{X}})\times N$ matrix of indices and a vector of values of length $\mathrm{nnz}(\boldsymbol{\mathcal{X}})$.

Sparse Tensor-Vector Products

Each column of $\mathbf{K}$ is a linear combination of the fibers of $\boldsymbol{\mathcal{X}}$ with columns of $\mathbf{B}$ and $\mathbf{C}$. MTTKRP can be viewed as a series of F tensor-vector products [21]. Using tensor-vector products is the chosen method for several major MATLAB implementations such as Tensor Toolbox [22] and Tensorlab [23].

A three-mode tensor requires two tensor-vector products per column of $\mathbf{K}$. A temporary array $\boldsymbol{t}$ of size $\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ is used to "stretch" the vectors $\mathbf{B}(:,f)$ and $\mathbf{C}(:,f)$ to map to non-zeros of $\boldsymbol{\mathcal{X}}$. For each column f, the two tensor-vector products are performed at once and stored within $\boldsymbol{t}$. Once $\boldsymbol{t}$ is filled, we need to "shrink" it to a vector of length I. This effectively sums all of the entries of $\boldsymbol{t}$ that correspond to non-zeros $\boldsymbol{\mathcal{X}}(i,:,:)$ and stores the result as $\mathbf{K}(i,f)$. Algorithm 5 presents pseudocode for computing tensor-vector products.

Using sparse tensor-vector products uses $3F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ FLOPs (i.e., $2F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ for the initial products and $F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ for the accumulation steps) and $\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ intermediate

Algorithm 5 MTTKRP via Sparse Tensor-Vector products.

Input: $\mathcal{X}$ stored in coordinate format

Output: $\mathbf{K} \leftarrow \mathcal{X}(\mathbf{C} \odot \mathbf{B})$

```
for f ← 0 to F do
    for z ← 0 to nnz(X) do                          ▷ Vector products
        t[z] ← vals[z]B(indJ[z], f)C(indK[z], f)
    end for
    for z ← 0 to nnz(X) do                          ▷ Accumulate K(:, f)
        K(indI[z], f) ← K(indI[z], f) + t[z]
    end for
end for
```

memory words for $\boldsymbol{t}$. Each non-zero can be processed in parallel during the "stretch" stage because a non-zero will only modify a single element of $\boldsymbol{t}$. The accumulation step does not have this guarantee, however, and must be executed serially. An advantage of this method is that $\mathcal{X}$ can be simply stored in coordinate format and the algorithm can be implemented in just a few lines of code in MATLAB.

3.1.2 CPD on Distributed-Memory Systems

Distributed-memory algorithm design brings new challenges over shared-memory algorithm design. In addition to concerns towards memory- and operation-efficiency, distributed-memory algorithms must be careful to load balance computations among the multiple machines and to also minimize network communication, which is often the primary overhead in parallel execution.

GigaTensor

GigaTensor [24] is a parallel CPD-ALS algorithm developed for the MapReduce [25] paradigm. GigaTensor utilizes the parallelism of MapReduce by reformulating MTTKRP as a series of Hadamard products. There are no dependencies during a Hadamard product and each element of the output can be computed in parallel.

GigaTensor avoids the construction of $\mathbf{C} \odot \mathbf{B}$ by separately computing the contributions of $\mathbf{B}(:, f)$ and $\mathbf{C}(:, f)$ with $\mathbf{X}_{(1)}$ via two Hadamard products. After computing

the separate contributions, the results are combined via a third Hadamard product. The resulting matrix $\mathbf{N}$ has the same sparsity pattern as $\mathbf{X}_{(1)}$ and each non-zero entry $\mathbf{N}(i, y)$ is equal to

$$\mathbf{N}(i, y) = \mathbf{X}_{(1)}(i, y)\mathbf{B}(y\%J, f)\mathbf{C}(y/J, f). \tag{3.2}$$

After computing the entries of $\mathbf{N}$, the rows of the resulting matrix are summed to form a column of $\mathbf{K}$. The total process requires $5F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ FLOPs and $\mathrm{nnz}(\boldsymbol{\mathcal{X}}) + \max(J, K)$ intermediate memory.

DFacTo

DFacTo [26] is a CPD-ALS algorithm designed for distributed tensor factorization. DFacTo uses an efficient MTTKRP algorithm that is posed as a series of sparse matrix-vector multiplications (SpMVs). $\mathbf{K}$ is computed one column at a time and each column is formed by two SpMVs. DFacTo first forms $\mathbf{X}_{(2)}^{\intercal}$, an $IK \times J$ sparse matrix whose rows consist of the mode-2 fibers of $\boldsymbol{\mathcal{X}}$. When forming column $\mathbf{K}(:, f)$ we first compute $\mathbf{X}_{(2)}^{\intercal}\mathbf{B}(:, f)$ and store the result as the values of $\mathbf{K}^r$, an $I \times K$ sparse matrix. Finally, we compute $\mathbf{K}^r\mathbf{C}(:, f)$ and store the result in $\mathbf{K}(:, f)$. The process is repeated for each of the F columns.

DFacTo requires $(2\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})+P+1)$ memory words to store $\boldsymbol{\mathcal{X}}$, where P is the number of non-empty mode-2 fibers. An additional $(2P+I+1)$ words are required to store $\mathbf{K}^r$ for a total memory footprint of $(2\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})+I+3P+2)$ words. DFacTo executes MTTKRP in $2F(\mathrm{nnz}(\boldsymbol{\mathcal{X}})+P)$ FLOPs. DFacTo consists entirely of SpMV operations and therefore can take advantage of a wealth of existing research that can be applied to an efficient parallel implementation.

HyperTensor

Kaya and Uçar presented HyperTensor, a parallel algorithm and software package for distributed CPD-ALS [27]. HyperTensor uses a *fine-grained* decomposition over $\boldsymbol{\mathcal{X}}$ in which non-zeros are individually assigned to processes. Several methods of computing such a decomposition are presented, with the most successful relying on hypergraph partitioning. HyperTensor maps $\boldsymbol{\mathcal{X}}$ to a hypergraph with $\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ vertices and $I+J+K$ hyperedges. The vertex representing non-zero $\boldsymbol{\mathcal{X}}(i, j, k)$ is connected to hyperedges

i, j, and k. Their experimental evaluation show that a balanced partitioning of the hypergraph leads to a load-balanced computation with low communication volume.

3.1.3 Constrained CPD

There is a large body of research dedicated to optimization algorithms for constrained tensor factorization, especially in the context of non-negativity constraints. Zhang et al. [28] presented a parallel algorithm for dense non-negative tensor factorization using projected gradient descent. Non-negative tensor factorization was formulated for the ADMM framework by Liavas and Sidiropoulos [29]. Kannan et al. [30] developed a parallel algorithm for dense and sparse non-negative *matrix* factorization using non-negative least squares. For additional background on non-negative factorizations, we direct the reader to the survey by Zhou et al. [31] and the book by Cichocki et al. [32].

3.2 Tensor Completion in the CPD Model

3.2.1 Alternating Least Squares

Parallel ALS algorithms exploit the independence of the I least squares problems and solve them in parallel. ALS was one of the first optimization algorithms applied to large-scale matrix completion [33]. Gates et al. [34] developed a high performance ALS algorithm for matrix completion on CPUs and GPUs was The algorithm exploits level-3 BLAS opportunities during the construction of $\mathbf{H}_i^T\mathbf{H}_i$.

ALS was first extended to tensor completion on shared-memory systems [12]. Shin and Kang presented a distributed-memory implementation based on the MapReduce paradigm [14]. They use a *coarse-grained* tensor decomposition which partitions each mode separately, assigning to each process a set of complete $\boldsymbol{\mathcal{X}}(i,:,:)$, $\boldsymbol{\mathcal{X}}(:,j,:)$, and $\boldsymbol{\mathcal{X}}(:,:,k)$ slices. After a process updates $\mathbf{A}(i,:)$, the new values are broadcasted to all other processes. Karlsson et al. developed a distributed-memory algorithm for MPI [13]. The distributed-memory algorithm assigns tensor entries to processes without restriction, allowing for $\text{nnz}(\boldsymbol{\mathcal{X}})$ parallelism. The added parallelism comes with the cost of communicating partial computations of $\mathbf{H}_i^T\mathbf{H}_i$ and $\mathbf{H}_i^T \text{vec}(\boldsymbol{\mathcal{X}}(i,:,:))$ with an all-reduce, requiring $\mathcal{O}(IF^2)$ words communicated per process.

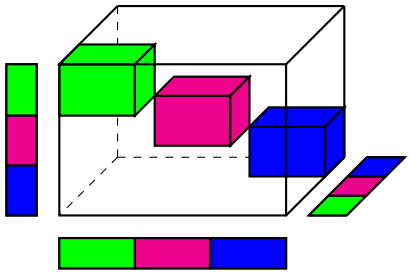


Figure 3.2: Stratified SGD. Colored blocks of observations can be processed in parallel without conflict.

3.2.2 Stochastic Gradient Descent (SGD)

SGD is parallelized by exploiting the independence of entries whose coordinates are disjoint. The popular *stratification*-based SGD method [35] partitions an N-mode tensor into a P^N grids for P processes. For example, Figure 3.2 shows a case with N=3 and P=3. The grid has P^{N-1} strata, each corresponding to P blocks in a diagonal line (i.e., $\{(1, t_2, ..., t_N), (2, (t_2+1) \bmod P, ..., (t_N+1) \bmod P), ..., (P, (t_2+P-1) \bmod P, ..., (t_N+P-1) \bmod P)\}$ for all $1 \leq t_2, ..., t_N \leq P$). Each epoch comprises processing P^{N-1} strata, which covers all of the entries of the tensor. Since no two entries in different blocks of a given stratum share the same index in any mode, P processes can work on P blocks of a stratum in parallel.

Instead of stratification, some parallel SGD methods allow entries with overlapping coordinates to be processed in parallel. Hogwild [36], a parallel algorithm for shared-memory systems, exploits the stochastic nature of SGD to have lock-free parallelism. The concept is simple: process the shuffled entries in parallel without stratification or synchronization constructs. Due to the sparse nature of the input, race conditions are expected to be rare. When they do occur, the stochastic nature of the algorithm will naturally fix any errors and continue to converge. A similar idea for distributed-memory systems is *asynchronous* SGD (ASGD). All entries are processed in parallel, and a few times per epoch processes combine local updates to overlapped rows via weighted averages. The communication and averaging of local updates are performed asynchronously [37].

3.2.3 Coordinate Descent (CCD++)

CCD++ is parallelized in the same manner as ALS in the tensor completion setting. All of the α_i's and β_i's for a mode are computed independently, again leading to a coarse-grained decomposition of $\boldsymbol{\mathcal{X}}$ [15, 14]. After updating a factor column in parallel, the new column factors are broadcasted to all processes. Karlsson et al. use a non-restrictive decomposition of the tensor entries as in ALS [13]. Partial products are again aggregated with an all-reduce and new columns are broadcasted. Only the components of the current column must be communicated, and so in one epoch there are $2F$ messages per factor matrix, each of size $\mathcal{O}(I)$.

3.3 Tucker Decomposition

TTMc is the key kernel when computing the Tucker Decomposition for sparse tensors. Like MTTKRP, considerations for scalability include memory consumption, degree of parallelism, and the number of required operations. Several approaches for TTMc have been proposed. They fall into two broad categories: those which use additional computation in order to avoid intermediate memory blowup, and those which use memoization to accelerate computation.

Memory-Efficient Tucker [38] avoids memory blowup by selectively computing columns or elements of $\mathbf{Y}_{(n)}$ at a time. Intermediate memory costs are minimized at the expense of additional FLOPs and passes over the tensor structure.

Baskaran et al. [39] observed that partial computations can be reused across TTMc kernels. Consider updating the first two factors of a four-mode tensor. Each TTMc kernel constructs the partial computation $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}^{(3)T} \times_4 \mathbf{A}^{(4)T}$, despite its value not changing between kernels. Baskaran et al. introduced memoization to TTMc by partitioning the tensor modes into two halves, and reusing the computations from one half to accelerate the computations in the other half. Kaya and Uçar extended this memoization strategy by using binary dimension trees to accelerate both the Tucker decomposition [40] and CPD [41]. They store intermediate computations in the nodes of the tree and can effectively limit the number of individual n-mode products to $\log(N)$ per TTMc operation.

Kaya and Uçar also showed that one can avoid intermediate blowup by processing

individual non-zeros [40]. For example, the following is used for mode-1:

$$\mathbf{Y}_{(1)}(i_1,:) \leftarrow \mathbf{Y}_{(1)}(i_1,:) + \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\left[\mathbf{A}^{(2)}(i_2,:)\otimes\cdots\otimes\mathbf{A}^{(N)}(i_N,:)\right]. \tag{3.3}$$

A row of $\mathbf{Y}_{(1)}$ is the only memory required to process a non-zero. The computational complexity of using (3.3) to perform one TTMc kernel via streaming through each non-zero is the cost of construction the Kronecker Products (KP) and accumulating them into the result:

$$\text{nnz}(\boldsymbol{\mathcal{X}}) \underbrace{\sum_{i=2}^{N}\prod_{j=2}^{i} F_j}_{\text{KP construction}} + \text{nnz}(\boldsymbol{\mathcal{X}}) \underbrace{2\prod_{j=2}^{N} F_j}_{\text{accumulation}} = \mathcal{O}\left(\text{nnz}(\boldsymbol{\mathcal{X}})\prod_{j=2}^{N} F_j\right). \tag{3.4}$$

Chapter 4

Accelerating the CPD on Multi-Core Systems

In order to decompose large and sparse tensors, we must address issues including operation- and memory-efficiency, parallelism, and cache locality. This chapter derives a new way of computing MTTKRP by exploiting redundancies resulting from the tensor structure. We detail methods of reordering the tensor indices and tiling over the tensor to improve cache locality. We then present a compressed data structure that exposes the redundancies inherent in the computations to enable the operation-efficient algorithm. This data structure is also shown to expose parallelism that is inherent in the algorithm. Lastly, we present methods of reducing the memory overhead of multiple tensor representations and evaluate the inherent trade-offs.

4.1 An Operation-Efficient Algorithm for MTTKRP

Recall the contribution of each non-zero in the elementwise MTTKRP formulation discussed in Section 3.1.1:

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k)\left[\mathbf{B}(j,:) * \mathbf{C}(k,:)\right].$$

If we consider subsequent contributions from two non-zeros that reside in the same $\boldsymbol{\mathcal{X}}(i,j,:)$ fiber

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\left[\mathbf{B}(j,:) * \mathbf{C}(k_1,:)\right],$$
$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\left[\mathbf{B}(j,:) * \mathbf{C}(k_2,:)\right],$$

we note that the same rows of $\mathbf{B}$ and $\mathbf{K}$ are accessed. Since multiplication is a distributive operation, we can rearrange the computations into a single expression in order to reduce the number of FLOPs and memory accesses:

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \mathbf{B}(j,:) * \left[\boldsymbol{\mathcal{X}}(i,j,k_1)\mathbf{C}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\mathbf{C}(k_n,:)\right].$$

This strategy can more generally be applied to complete fibers of $\boldsymbol{\mathcal{X}}$:

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \mathbf{B}(j,:) * \sum_{\boldsymbol{\mathcal{X}}(i,j,:)} \boldsymbol{\mathcal{X}}(i,j,k_n)\mathbf{C}(k_n,:). \tag{4.1}$$

Intuitively, Equation (4.1) simultaneously accumulates the F inner products between the tensor non-zeros in $\boldsymbol{\mathcal{X}}(i,j,:)$ and the columns of $\mathbf{C}$, and then performs a single Hadamard product with $\mathbf{B}(j,:)$ before updating the resulting row $\mathbf{K}(i,:)$ once. If $\boldsymbol{\mathcal{X}}$ is sparse and fiber $\boldsymbol{\mathcal{X}}(i,j,:)$ has $\hat{K}$ non-zeros, then $F(\hat{K}-1)$ FLOPs are saved, resulting in a total $2F(\text{nnz}(\boldsymbol{\mathcal{X}})+P)$ FLOPs when there are P fibers in the tensor.

It is important to note that the choice to factor out accesses to $\mathbf{B}$ as opposed to $\mathbf{C}$ is arbitrary. Equation (4.1) can equivalently be expressed

$$\mathbf{K}(i,:) \leftarrow \mathbf{K}(i,:) + \mathbf{C}(k,:) * \sum_{\boldsymbol{\mathcal{X}}(i,:,k)} \boldsymbol{\mathcal{X}}(i,j_n,k)\mathbf{B}(j_n,:).$$

Deciding which term to factor out impacts storage and computational performance. The decision is most relevant when the mode lengths of $\boldsymbol{\mathcal{X}}$ are not equal. By storing fibers along the longer mode, we are able to minimize the number of stored fibers and increase the average fiber length. The benefit of this scheme is twofold: we reduce the amount of memory required to store the tensor and reduce the number of memory accesses and FLOPs due to a larger number of factored multiplications. Section 4.5.2 demonstrates the benefits of selecting the best mode to factor.

Figure 4.1 illustrates the fiber-centric algorithm. The algorithm that follows from Equation (4.1) is detailed in Algorithm 6.

Algorithm 6 A fiber-centric MTTKRP algorithm.

Input: Sparse tensor $\mathcal{X}$

Output: $\mathbf{K} \leftarrow \mathcal{X}(\mathbf{C} \odot \mathbf{B})$

```
for all unique i ∈ X(:,:,:) do                    ▷ Each X(i,:,:) slice
    K(i,:) ← 0
    for all unique j ∈ X(i,:,:) do                ▷ Each X(i,j,:) fiber
        z ← 0                                     ▷ F×1 vector for accumulation
        for all k ∈ X(i,j,:) do
            z ← z + X(i,j,k)C(k,:)
        end for
        K(i,:) ← K(i,:) + B(j,:) * z
    end for
end for
```

4.1.1 Parallelization

Algorithm 6 can be parallelized by using a task decomposition on the rows of $\mathbf{K}$. Since the computation of $\mathbf{K}(i,:)$ requires only the non-zeros in slice $\mathcal{X}(i,:,:)$, the mode-1 slices of $\mathcal{X}$ can be distributed among threads. All threads write to distinct rows of $\mathbf{K}$ and thus parallel execution requires no locks or synchronization. Each thread requires only F words of additional storage to accumulate inner products. This method is memory scalable because $F \ll \text{nnz}(\mathcal{X})$.

The unstructured sparsity pattern of $\mathcal{X}$ poses the issue of potential load imbalance. The non-zeros of $\mathcal{X}$ are rarely distributed in a uniform fashion. For example, the number of non-zeros across the slices of real-world datasets can vary by several orders of magnitude and often exhibit a *power law* distribution. A static decomposition of rows can assign disproportionate amounts of work to the processes, resulting in severe load imbalance and reduced scalability. Therefore our implementation in SPLATT uses dynamic load balancing when distributing slices to threads.

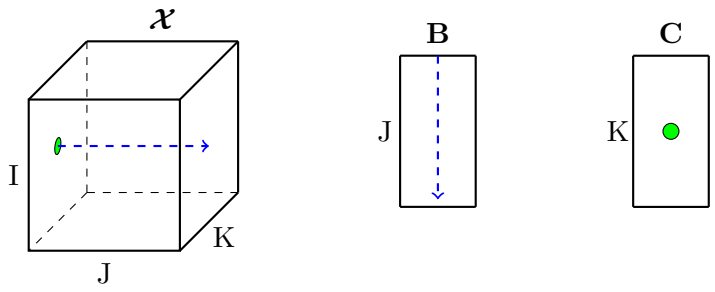


Figure 4.1: The fiber-centric MTTKRP algorithm. The dashed blue line shows the fiber of $\boldsymbol{\mathcal{X}}$ and its inner product with a column of $\mathbf{B}$. The inner product is then scaled by the circled value of $\mathbf{C}$. All columns are computed at once.

4.1.2 Extensions to Higher-Order Tensors

Algorithm 6 can be extended to higher-order tensors by considering the block structure of the Khatri-Rao product. If $\boldsymbol{\mathcal{X}}$ is an N-mode tensor, then MTTKRP in the first mode becomes

$$\mathbf{K} = \mathbf{X}_{(1)}(\mathbf{A}^{(N)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(2)}).$$

The block structure in the Khatri-Rao product becomes more pronounced as N increases. The fiber-centric algorithm is able to exploit this block structure by factoring out a new set of multiplications per mode:

$$\mathbf{K}(i_1,:) = \sum \mathbf{A}^{(2)}(i_2,:) * \Big(\sum \mathbf{A}^{(3)}(i_3,:) * \Big(\dots \Big(\sum \boldsymbol{\mathcal{X}}(i_1, i_2, \dots, i_N) \mathbf{A}^{(N)}(i_N,:) \Big) \Big) \Big).$$

The Khatri-Rao product operates on $N-1$ modes, requiring $F(N-2)$ words of intermediate memory. The last mode does not need intermediate memory because it writes to $\mathbf{K}$ directly. Like before, fibers of $\boldsymbol{\mathcal{X}}$ are used for inner products with $\mathbf{A}^{(N)}$, which are then scaled by the corresponding row of $\mathbf{A}^{(N-1)}$ and so on.

When forming each of the N representations of $\boldsymbol{\mathcal{X}}$, we must choose an ordering of the remaining $N-1$ modes. As discussed in Section 4.1, arranging the modes to minimize the number of fibers (and maximize the average fiber length) can have a significant impact on the storage and computation required. This is achieved by sorting modes by their length such that the shortest modes correspond to outer loops and the longest mode corresponds to the direction that $\boldsymbol{\mathcal{X}}$ stores its fibers.

4.2 Optimizing for Cache Performance

We now present a method of achieving further speedup of Algorithm 6 by efficiently utilizing the CPU memory hierarchy through means of reordering and cache blocking.

4.2.1 Tensor Reordering

Permuting the indices within one or more modes, or *reordering*, can lead to significant performance gains as it can potentially improve cache utilization by exploiting spatial and temporal locality. Figure 4.2 illustrates a tensor before and after reordering.

The goal of reordering a sparse tensor is to group non-zeros into semi-dense regions. non-zeros form a sequence of semi-dense cuboids along the super-diagonal after an ideal reordering. Dense regions are attractive because they offer increased cache performance while accessing **B** and **C**. Consider the execution of Algorithm 6 along the first mode. The mode-2 indices in a fiber determine which rows of **B** are accessed and the constant mode-3 index determines the accessed row of **C**. Consecutive mode-2 indices result in an unstrided access pattern that offers spatial locality in memory and can effectively utilize hardware prefetching mechanisms. If the accessed portion of **B** is sufficiently small and there are shared mode-2 indices in nearby fibers, the required portions of **B** will still reside in cache. Additionally, as other slices are processed we can also see the same reuse in **C** due to repeated mode-3 indices.

In this work we identify two methods of reordering sparse tensors. The first is based on the partitioning of a graph that models the interactions between slices of each mode of $\boldsymbol{\mathcal{X}}$. This method is *mode-independent* because a single reordering is used for each mode of computation. The second method is based on the partitioning of a hypergraph that models the memory accesses to **K**, **B**, and **C**. Unlike the tripartite graph model, the hypergraphs are specific to a given mode (*mode-dependent*) and thus multiple reorderings are needed.

Mode-Independent Reorderings

The objective of a mode-independent reordering is to find a single tensor permutation that results in improved execution time regardless of which mode MTTKRP is being performed on. We achieve this goal by permuting modes of the tensor such that indices

$$\left[\begin{array}{cccc|cccc|cccc} 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 2 & 0 & 0 & 2 \\ 0 & 3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

(a) An unordered 4×4×3 tensor.

$$\left[\begin{array}{cccc|cccc|cccc} 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$$

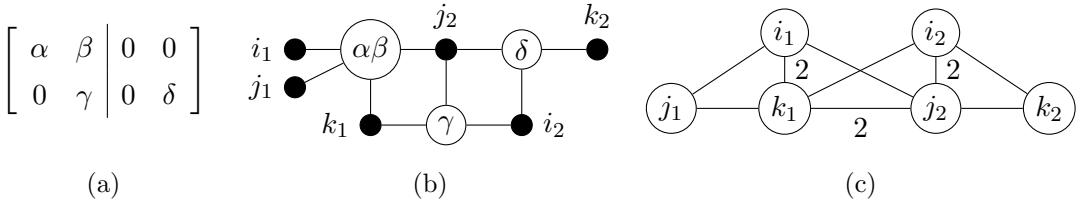
(b) The same tensor, reordered to improve memory locality.

Figure 4.2: A sparse tensor before and after reordering.

with high levels of similarity are adjacent.

We draw from the *bipartite graph model* for reordering sparse matrices [42]. Suppose $\boldsymbol{\mathcal{X}}$ is an N-mode tensor with dimensions $I_1 \times I_2 \times \ldots \times I_N$. We construct an N-partite graph whose vertex sets are of cardinalities $I_1 \times I_2 \times \ldots \times I_N$. Non-zero $\boldsymbol{\mathcal{X}}(i_1, i_2, \ldots, i_N)$ generates a clique that connects nodes $i_1, i_2, \ldots, i_N$. Using this scheme, edge (i_a, i_b) will be created every time a non-zero is processed with indices i_a and i_b appearing together. To account for this, we weight edges based on the number of times they are generated. For example, when $\boldsymbol{\mathcal{X}}$ has three modes the resulting graph is tripartite and edge (i, k) has weight equal to the number of non-zeros in the $\boldsymbol{\mathcal{X}}(i, :, k)$ fiber. Figure 4.3 shows a small tensor and its corresponding graph.

After generating a tensor's graph, a graph partitioner is used to create a partitioning. The graph is next relabeled such that vertices in the same partition are given consecutive labels. Finally, we generate a reordered tensor from the relabeled graph.



(a) (b) (c)

Figure 4.3: (a) $\boldsymbol{\mathcal{X}}$, a 2×2×2 tensor. (b) $\boldsymbol{\mathcal{X}}$ mapped to a mode-1 hypergraph whose nodes are the $\boldsymbol{\mathcal{X}}(i,:,k)$ fibers. Filled nodes are hyperedges. (c) $\boldsymbol{\mathcal{X}}$ mapped to a tripartite graph with unlisted weights assumed to be unit.

Mode-Dependent Reorderings

Mode-dependent reorderings offer further opportunities for optimization at the cost of additional work during the reordering stage. When operating within a certain mode we know precisely which memory accesses will result from the tensor's sparsity pattern.

Our hypergraph model is an extension of the *column-net model* originally used for parallel sparse matrix-vector multiplication [43]. Fibers are our unit of work and are analogous to rows in a sparse matrix. Fibers are mapped to vertices in the hypergraph. Each mode emits as many hyperedges as its own dimension. A three-mode tensor will have $I+J+K$ hyperedges. Each hyperedge connects all fibers that its corresponding index can be found within. For example, if fiber $\boldsymbol{\mathcal{X}}(i,:,k)$ has three non-zeros, then that vertex will be connected by five hyperedges. Two connections will come from the i and k indices and the final three will come from each non-zero mode-2 index found in the fiber.

Our goal is to model memory accesses as hyperedges. The number of partitions in which a hyperedge is found (or, its *connectivity*) exactly models the number of times that its corresponding row in $\mathbf{K}$, $\mathbf{B}$, or $\mathbf{C}$ must be fetched from memory. Thus, by minimizing the connectivity of all hyperedges (known as the *sum of external degrees*), we minimize the number of total memory accesses.

We partition the hypergraph to induce a reordering of the tensor. Fibers (vertices) are relabeled such that fibers in the same partition are given consecutive labels. Relabeling a fiber means to relabel all indices found in its non-zero entries. Indices are not unique to fibers and so we ensure that we only label an index the first time it is

encountered. Consider fibers stored along the second mode. Mode-1 and mode-3 indices determine the order in which fibers are processed. This affects temporal locality because it allows fibers with similar sparsity pattern to be processed nearby in time. Relabeled mode-2 indices affect spatial locality and allow a fiber and its neighbors to access consecutive rows of $\mathbf{B}$.

A clear drawback of a mode-dependent reordering is the need to construct and partition a hypergraph for each mode. Fortunately, much of this cost is mitigated due to the ordering of modes. Recall that we store fibers along the mode with the largest dimension. Consider a tensor of dimensions $I \times J \times K$ and $I < J < K$. SPLATT will store fibers along the third mode for the first two modes of computation. During the third mode, fibers will be stored along the second mode because it has the next largest dimension. The only difference in execution between the first and second modes is the order in which fibers are processed. Thus, the hypergraphs of the first and second modes will be identical except for the labels of mode-1 and mode-2 hyperedges. A consequence is that a partitioning of one hypergraph will be equally suited for the other. Therefore, only partitionings of the mode-1 and mode-3 hypergraphs are needed for a complete reordering.

4.2.2 Cache Blocking over Sparse Tensors

The long modes that sparse tensors often exhibit are prohibitive to memory performance, even with a good reordering. Assume that fibers run along the second mode and are defined by a unique (i, k) pair. Long fibers will fetch enough of the rows of $\mathbf{B}$ to evict cache lines that would otherwise be reused in other nearby fibers. In order to maximize data reuse, we use cache blocking.

Our method of blocking over a sparse tensor during MTTKRP is a generalization of the blocking used for matrix-vector multiplication. We seek to define three-dimensional tiles over the sparsity pattern of $\boldsymbol{\mathcal{X}}$. If a tile has dimension $I' \times J' \times K'$ then accesses to $\mathbf{K}$, $\mathbf{B}$, and $\mathbf{C}$ are limited to a maximum of I', J', and K' rows, respectively. Thus, by carefully selecting tile dimensions such that the entire working set fits in CPU cache we can increase reuse of $\mathbf{K}$, $\mathbf{B}$, and $\mathbf{C}$.

Tiling over a sparse tensor is not a trivial task. An implementation that statically assigns non-zeros to tiles based on their coordinates and the tile dimensions will result in

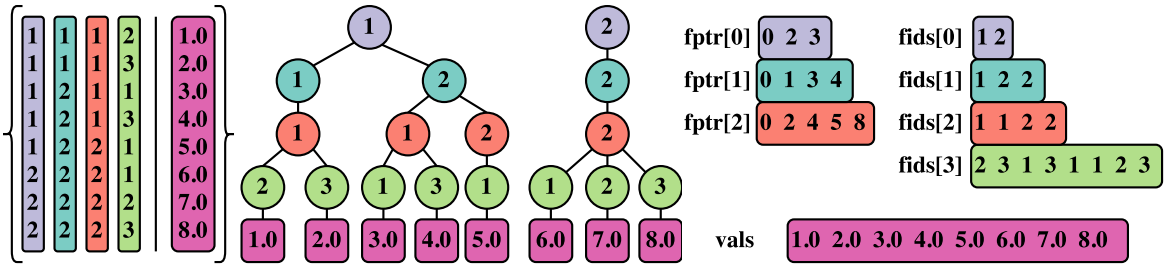
mostly empty or near-empty tiles due to the high levels of sparsity present. Additionally, most datasets feature unstructured sparsity patterns that can result in tiles of wildly varying density. We propose a method of *growing* tiles to adapt to the sparsity pattern of a given tensor.

First, we divide the mode-1 slices into sets of size I'. We call a set of slices a *layer*. Since empty slices can be trivially removed from the dataset, we assume that they are either not present or are rare enough such that we may statically assign slice i to layer i/I'. The sparsity pattern of each layer may differ dramatically and thus each layer is given an independent tiling.

We proceed one layer at a time. Within each layer we first construct the set of mode-3 indices present. We divide the indices into sets of size K' and arrange the $\boldsymbol{\mathcal{X}}(i,:,k)$ fibers into *tubes*, each with a maximum of I' mode-1 indices and K' mode-3 indices. Each tube must be tiled independently due to their varying sparsity patterns.

Finally, within each tube we construct the set of mode-2 indices that are present. This set is used to divide the tube into tiles with $I' + J' + K'$ unique indices. If we choose dimensions so that $F(I' + J' + K')$ floating-point numbers can comfortably fit in cache, and the ordering of $\boldsymbol{\mathcal{X}}$ provides regions which are relatively dense, then we have effectively increased reuse in $\mathbf{K}$, $\mathbf{B}$, and $\mathbf{C}$.

All of the fibers within a slice are no longer adjacent in memory after tiling. Consequently, parallel execution within a layer is difficult because writes to the same row of $\mathbf{K}$ can occur at any time. We identify two methods of modifying Algorithm 6 to execute over a tiled tensor. The first method is to distribute the tiled layers among threads and prevent race conditions while avoiding synchronization or atomics. The drawback of distributing entire layers is that the working set of each tile is now local to individual threads. The data reused between threads will be limited to similarities in sparsity pattern between layers. The second method of tiling is a cooperative scheme. Each thread uses its own $I' \times F$ matrix of scratch space to accumulate writes to $\mathbf{K}$. All threads execute concurrently within a tile but must synchronize at the end of each layer. After the synchronization, threads cooperate to do a summation of the scratch matrices. Since we operate in a shared address space we are able to evenly distribute the I' rows of scratch space among threads and do a reduction with only a synchronization at the end of the algorithm.



(a) Coordinate. (b) CSF: conceptual. (c) CSF: implementation.

Figure 4.4: A four-mode tensor represented in (a) coordinate format and (b) compressed sparse fiber. Each path from root to leaf in (b) encodes a non-zero found in (a).

4.3 A Compressed Data Structure for Sparse Tensors

Algorithm 6 necessitates a storage scheme for $\boldsymbol{\mathcal{X}}$ that provides efficient traversals of the fiber structure of the tensor. We propose to represent sparse tensors in a hierarchical, fiber-centric fashion in order to extract the opportunities for computational savings via the data representation itself.

Our proposed storage format is called *compressed sparse fiber* (CSF). Suppose we have a sparse tensor with N modes. We form a tree with N levels from each of the $\boldsymbol{\mathcal{X}}(i, :, \ldots, :)$ subtensors. Nodes in level l contain indices found in the lth mode. Paths from root to leaf encode a non-zero coordinate and their values are also stored in the leaves. Figure 4.4 is an example of a four-mode tensor in this format.

CSF can be viewed as a generalization of the compressed sparse row (CSR) data structure that is popular for sparse matrices. CSF is implemented with three multidimensional arrays: (i) `fptr`, which encodes the sparsity structure; (ii) `fids`, which stores the label of each node; and (iii) `vals`, which stores the non-zero values. Conceptually, the `fptr` structure is a sequence of N `rowptr` arrays used in CSR. Each array is used to index into the next in the sequence, with the final array indexing directly into the non-zeros. Figure 4.4c illustrates this implementation.

Compression in a CSF tensor is visualized by the branching factor. Nodes with more than one child are the result of joining two or more repeated indices. Storage savings compound as we move deeper into a node's subtrees. For example, in Figure 4.4b, the tree uses only two nodes to store the mode-1 coordinates for all eight non-zeros.

We note that storage requirements for CSF and DFacTo are similar. Both are ultimately focused on a representation of the sparse fibers in $\boldsymbol{\mathcal{X}}$. However, CSF only needs to store F additional floating-point numbers during computation and does not need the $2P$ memory words used to store $\mathbf{K}^r$ that DFacTo requires. Additionally, Algorithm 6 exhibits memory access patterns that have better spatial locality because the sparse structure of $\boldsymbol{\mathcal{X}}$ is traversed only once and each non-zero value is used in F multiplications after being fetched from memory. This is a result of the row-oriented approach taken by Algorithm 6, which to our knowledge is the first of its kind in the sparse tensor community.

4.3.1 Implementing MTTKRP with a CSF Tensor

We now formalize Algorithm 6 to a general number of modes using the CSF data structure. We assume that CSF is constructed with the mode-of-interest at the top level of the CSF data structure, and thus the keys of the root nodes determine which row of $\mathbf{K}$ to update. For simplicity of notation, in the following discussion we assume that the modes of $\boldsymbol{\mathcal{X}}$ have been permuted such that the root level is the first mode, the second level of the CSF data structure corresponds to the second mode, and so on. We call this adaptation CSF-ROOT.

Non-zero $\boldsymbol{\mathcal{X}}(i_1, i_2, \ldots, i_N)$ scales the row vector $\mathbf{A}^{(2)}(i_2, :) * \cdots * \mathbf{A}^{(N)}(i_N, :)$. A sibling non-zero, $\boldsymbol{\mathcal{X}}(i_1, i_2, \ldots, i'_N)$, will likewise require $\mathbf{A}^{(2)}(i_2, :) * \cdots * \mathbf{A}^{(N)}(i'_N, :)$. Note that the multiplications only differ in the $\mathbf{A}^{(N)}$ terms which correspond to leaf nodes in the tree. CSF-ROOT saves operations by factoring out the repeated multiplications and instead accumulates sibling non-zeros into a buffer. Once completed, the buffer is scaled by the factored multiplication and the result is propagated up the tree. The optimization is not unique to the leaf level; we can save operations by factoring redundant multiplications at each level of the tree as discussed in Section 4.1.2.

Algorithm 7 shows a recursive implementation of CSF-ROOT. CSF-ROOT performs a depth-first traversal on the CSF tensor in which row $\mathbf{K}(i, :)$ is completed at the end of the traversal of the ith tree. Hadamard products are accumulated in a buffer $\mathbf{Z}$, an $N \times F$ matrix which in general is a small constant in size. A node in the dth level accumulates the contributions from all children into $\mathbf{Z}(d, :)$ before propagating its own contributions up the tree.

Algorithm 7 MTTKRP for the root nodes of a CSF tensor.

```
1: function PROPAGATE-UP(node)
2:     d ← DEPTH(node)
3:     id ← IDX(node)                          ▷ Extract the dth coordinate
4:     if d = m then
5:         return VAL(node) · A^(d)(id, :)     ▷ Scale by the non-zero value in the leaf
6:     end if
7:     Z(d, :) ← 0                             ▷ Accumulate all children into Z
8:     for c ∈ children(node) do
9:         Z(d, :) ← Z(d, :) + PROPAGATE-UP(c)
10:    end for
11:    if d = 1 then
12:        return Z(d, :)
13:    else
14:        return Z(d, :) ∗ A^(d)(id, :)
15:    end if
16: end function
17:
18: function CSF-ROOT(X)
19:    for i ∈ I_1 do
20:        K(i, :) ← PROPAGATE-UP(X(i, :, . . . , :))
21:    end for
22: end function
```

4.4 Memory-Efficient MTTKRP Algorithms

The alternating nature of CPD-ALS means that MTTKRP must be computed once for each mode of the tensor, per iteration. These computations are *mode-centric*, meaning that we are computing for a specific mode each time. Thus, to use Algorithm 7 one requires N representations of $\boldsymbol{\mathcal{X}}$ (i.e., each of the N modes must be placed at the top level of the tree). Storing N different compressed tensors makes it easy to extract coarse-grained parallelism in Algorithm 7, but is not memory-efficient and does not scale to tensors with more than a few modes. We will now detail how it is possible to perform MTTKRP on any mode of a CSF tensor while reducing floating-point operations in a manner similar to CSF-ROOT.

Computing for the Leaf Level

MTTKRP on the lowest level of the tree uses the keys of the leaves to determine the output rows of $\mathbf{K}$. We use the same principle from CSF-ROOT to factor redundant Hadamard products and save floating-point operations. In this case, instead of accumulating non-zeros in a buffer to propagate up the tree, we recursively propagate Hadamard products down the tree and allow siblings to reuse previous multiplications. We call this algorithm CSF-LEAF and present a recursive implementation in Algorithm 8.

Computing for Internal Levels

Processing internal nodes relies on a combination of the CSF-ROOT and CSF-LEAF algorithms. Whereas CSF-ROOT determines write locations by the root nodes and CSF-LEAF uses the leaves, we now use one of the middle modes. We must process both parent and children subtrees before results can be stored. A recursive implementation of this algorithm, called CSF-INTL, is detailed in Algorithm 9.

Assume we are computing for the wth level of the tree. We first propagate Hadamard products down the tree using CSF-LEAF until the we are $(w-1)$ levels deep. Next, we use CSF-ROOT to propagate the contributions of the entire node's subtree up to $\mathbf{Z}(w,:)$. The final result is the Hadamard product of $\mathbf{Z}(w-1,:)$ and $\mathbf{Z}(w,:)$.

Algorithm 8 MTTKRP for the leaf nodes of a CSF tensor.

```
1:  function PROPAGATE-DOWN(node)
2:      d ← DEPTH(node)
3:      id ← IDX(node)                          ▷ Extract the dth coordinate
4:      if d = m then                ▷ Last level uses non-zeros and writes to K
5:          K(id, :) ← K(id, :) + VAL(node)Z(d, :)
6:          return
7:      end if
8:
9:      if d = 1 then                                          ▷ Initialize Z
10:         Z(1, :) ← A^(1)(id, :)
11:     else
12:         Z(d, :) ← Z(d − 1, :) ∗ A^(d)(id, :)          ▷ Propagate down the tree
13:     end if
14:     for c ∈ children(node) do
15:         CSF-LEAF(c)
16:     end for
17: end function
18:
19: function CSF-LEAF(X)
20:     for i ∈ I_1 do
21:         PROPAGATE-DOWN(X(i, :, . . . , :))
22:     end for
23: end function
```

Algorithm 9 MTTKRP on mode w, an internal level

function PROPAGATE-INTL($node$)
  $d \leftarrow$ DEPTH($node$)
  $id \leftarrow$ IDX($node$) ▷ Extract the dth coordinate
  if $d = 1$ **then** ▷ Initialize the first level
    $\mathbf{Z}(1, :) \leftarrow \mathbf{A}_d(id, :)$
  else if $d < w$ **then** ▷ Move down to level $w - 1$
    $\mathbf{Z}(d, :) \leftarrow \mathbf{Z}(d-1, :) * \mathbf{A}^{(d)}(id, :)$
    for $c \in children(node)$ **do**
      PROPAGATE-INTL(c)
    end for
  else ▷ Use CSF-ROOT to complete computation
    $\mathbf{Z}(d, :) \leftarrow$ PROPAGATE-UP($node$)
    $\mathbf{K}(id, :) \leftarrow \mathbf{K}(id, :) + (\mathbf{Z}(d-1, :) * \mathbf{Z}(d, :))$
  end if
end function

function CSF-INTL($\boldsymbol{\mathcal{X}}$)
  for $i \in I_1$ **do**
    PROPAGATE-INTR($\boldsymbol{\mathcal{X}}(i, :, \ldots, :)$)
  end for
end function

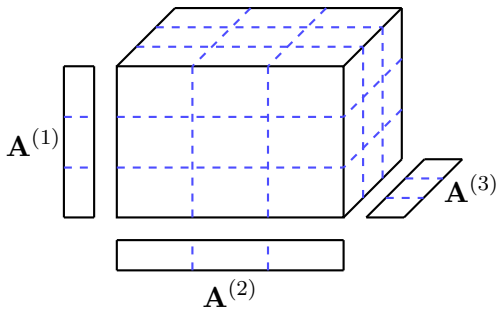


Figure 4.5: An N-dimensional tiling over $\boldsymbol{\mathcal{X}}$ for three processors. The tiles in whichever mode we are computing on form a 1D decomposition of the tensor. The tiling induces a partitioning of the matrix factors, which are distributed among threads to remove race conditions.

4.4.1 Parallel Formulation and Tiling

CSF-ROOT can be parallelized in the same way as Algorithm 6. Since each tree stores the non-zeros found in a unique $\boldsymbol{\mathcal{X}}(i, :, \ldots, :)$ slice, they can be computed in parallel without fine-grained synchronization.

When parallelizing CSF-INTL and CSF-LEAF, an important observation is that it is no longer safe to simply parallelize over trees and avoid race conditions. There is no guarantee that non-root keys are unique to the any tree, and thus we must now synchronize updates to the output matrix. One way of doing this is to reduce contention by storing a large number of mutexes, M, and using mutex $i \pmod M$ when writing to row i. This, however, is not an adequate solution. Even if we completely eliminate contention, at the leaf level we pay the price of locking and unlocking a mutex for every non-zero.

Instead, we use a method that eliminates the need for locks by imposing an N-dimensional tiling over $\boldsymbol{\mathcal{X}}$. Suppose we are computing on the second mode of a tensor. We use the tiling along the second mode to induce a 1D partitioning of the tensor non-zeros and $\mathbf{A}^{(2)}$. If the 1D blocks of non-zeros are distributed to threads, due to the nature of the tiling they will write to non-overlapping rows of $\mathbf{A}^{(2)}$. Figure 4.5 illustrates our tiling procedure for three threads on a three-mode tensor.

Lock-free parallelism comes with a price. Fibers that cross tile boundaries are treated

as separate fibers, which increases storage overhead and decreases the number of factored multiplications. It is tempting to use a large number of small tiles in order to improve cache locality, but in practice this results in more storage than the uncompressed tensor and little to no performance benefit. Thus, we only use as many tiles per mode as there are available threads.

4.5 Evaluating Fiber-Centric MTTKRP

4.5.1 Experimental Methodology

Experimental Setup

SPLATT is implemented in C with double precision floating-point numbers and 64-bit integers. SPLATT uses OpenMP for shared memory parallelism. All source code is available for download[1] . Load balance is achieved by OpenMP's dynamic scheduling with a chunksize of 16. Experiments were carried out on an HP ProLiant BL280c G6 blade server with dual 8-core E5-2670 Xeon processors running at 2.6 GHz. Source code was compiled with GCC 4.8.0 using optimization level O2. For all experiments we used F=10.

Datasets

We evaluated our method across several datasets of varying properties. Table 4.1 is a summary of the mentioned datasets.

The Netflix dataset is taken from the Netflix Prize competition [44]. The dataset forms a *user-item-time* ratings tensor. Two datasets come from the Never Ending Language-Learning (NELL) project [45] which is freely available. Both tensors represent *noun-verb-noun* triplets. NELL-1 is the complete, extremely sparse dataset and NELL-2 is a smaller, more dense version in which the infrequent items have been pruned. BrainQ [46] is derived from fMRI measurements of brain activity. Its three modes are *noun-voxel-human subject*. BrainQ is an interesting dataset because its dimensions are relatively small, resulting in a tensor several orders of magnitudes more dense than

[1] *http://cs.umn.edu/~splatt/*

Table 4.1: Summary of datasets.

Dataset	I	J	K	nnz	density
Netflix	480K	18K	2K	100M	5.4e-06
NELL-1	4M	4M	25M	144M	3.1e-13
NELL-2	15K	15K	30K	77M	1.3e-05
BrainQ	60	70K	9	11M	2.9e-01
Delicious	532K	17M	2.5M	140M	6.1e-12

nnz is the number of nonzero entries in the dataset.
density is defined by $nnz/(I \times J \times K)$.

the other tensors studied in this work. Delicious is a *user-item-tag* dataset originally crawled by Görlitz et al. [47] and is also available for download.

4.5.2 Results

Effects of Fiber Direction Selection

SPLATT chooses at runtime which direction to store fibers in each of its modes. For example, the slices of the first mode can either have fibers that run along the second or third mode and the slices of the second mode will follow either the first or third mode. This is analogous to determining whether the sparse matrix representing each slice is stored in a row or column major format. Each fiber comes with some storage overhead and the number of saved FLOPs is dependent on the number of non-zeros per fiber. When there is a large disparity between the dimensions of $\boldsymbol{\mathcal{X}}$, choosing to have fewer, longer fibers is beneficial.

We evaluated this optimization on each of our datasets and present results in Table 4.2. SPLATT requires less memory when storing fibers along the longer dimensions for all tested datasets. Additionally, faster runtimes are exhibited on all datasets except BrainQ, in which shorter fibers had a 1.10× speedup. Speedup peaks at 1.45× on Netflix.

Table 4.2: Difference in storage requirements and runtime for the mode-1 slices of each dataset.

	Storage (Improvement)		**Time (Speedup)**	
Dataset	**Short**	**Long**	**Short**	**Long**
Netflix	7.75	5.02 (1.54×)	8.77	6.02 (1.45×)
NELL-1	11.91	8.88 (1.34×)	25.74	19.83 (1.29×)
NELL-2	4.32	3.69 (1.17×)	3.18	2.78 (1.14×)
BrainQ	0.54	0.50 (1.08×)	0.28	0.31 (0.90×)
Delicious	9.28	8.23 (1.12×)	17.66	15.61 (1.13×)

Short and **Long** refer to storing three copies of the tensor using fibers along the shortest or the longest modes, respectively. Storage is measured in gigabytes. Runtime is the average time in seconds to perform an execution of MTTKRP in all three modes. Storage and runtime for Short is measured relative to Long. No cache tiling is used. × denotes improvement over Short.

Effects of Tensor Reordering and Cache Tiling

To evaluate our methods of improving cache performance we measured runtime of SPLATT across orderings and tile sizes. The baseline is a randomly permuted tensor without tiling. Since reordering will only offer speedup on the most sparse tensors, we omitted BrainQ from the reordering experiments. Times reported are the average time of executing SPLATT with one thread across all three modes. Results are shown in Table 4.3. Delicious saw the largest benefit and reached 1.24× speedup after a mode-dependent reordering.

We found that reordering alone is not sufficient for maximizing performance. On all datasets, the best parallel speedups were found using a combination of reordering and cache tiling. The best results that we achieved using 16 threads are shown in Table 4.4. Note that these configurations are the most *scalable* configurations and not necessarily the fastest at small numbers of threads. This is because tiling increases arithmetic operations and the memory footprint of the tensor due to fibers being split across boundaries. After tiling we found the runtimes of mode independent and mode dependent reorderings to be similar, with mode-independent reorderings slightly faster.

Datasets with modes of relatively small dimension (BrainQ, Netflix, and NELL-2) saw benefit from cooperative tiling with up to a 1.22× speedup on BrainQ compared to traditional tiling. The number of synchronizations and reductions scale with the mode dimensions and thus large datasets such as NELL-1 and Delicious saw impaired scalability when using cooperative tiling. We experimentally found that tiles of dimension 2048×2048×4096 gave the best performance when executing cooperatively and tiles of dimension 32×1024×1024 gave the best performance when distributing entire layers to threads.

4.6 Evaluating Memory-efficient MTTKRP

We now evaluate the trade-offs present when using a single CSF representation instead of N, as was presented in Section 4.5.

Table 4.3: Effects of Tensor Reordering.

	Time (Speedup)		
Dataset	**Random**	**Mode-Independent**	**Mode-Dependent**
Netflix	6.02	5.26 (1.14×)	5.43 (1.10×)
NELL-1	19.83	17.83 (1.11×)	17.55 (1.12×)
NELL-2	2.78	2.61 (1.06×)	2.60 (1.06×)
Delicious	15.61	13.10 (1.19×)	12.51 (1.24×)

Runtime is the average time to perform a serial execution of MTTKRP across all three modes. When reordering, the number of partitions was scaled from 32 to 1024 and the best result used. Time is measured in seconds. × denotes speedup over a random ordering.

Table 4.4: Effects of Cache Tiling.

	Time (Speedup)			
Threads	**Baseline**	**tiled**	**MI+tiled**	**MD+tiled**
1	8.14 (1.0×)	8.90 (0.9×)	8.70 (1.0×)	9.18 (0.9×)
2	4.73 (1.7×)	4.88 (1.7×)	4.37 (1.9×)	4.52 (1.8×)
4	2.54 (3.2×)	2.58 (3.2×)	2.29 (3.6×)	2.35 (3.5×)
8	1.42 (5.7×)	1.41 (5.8×)	1.26 (6.5×)	1.26 (6.4×)
16	0.90 (9.0×)	0.85 (9.5×)	0.74 (11.0×)	0.75 (10.8×)

Time is measured in seconds and is the geometric mean across all datasets. **MI** and **MD** are mode-independent and mode-dependent reorderings, respectively. When reordering, the number of partitions was scaled from 32 to 1024 and the best result used. × denotes speedup over a random ordering without tiling.

4.6.1 Experimental Methodology

Experimental Setup

The memory-efficient algorithms were implemented in SPLATT and evaluated with double-precision floating-point numbers and 64-bit integers. We used $F = 16$ for all experiments. Experiments were performed on an HP ProLiant BL280c G6 blade server with two oct-core E5-2670 Xeon processors running at 2.6 GHz.

During our discussion, we will refer to experiments on a single untiled tensor as CSF-M (i.e., CSF with mutexes) and a single tiled tensor as CSF-T (i.e, CSF with tiling). We compare CSF-M and CSF-T against two competing approaches. COORD is a direct implementation of Equation (3.1), which streams through the tensor non-zeros stored in coordinate format and performs MTTKRP in an elementwise fashion. We use the original fiber-centric kernel as a benchmark for compressed tensors, which stores N representations of the tensor. The fiber-centric algorithm with N CSF representations is denoted as SPLATT.

Datasets

The tensors evaluated in this section are detailed in Table 4.5. The evaluation features largely the same datasets as Section 4.5, but with two additions. Beer [48] and Amazon [49] are *user-item-word* tensors parsed from Beer Advocate and Amazon product reviews, respectively. We used Porter stemming [50] on review text during parsing.

4.6.2 Comparison of Storage Requirements

Figure 4.6 shows the storage required for the uncompressed (COORD) tensor and the compressed tensor storage formats. CSF-M consistently has the smallest memory footprint and averages 68% less memory than SPLATT and 42% less memory than COORD. CSF-M achieves its best compression on NELL-1, with CSF using 73% less memory than SPLATT. After adding storage overhead from tiling for 16 threads, CSF-T still uses 58% less memory than SPLATT and 24% less than COORD. Overall, CSF-M and CSF-T always use less memory than COORD and SPLATT.

Table 4.5: Summary of datasets.

Dataset	I	J	K	nnz	density
NELL-2	12K	9K	28K	77M	1.3e-05
Beer	33K	66K	960K	94M	4.4e-08
Netflix	480K	18K	2K	100M	5.4e-06
Delicious	532K	17M	3M	140M	6.1e-12
NELL-1	3M	2M	25M	143M	3.1e-13
Amazon	5M	18M	2M	1.7B	1.1e-10

nnz is the number of nonzero entries in the dataset. **density** is defined by $nnz/(I \times J \times K)$. **K**, **M**, and **B** stand for thousand, million, and billion, respectively.

4.6.3 Performance Benchmarks

We evaluate the performance of the memory-efficient CSF-based algorithms against SPLATT and COORD. We first examine performance on individual modes using the three parallel algorithms and compare against SPLATT and COORD timings on the same modes. We then examine total MTTKRP runtime.

Table 4.6 compares the runtimes of performing MTTKRP on the smallest tensor modes (the root nodes). Since CSF-ROOT is the SPLATT algorithm adapted to the CSF data structure, performance between the two methods is nearly identical. With the exception of the Netflix dataset, CSF-ROOT sees little benefit from tiling. Since tiles must be large, they offer little cache benefit and do not mitigate any locks. Netflix is an exception because its root and internal dimensions are small compared to the leaf dimension. Most of the computation is done during non-zero accumulation instead of propagating results back up the tree.

Table 4.7 shows the runtimes of CSF-INTL, which computes MTTKRP on the middle tensor modes (interior nodes). CSF-M outperforms SPLATT on half of the datasets. CSF-M and CSF-T both outperform SPLATT on the Amazon tensor with 1.7 billion non-zeros, with CSF-T reaching 2.3× speedup over parallel SPLATT. We attribute this to the single-CSF algorithms sometimes performing fewer FLOPs than SPLATT on

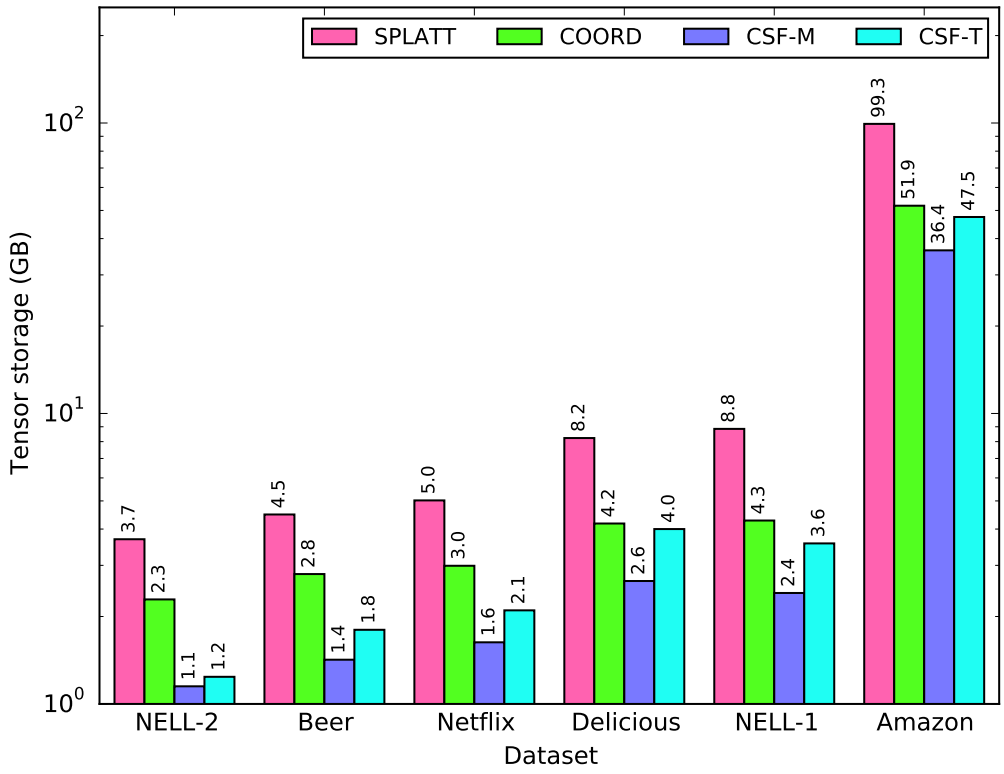


Figure 4.6: Required storage in gigabytes for each dataset, logarithmic scale.

non-root modes. This is possible because different views of the same tensor can result in differences in computation. For example, a *user-item-time* ratings tensor will have more non-zeros in the (*item, time*) fibers than (*user, item*) because users are unlikely to rate items more than once.

Tiling improves performance of CSF-INTL on three tensors. We only have to lock when writing to an internal node, and thus if there is a large amount of computation in the leaves of the tree it may be more beneficial to skip tiling.

CSF-LEAF runtimes are shown in Table 4.8. SPLATT is faster than CSF-M on all but the smallest tensor, NELL-2. Leaf computation operates on the longest mode of the tensor and is the most taxing MTTKRP step on the CPU memory hierarchy because each non-zero can result in a write to distant memory outside of the CPU cache. As expected, parallel scalability is poor without tiling because a mutex is locked and unlocked on every non-zero.

Table 4.6: CSF-ROOT with 16 threads.

Dataset	**COORD**	**SPLATT**	**CSF-M**	**CSF-T**
NELL-2	4.30	0.10	0.10	0.11
Beer	5.20	0.12	0.12	0.17
Netflix	15.46	0.23	0.24	0.18
Delicious	11.25	0.55	0.56	1.22
NELL-1	18.95	1.11	1.00	1.29
Amazon	96.04	3.61	3.71	6.18

Values are the best time in seconds to execute MTTKRP on the smallest mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Finally, we examine the total MTTKRP runtime in Table 4.9. CSF-M averages 40% as fast as SPLATT without tiling and 81% as fast with tiling enabled. CSF-T is comparable in performance to SPLATT even on our largest tensor, while using over 50 gigabytes less memory.

Table 4.7: CSF-INTL with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	4.35	0.08	0.15	0.11
Beer	5.24	0.12	0.17	0.19
Netflix	15.55	0.50	0.33	0.23
Delicious	15.66	1.15	0.96	1.27
NELL-1	24.22	1.10	1.18	1.73
Amazon	97.12	15.91	12.28	6.96

Values are the best time in seconds to execute MTTKRP on the interior mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Table 4.8: CSF-LEAF with 16 threads.

Dataset	COORD	SPLATT	CSF-M	CSF-T
NELL-2	4.49	0.10	1.21	0.10
Beer	5.23	0.17	1.71	0.21
Netflix	15.65	0.15	1.51	0.40
Delicious	15.06	1.07	2.43	2.30
NELL-1	18.44	1.58	3.16	3.01
Amazon	96.47	3.79	121.05	11.07

Values are the best time in seconds to execute MTTKRP on the longest mode of the tensor. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Table 4.9: Total MTTKRP iteration time with 16 threads.

Dataset	**COORD**	**SPLATT**	**CSF-M**	**CSF-T**
NELL-2	13.14	0.28	1.46	0.31
Beer	15.67	0.41	2.00	0.57
Netflix	46.66	0.89	2.08	0.81
Delicious	41.97	2.77	3.95	4.79
NELL-1	61.60	3.79	5.34	6.03
Amazon	289.64	23.31	137.03	24.21

Values are the best time in seconds to execute MTTKRP across all three modes. COORD is serial while SPLATT, CSF-M, and CSF-T all use 16 threads.

Chapter 5

Accelerating the CPD on Many-Core Systems

In order to harness the power of many-core processors, applications must expose a high degree of parallelism, load balance tens to hundreds of parallel threads, and effectively utilize the high-bandwidth memory. Algorithm developers must embrace these challenges and, in many cases, re-evaluate existing parallel algorithms.

This chapter details our method of obtaining high performance on many-core processors, with Intel Knights Landing (KNL) as a motivating architecture. This is a challenge that spans both high-level design and low-level implementation. The problem decomposition must expose a sufficient amount of parallelism, load balance hundreds of threads, and minimize fine-grained synchronization. Additionally, the implementation must utilize advanced hardware features such as vector instructions, efficient synchronization primitives, and MCDRAM.

5.1 Problem Decomposition for Many-Core Processors

5.1.1 Partial Tensor Tiling

The existing CSF-based algorithms presented in Chapter 4 use coarse-grained parallelism via distributing individual trees to threads. Computing with respect to the root mode has no race conditions to consider, as each root node ID is unique. There are no

uniqueness guarantees for levels below the root, and thus we must consider the case of threads overlapping additions to $\mathbf{K}(i_n, :)$, where n is a level below the root. Two solutions were proposed Chapter 4: a mutex pool can be indexed by node IDs to protect rows during updates; or $\boldsymbol{\mathcal{X}}$ can be tiled using a grid of dimension P^N, where P is the number of threads. Note that root nodes are no longer unique if tiling is used, and thus it must be performed on all N modes. Expensive synchronization is avoided by distributing the mode-n layers of tiles to threads.

This approach of decomposing the computations is limited in two major ways. First, coarse-grained parallelism is only effective when the tensor modes are sufficiently long. Many real-world tensors exhibit a combination of long, sparse modes and short, substantially more dense ones. For example, tensors used in context-aware recommendation will have many users but only a few contexts (e.g., time or location of purchase). Indeed, the performance evaluation by Rolinger et al. [51] showed that CSF-based computation is severely impacted by tensors with short modes. Second, tiling each tensor mode to avoid synchronization faces serious scalability issues when there are many threads or many tensor modes. For example, a six-mode tensor on a 68-core KNL system would require $68^6 \approx 99$ billion tiles. The alternative of relying solely on a mutex pool performs poorly even at small thread counts.

To address these problems, we propose a method that tiles a subset of the tensor modes and uses atomics for the remaining ones. In determining which modes to tile, priority is given to the longest modes for two reasons. First, they provide more opportunities for parallelism by permitting us to decompose the long tensor modes. As long as the tensor has at least one mode which is sufficiently long to distribute among P threads, then this decomposition is not hindered by the presence of short modes. Second, the longer modes contain more nodes in the CSF structure than shorter ones, and thus require more frequent synchronization. This intuition is supported in a previous evaluation which showed that the last modes (i.e., the lower levels of the CSF structure) experienced more overhead from fine-grained synchronization than the first modes (see Section 4.6.3).

Tiling long modes fits easily into the CSF framework because the modes are sorted by length. Our partial tiling is parameterized by h, referred to as the *tiling height*. The tiling height which defines a level in the CSF data structure and also how many modes

will be tiled. When we compute for a mode which resides above level h, atomics are used to prevent race conditions. At or below level h, tile layers are distributed to threads to avoid using atomics. The resulting tensor will have P^h tiles, where P is the number of threads being used. For example, h=0 results in an untiled tensor, h=1 results in a 1D decomposition on the longest mode, and h=N tiles each mode.

5.1.2 Multiple Tensor Representations

A single CSF representation is attractive because of its low memory overhead compared to N specialized representations. However, beyond the previously discussed synchronization challenges, an additional disadvantage of using a single CSF representation is less favorable writes to memory. Consider the difference between computing with respect to the first and Nth mode. During the first mode, $\mathbf{K}$ is written to once at the conclusion of each tree and $\mathbf{A}^{(N)}$ is read for each non-zero (see Section 4.3). In contrast, updating the Nth mode involves reading from $\mathbf{A}^{(1)}$ once at the start of each tree and updating $\mathbf{K}$ for each non-zero. Writing for each non-zero places significantly more pressure on memory bandwidth and cache coherency mechanisms. Furthermore, the updates to $\mathbf{K}$ follow the sparsity pattern and are generally scattered, which challenges the hardware prefetcher. Additionally, when $\mathbf{K}$ has few columns, only a few bytes out of a cache line may be utilized.

We propose to use two CSF representations when memory allows. The first CSF is as before, with the modes sorted and the shortest mode at the root level. The second CSF places the longest mode at the root level and sorts the remaining modes by non-decreasing length. When computing for the longest mode, we use the second CSF in order to improve access patterns. Since the longest mode is placed at the root level, we forego tiling on the second CSF in order to avoid any additional storage or computational overheads.

Note that this concept is not limited to one, two, or N representations. However, the combinations of CSF representations and orderings grows exponentially. Due to the combinatorial nature of the problem, we restrict our focus to either one, two, or N representations; each with modes sorted by length except for the specialized root.

5.1.3 Load Balancing the Computations

Many tensors have a non-uniform distribution of non-zeros. Algorithms which rely on distributing whole slices or tiles to many threads (i.e., a coarse-grained decomposition) can exhibit severe load imbalance. On the other hand, a fine-grained decomposition which distributes individual non-zeros can load balance a computation at the cost of frequent synchronizations.

We refer to a slice with a disproportionately large number of non-zeros as a *hub* slice. We call slice i a hub slice if

$$\text{nnz}(\boldsymbol{\mathcal{X}}(i,:,\ldots,:)) \geq \delta \left(\frac{\text{nnz}(\boldsymbol{\mathcal{X}})}{P} \right),$$

where δ is a user-supplied threshold. We empirically found δ=0.5 to be an effective value.

When a hub slice is identified during the construction of the CSF, it is not assigned to a thread. Instead, we evenly distribute all of its non-zeros among threads as a form of fine-grained parallelism. During the MTTKRP operation, all threads first process the set of non-hub slices in parallel using any synchronization construct as before. Second, each thread processes its assigned portion of the hub slices. By definition, there cannot be many hub slices relative to the total number of slices, and thus synchronization overheads are negligible.

5.2 Leveraging Architectural Features

KNL is the second generation Xeon Phi many-core processor from Intel [52]. KNL has up to 72 cores, each with two 512-bit vector processing units and 4-way simultaneous multi-threading. Cores have 32KB of L1 cache and are arranged in pairs which share 1MB of L2 cache.

KNL includes two types of memory: DDR4 and multi-channel DRAM (MCDRAM). MCDRAM offers up to 480GB/s achievable memory bandwidth measured by the STREAM benchmark [53], or approximately 4× the bandwidth of the latest 2-socket Xeon systems with DDR4. MCDRAM can be configured to either be explicitly managed by the software (*flat mode*), used as a last-level cache (*cache mode*), or a combination of the two (*hybrid mode*).

5.2.1 Vectorization

KNL (and other modern architectures) heavily rely on vectorization to achieve peak performance. KNL uses the new AVX-512 instruction set and has two 512-bit vector units per core. The MTTKRP formulation that we use accesses the tall, skinny factor matrices in a row-major fashion. Processing a node in the CSF structure requires $\mathcal{O}(F)$ operations. The micro-kernels are either in the form of (i) scaling a row by a non-zero value (i.e., a BLAS-1 *axpy*), or (ii) an element-wise multiplication of two rows (i.e., a Hadamard product). In practice, useful values of F will saturate at least one vector width. Therefore, we vectorize each of the micro-kernels.

5.2.2 Synchronization

Partial tiling of the tensor requires synchronization of some form to be used on the untiled tensor modes. The choice of synchronization primitive is heavily dependent on both hardware support and the characteristics of the data (e.g., whether conflicts are expected to be common). We will now discuss several options and evaluate them in Section 5.4.2.

Mutexes

The most simple synchronization strategy is to surround updates with a mutex. Most writes will be to unique data when the tensor mode is sufficiently large. We can maintain a pool of mutexes in order to reduce lock contention, but performance is still limited when mutexes have a high overhead in hardware. A challenge of mutexes is that we must tune the size of the pool to find the best trade-offs between storage overhead and contention.

Compare-and-Swap

CAS instructions are often optimized in hardware and bring no storage overhead unlike a mutex pool. Their limitation comes from the granularity of the protected memory. CAS is currently limited to 16 bytes on KNL and other popular architectures. Thus, four CAS instructions must be issued to utilize a full cache line (or full vector register) on a KNL system.

Transactional Memory

Modern architectures such as Intel's Haswell and Broadwell include hardware support for restricted transactional memory (RTM). While KNL does not include RTM, it is an efficient option for other parallel architectures.

Privatization

Hardware atomics may introduce a large overhead when the tensor mode is small and contention is probable. In that event, we instead allocate a thread-local matrix which is the same size as $\mathbf{K}$. Each thread accumulates into its own buffer without need for atomics. Finally, all buffers are combined with a parallel reduction. The memory overhead of privatization makes it only practical for short modes. We privatize mode n if

$$I_n P \leq \gamma \operatorname{nnz}(\boldsymbol{\mathcal{X}}), \tag{5.1}$$

where P is the number of threads and γ is a user-supplied threshold. We empirically found γ=0.2 to be effective.

5.2.3 Managing High-Bandwidth Memory

The 16GB capacity of MCDRAM on KNL is sufficient to factor some, but not all tensors. When the working set entirely fits in memory, explicitly managing MCDRAM and running in cache mode should offer similar performance.

When the working set exceeds the MCDRAM capacity, we prioritize placement of the factor matrices in MCDRAM. Each node in the CSF structure consumes $\mathcal{O}(1)$ memory but spawns $\mathcal{O}(F)$ accesses to the factors. In total, the CSF structure consumes $\mathcal{O}(\operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ bandwidth. When the tensor and factors exceed the size of MCDRAM, it is likely that the the factors do not fit in the on-chip caches and thus consume $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ bandwidth.

5.3 Experimental Methodology

5.3.1 Experimental Setup

We use two hardware configurations for experimentation. One machine has two sockets of 22-core Intel Xeon E5-2699v4 Broadwell processors, each with 55MB of last-level cache, and 128GB of DDR4 memory. The second machine has an Intel Xeon Phi Knights Landing 7250 processor with 68 cores, 16GB of MCDRAM, and 94GB of DDR4 memory. Throughout our discussion, we will refer to the dual-socket Broadwell machine as BDW and the Knights Landing machine as KNL. Importantly, KNL is a socketed processor and all application code runs directly on the hardware. Thus, there are no PCIe transfer overheads to consider. Unless otherwise specified, KNL is configured in flat mode with quadrant configuration.

Source code is written in C++ and modified from SPLATT v1.1.1, a library for sparse tensor factorization. We use double-precision floating point numbers, 64-bit integers for indexing non-zeros, and 32-bit integers for node IDs. The MTTKRP kernel is optimized with both AVX2 intrinsics for BDW and AVX-512 intrinsics for KNL. We use the Intel compiler version 17.0.0 with `-xCORE-AVX2` on BDW and `-xMIC-AVX512` on KNL, and Intel MKL for LAPACK routines. All source code is publicly available[1] . We set the environment variable `KMP_LOCK_KIND=tas` to use test-and-set locks [54].

Reported runtimes are the arithmetic mean of thirty iterations and error bars mark the standard deviation. Unless otherwise noted, we use F=16 for experiments.

5.3.2 Datasets

Table 5.1 details the tensors used in our evaluation. We selected tensors from a variety of real-world applications which extensively use the CPD. Outpatient is a six-mode *patient-institution-physician-diagnosis-procedure-date* tensor formed from synthetic electronic medical records [55]. Netflix [44] and Yahoo [56] are both *user-movie-date* tensors formed from movie ratings. Delicious is a *user-item-tag* tensor formed from user-supplied tags of websites [47]. NELL is a *noun-verb-noun* tensor from the Never Ending

[1] `https://github.com/ShadenSmith/splatt-ipdps17`

Table 5.1: Summary of datasets.

Dataset	NNZ	Dimensions	Size (GB)
Outpatient [55]	87M	1.6M, 6K, 13K, 6K, 1K, 192K	4.1
Netflix [44]	100M	480K, 18K, 2K	1.6
Delicious [47]	140M	532K, 17M, 3M	2.7
NELL [45]	143M	3M, 2M, 25M	2.4
Yahoo [56]	262M	1M, 624K, 133	4.3
Reddit [57]	924M	1.2M, 23K, 1.3M	15.0
Amazon [49]	1.7B	5M, 18M, 2M	36.4

NNZ is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively. **Size** is the amount of memory in gigabytes required to represent the tensor in a single CSF.

Language Learning project [45]. Reddit [57] is a *user-community-word* tensor representing a subset of user comments from Reddit[2] from 2007 to 2012. Amazon is a *user-item-word* tensor representing product reviews [49]. The Delicious, NELL, Reddit, and Amazon datasets are publicly available in the FROSTT collection [58].

5.4 Results

5.4.1 Exploring Decompositions on Many-Core Processors

We first explore the performance implications of problem decomposition on a many-core processor. In order to separate the effects of decomposition and KNL-specific hardware features, we explicitly place all allocations in DDR4 memory and use one thread per core. We use a pool of 1024 OpenMP mutexes for synchronization.

Partial Tiling

Figure 5.1 shows the effects of tiling one, two, and three modes with a single CSF representation according to the strategy described in Section 5.1. No strategy consistently

[2] `https://reddit.com/`

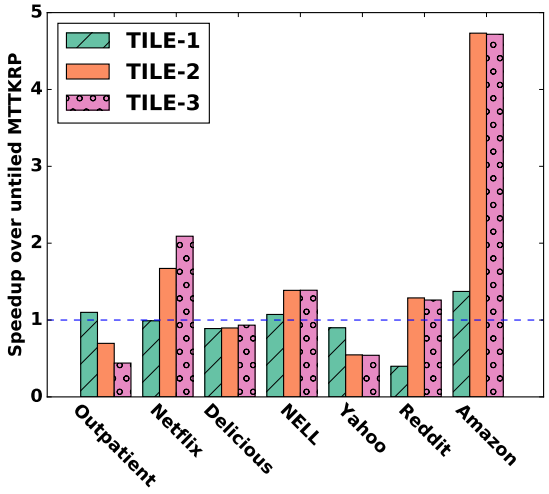


Figure 5.1: Speedup over untiled MTTKRP while tiling the longest (**Tile-1**), two longest (**Tile-2**), and three longest modes (**Tile-3**).

outperforms the others. Amazon sees the most benefit from tiling two and three modes, achieving a 4.5× speedup over the untiled implementation. The large disparity between tiling one and multiple modes of Amazon is not due to synchronization costs, but due to load imbalance. We further explore this challenge in Section 5.4.1.

Privatization

The Netflix, Outpatient, Reddit, and Yahoo tensors have a combination of long and short modes. The short modes result in lock contention. Figure 5.2 shows the effects of privatization (Section 5.2) as we change the number of tiled modes with a single CSF representation. The two- and three-mode tiling schemes see small performance gains from privatization, but tiling a single mode achieves significant speedups compared to untiled and also mutex-only synchronization (Figure 5.1). The slowdowns beyond a single tiled mode are attributed to the overheads of storing and operating with additional tiles. We use privatization with single-mode tiling for Netflix, Outpatient, Reddit, and Yahoo in the remaining experiments.

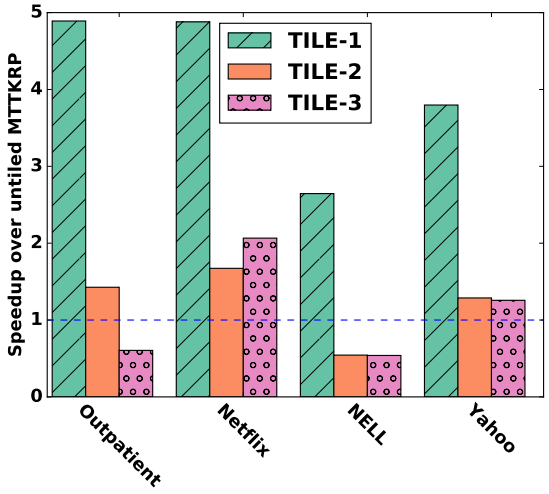


Figure 5.2: Speedup over untiled MTTKRP using one, two, and three tiled modes with privatization for synchronization. Privatized modes were selected per Equation (5.1) with γ=0.2.

Number of CSF Representations

Figure 5.3 shows MTTKRP's performance as we increase the number of CSF representations. We follow the scheme presented in Section 5.1 and use one, two, and N CSF representations. The tensors fall into two categories. Delicious and NELL benefit from a second and a third CSF representation, with diminishing returns after the second. The remaining tensors achieve the best performance with either one or two representations. We note that these are the four tensors that have highly skewed mode lengths. When a short mode is moved from the top CSF level, the resulting tree structure is changed and often achieves less compression than before. Since we have already improved performance on the skewed tensors through partial tiling and privatization, there is little to be gained from additional representations.

Load Imbalance

Table 5.2 shows load imbalance and runtime on the Amazon dataset with one tiled mode. We measure load imbalance as the relative difference between the maximum and

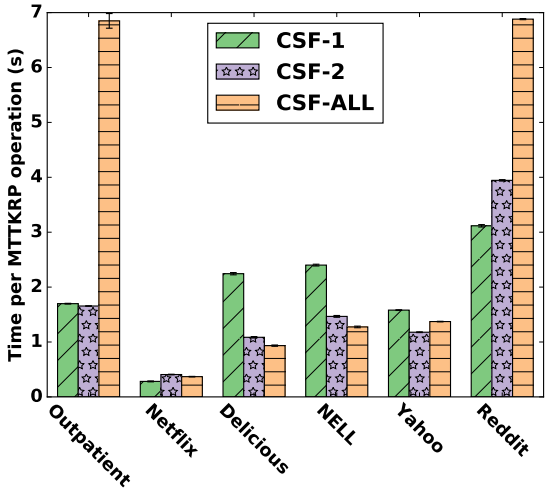


Figure 5.3: Effects of the number of CSF representations on MTTKRP runtime, using 1, 2, and N representations. Amazon is omitted due to memory constraints.

average time spent by all threads:

$$\text{imbalance} = \frac{t_{max} - t_{avg}}{t_{max}}.$$

The first mode of Amazon has a highly skewed distribution, with 6.5% of all non-zeros residing in a single slice. This overloaded slice prevents load balance for any coarse-grained (i.e., slice-based) parallelism.

BDW and KNL suffer from 72% and 84% load imbalance, respectively. When we switch to a fine-grained parallelism for the hub slices, load imbalance reduces to 4% and 5% on BDW and KNL, leading to 3.3× and 6.1× speedups, respectively. This resulting performance exceeds that of tiling with two and three modes.

5.4.2 Harnessing the KNL Architecture

We now explore performance optimizations specific to the KNL architecture. Unless otherwise noted, we work from the best decompositions learned in the previous experiments, which are summarized in Table 5.3. We note that every dataset in the evaluation benefits from at least one algorithmic contribution (i.e., partial tiling, privatization, multiple CSF representations, or hub slices).

Table 5.2: Load imbalance on the Amazon dataset.

	Load Imbalance				Time (s)			
	BDW		KNL		BDW		KNL	
Mode	**slice**	**hub**	**slice**	**hub**	**slice**	**hub**	**slice**	**hub**
1	0.72	0.04	0.84	0.05	2.37	0.71	3.22	0.53
2	0.13	0.04	0.05	0.03	1.31	0.79	0.73	0.72
3	0.07	0.03	0.24	0.18	2.67	2.61	1.96	1.82

Load imbalance is defined as the relative difference between the maximum and average time spent by all threads. **slice** denotes coarse-grained parallelism in which full slices are distributed to threads. **hub** denotes using fine-grained parallelism on "hub" slices and coarse-grained parallelism on all others.

MCDRAM

Figure 5.4 illustrates the benefits of MCDRAM over only DDR4 memory. We computed memory bandwidth by measuring the amount of data transferred from DDR4 or MCDRAM via hardware counters and divided this number by the time to compute MTTKRP [52].

Reddit and Amazon do not fit entirely inside of MCDRAM, and so we place only the factors inside of MCDRAM and measure the bandwidth from both memories. Interestingly, placing additional structures in MCDRAM (e.g., the tensor values or indices) does not improve performance due to KNL's ability to access DDR4 and MCDRAM concurrently. Any additional MCDRAM allocations simply increase the observed MCDRAM bandwidth while equally decreasing the observed DDR4 bandwidth, resulting in no net performance increase.

Outpatient is not memory-bound and sees little benefit from MCDRAM. We attribute this to its short mode lengths, which encourage temporal reuse. Additionally, Outpatient has a large number of modes, forcing it to incur high synchronization costs relative to the lower order tensors. We therefore omit it from the remaining MCDRAM evaluation.

When constrained to DDR4 memory, the remaining datasets are bounded by the maximum achievable bandwidth. MCDRAM increases the achieved bandwidth from

Table 5.3: Summary of the best known decompositions.

Dataset	Tiled Modes	CSF Reps.	Hub	Prv.
Outpatient	1	2		✓
Netflix	1	1		✓
Delicious	0	3	✓	
NELL	0	3	✓	
Yahoo	1	2		✓
Reddit	1	1		✓
Amazon	1	1	✓	

Tiled Modes is the number of tiled modes. **CSF** is the number of CSF representations. **Hub** indicates if there are any hub slices. **Prv.** indicates if we used privatization for at least one mode.

2.7× on Delicious to 3.7× on Netflix. The three datasets with the longest mode lengths (i.e., Delicious, NELL, and Amazon) are heavily dominated by read-bandwidth. NELL and Amazon achieve approximately 80% of the maximum 380 GB/s of read-bandwidth. The observed bandwidths do not fully saturate the MCDRAM's capabilities. We note that the MTTKRP time also includes computation which may not be overlapped with data movement, leading to an observed bandwidth which is lower than actually achieved. Thus, the presented bandwidth is a lower bound for the achieved bandwidth. A more detailed profiling or a formal performance analysis would be beneficial in determining the precise achieved bandwidth and whether MTTKRP is still bandwidth-bound in the presence of MCDRAM. We leave these tasks to future work.

The two-level memory hierarchy provided by KNL facilitates large scale computations which could not be performed in the presence of only MCDRAM. Reddit and Amazon demonstrate that a problem does not need to entirely fit in MCDRAM to obtain significant speedup, and instead we can focus on the bandwidth-intensive data structures for MCDRAM allocation.

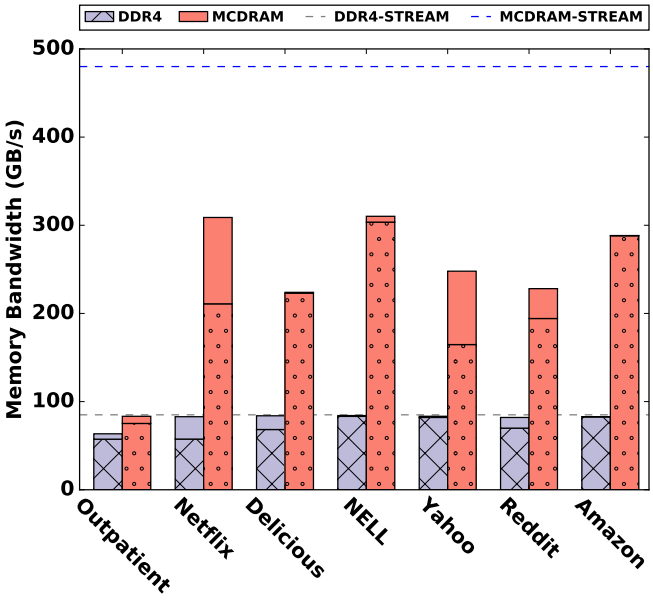


Figure 5.4: Observed memory bandwidth (BW) on KNL in flat mode with DDR4 and MCDRAM. Values indicate the maximum BW achieved for an MTTKRP kernel during CPD. Stacked bars encode read-BW (bottom) and write-BW (top). **DDR4-STREAM** and **MCDRAM-STREAM** indicate the maximum attainable read+write-BW per the STREAM benchmark [53]. KNL's maximum read-BW out of MCDRAM is 380 GB/s.

Synchronization Primitives

Figure 5.5 illustrates the overheads associated with various synchronization primitives during MTTKRP execution on KNL, compared to BDW. We report runtimes on Outpatient, which has the highest number of modes and also the shortest modes in our evaluation, and therefore the highest synchronization overheads. We do not use any tiling or privatization constructs in order to better evaluate the synchronization primitives provided by hardware. NOSYNC uses no synchronization and serves as a performance baseline. OMP uses a pool of 1024 OpenMP mutexes. 16B CAS uses 16B compare-and-swap (CAS) and 64B CAS simulates 64B CAS by issuing one 16B CAS for every 64B of data. RTM uses restricted transactional memory, which is available on BDW.

OMP and 16B CAS introduce over 100% overhead on KNL. The large vector instructions that AVX-512 offers are not well utilized with 16B CAS, as four CAS must

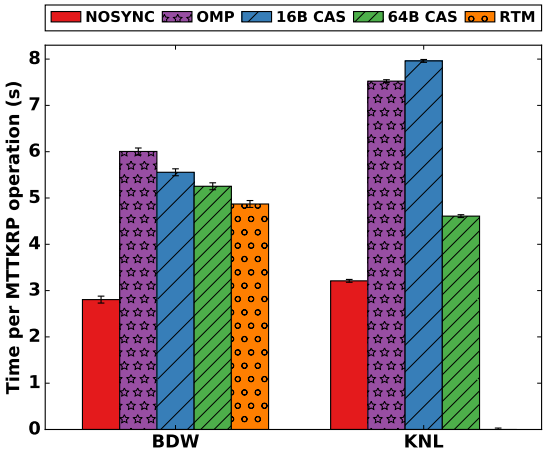


Figure 5.5: Comparison of synchronization primitives on the Outpatient dataset. All data is placed in MCDRAM on KNL.

be issued per fused multiply-add (FMA) instruction. The simulated 64B CAS reduces overhead to 30% due to KNL utilizing an entire AVX-512 FMA per CAS. Future many-core architectures could benefit significantly if CAS instructions are made wide enough to support the large vector registers.

RTM introduces the least overhead on BDW, including the simulated 64B CAS instructions. There is still a 42% overhead associated with RTM, however, suggesting that relying solely upon hardware-provided synchronization is insufficient for the best performance.

Simultaneous Multi-threading

KNL supports 4-way simultaneous multi-threading (SMT) as a method of hiding memory access latency. We examine the benefits of SMT in Figure 5.6. We execute in MCDRAM cache mode and run with 1, 2, and 4 threads per core for a total of 68, 136, and 272 threads. If a tensor is tiled, we fix the tile dimensions to be 272 and distribute additional tile layers to threads. Thus, each configuration performs the same amount of work and has the same sparsity structure. This allows us to eliminate the effects of varying decomposition while observing the effects of hiding latency.

Using two threads per core universally improves performance by masking access

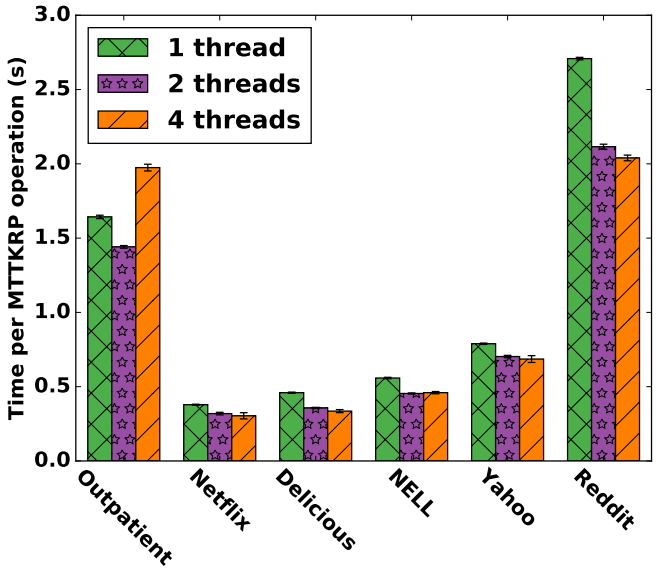


Figure 5.6: Evaluation of simultaneous multi-threading on KNL in cache mode. Amazon is omitted due to memory limitations when using four threads (due to the changed sparsity structure induced by tiling).

latencies. Performance is mixed beyond two threads due to increased synchronization costs resulting from lock contention or larger data reductions when using privatization. In the worst case, Outpatient spends 3.5× more time on synchronization. We note that load imbalance is not affected due to the same decomposition being used across thread configurations. We recommend using two threads per core due to its consistent benefit.

5.4.3 Comparing BDW and KNL

Figure 5.7 shows the performance of best-performing decompositions on KNL and BDW. We include KNL in both cache and flat mode configurations.

Observe that flat mode is up to 30% faster than cache mode when the dataset does not fit in MCDRAM. The tensor is accessed in a streaming fashion and exhibits no temporal locality, but still may be placed in the MCDRAM cache. By fixing the matrix factors in MCDRAM, which do exhibit temporal locality, we can ensure better utilization of the valuable MCDRAM resource.

KNL ranges from 0.84× slowdown on Reddit to 1.24× speedup on Amazon over

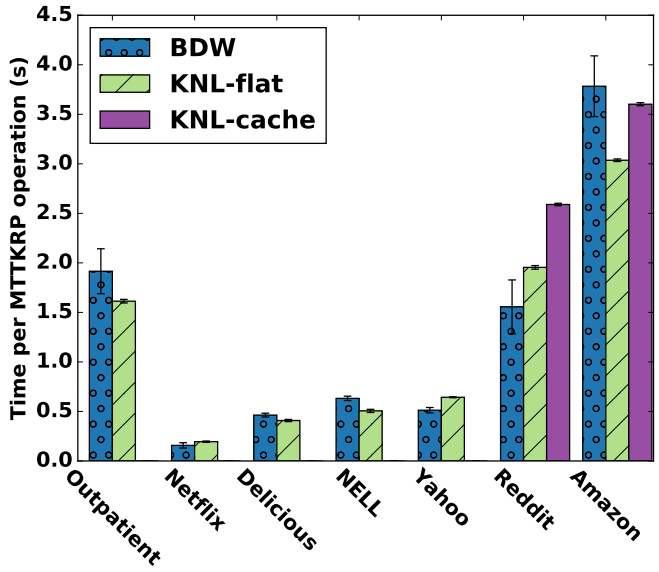
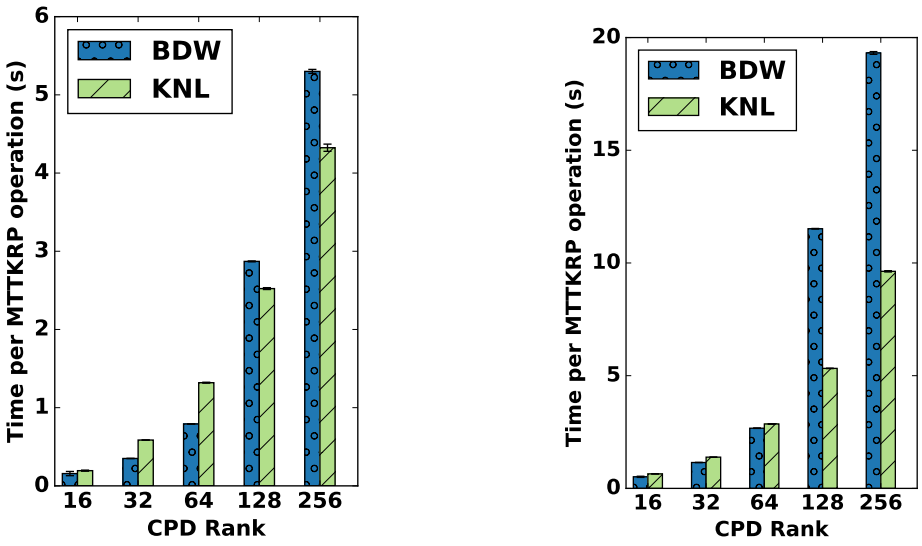


Figure 5.7: Comparison of MTTKRP performance on KNL and BDW. **KNL-flat** and **KNL-cache** denote KNL in flat and cache mode, respectively. Datasets which fit entirely in MCDRAM have identical running times in cache and flat mode and are thus omitted.

BDW. Unsurprisingly, we can see that KNL is most advantageous on the large, sparse tensors which are bandwidth-bound and benefit the most from MCDRAM. The last-level cache (LLC) of BDW allows it to outperform KNL on tensors with short modes. Netflix, for example, only requires 64MB to store all three factors. We explore larger CPD ranks with Netflix and Yahoo in Figure 5.8. In both cases, BDW is either faster or competitive to KNL for the smaller ranks due to the small factors mostly fitting in BDW's large LLC. BDW sees a sharp slowdown between ranks 64 and 128 as the factors no longer fit in LLC. KNL then proceeds to outperform BDW up to 1.8× due to MCDRAM's larger capacity.



(a) Netflix (b) Yahoo

Figure 5.8: Effects of increasing CPD rank on MTTKRP.

Chapter 6

Accelerating the CPD on Distributed-Memory Systems

State-of-the-art distributed CPD algorithms such as DFacTo [26] use *coarse-grained* decompositions in which independent one-dimensional (1D) decompositions are used for each tensor mode. Processes own a set of contiguous slices for each mode and are responsible for the corresponding factor rows. Figure 6.1 is an illustration of this decomposition scheme. An advantage of this scheme is the simplicity of performing MTTKRP operations. Each process owns all of the non-zeros that contribute to its owned output and thus the only communication required is exchanging updated factor rows after each iteration. Independent 1D decompositions can be interpreted as a task decomposition on the problem output, often called the *owner-computes rule*.

A limitation of these coarse-grained methods is that by owning slices in each mode of the tensor, processes own non-zeros that can span the complete modes of $\boldsymbol{\mathcal{X}}$. As a result, from the MTTKRP formulation (i.e., Equation (3.1)) we can see that processes will require access to the factors in their entirety over the course of the MTTKRP operations during an ALS iteration. The memory footprint of all factors can rival that of the entire tensor when the input is sparse. Thus, memory consumption is not scalable and since updated factors must be communicated, communication is also not scalable.

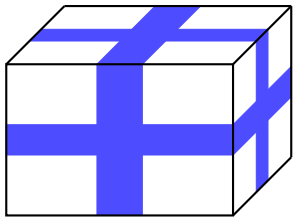


Figure 6.1: A coarse-grained decomposition of a sparse tensor. Slices owned by a single process are shaded.

6.1 Medium-Grained CPD-ALS

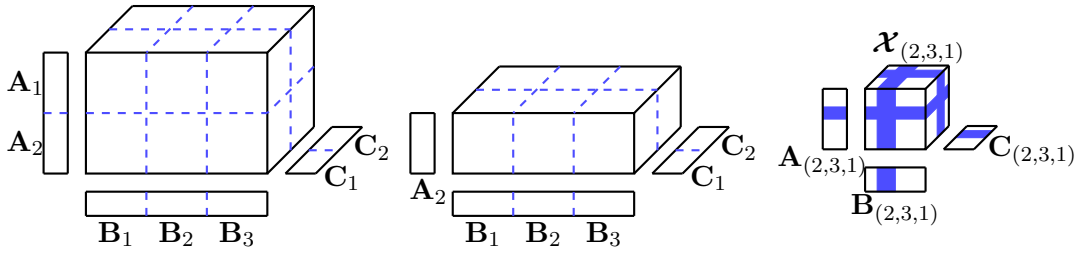
In order to address the high memory and communication requirements of the coarse-grained decomposition while at the same time eliminate the need to perform the expensive pre-processing step associated with hypergraph partitioning, we developed an approach that uses a medium-grained decomposition. Like coarse- and fine-grained methods, medium-grained have roots in the sparse matrix community [59, 60, 61]. The medium-grained decomposition uses an N-mode decomposition over the tensor and related 1D decompositions on the factor matrices. The medium-grained CPD-ALS algorithm is parallelized at the process-level using a message passing model and exploits multi-core architectures as well with thread-level parallelism on each node.

In order to simplify the presentation, this section considers only three mode tensors and the generalization of the algorithms to higher-order tensors is discussed in Section 6.1.4.

6.1.1 Data Distribution Scheme

Assume that there are $p = q \times r \times s$ processing elements available. We form a 3D decomposition of $\boldsymbol{\mathcal{X}}$ by partitioning its three modes into q, r, and s chunks, respectively. The intersections of these partitions form a total of p partitions arranged in a $q \times r \times s$ grid. We denote $\boldsymbol{\mathcal{X}}_{(x,y,z)}$ as the partition of $\boldsymbol{\mathcal{X}}$ with coordinate (x, y, z), and $p_{(x,y,z)}$ as the process that owns $\boldsymbol{\mathcal{X}}_{(x,y,z)}$. We refer to a group processes which share a coordinate as a *layer*. For example, $p_{(i,:,:)}$ is a layer of $r \times s$ processes along the first mode and $p_{(:,j,:)}$ is layer of $q \times s$ processes along the second mode.

In our implementation, each process stores its subtensor in the CSF data structure.



(a) (b) (c)

Figure 6.2: A medium-grained decomposition for twelve processes. (a) $\boldsymbol{\mathcal{X}}$ is distributed over a 2×3×2 grid. (b) Process layer $p_{(2,:,:)}$ collectively owns $\mathbf{A}_2$. (c) Process p_7 owns $\boldsymbol{\mathcal{X}}_{2,3,1}$ and the shaded factor rows.

This allows us to use the operation-efficient MTTKRP algorithm included in SPLATT to extract parallelism on shared-memory architectures.

We use the 3D decomposition of $\boldsymbol{\mathcal{X}}$ to induce partitionings of the rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. The rows of $\mathbf{A}$ are divided into chunks $\mathbf{A}_1, \ldots, \mathbf{A}_q$ which have boundaries aligned with the q partitions of the first mode of $\boldsymbol{\mathcal{X}}$. The rows in $\mathbf{A}_i$ are collectively owned by all processes in layer $p_{(i,:,:)}$. The rows of $\mathbf{B}$ and $\mathbf{C}$ are similarly divided into $\mathbf{B}_1, \ldots, \mathbf{B}_r$ and $\mathbf{C}_1, \ldots, \mathbf{C}_s$, respectively. This decomposition is illustrated in Figure 6.2a.

We further partition the rows of each chunk of $\mathbf{A}$ into $r \times s$ groups such that each process in layer $p_{(i,:,:)}$ owns a subset of the rows of $\mathbf{A}_i$. We note that the partitioning need not assign a contiguous set of rows to a process and a process is not required to be assigned any rows. The output of the MTTKRP operation, $\mathbf{K}$, has the same distribution as $\mathbf{A}$. Process p_i owns the same rows of $\mathbf{K}_i$ as it does $\mathbf{A}_i$. The process is repeated for $\mathbf{B}$ and $\mathbf{C}$ similarly. We relabel the slices of $\boldsymbol{\mathcal{X}}$ in order to make the rows owned by each process contiguous. This is illustrated in Figure 6.2c.

In subsequent discussions we will refer to process-level partitions of $\mathbf{A}$ in two ways: $\mathbf{A}_{p_i}$ refers to the chunk of $\mathbf{A}$ owned by process p_i, and $\mathbf{A}_{(x,y,z)}$ refers to the chunk of $\mathbf{A}$ owned by process with coordinate (x, y, z). The coordinate form will simplify discussion during the MTTKRP operation that relies on the 3D decomposition.

6.1.2 Distributed CPD-ALS

We will now detail each step of a CPD-ALS iteration using our 3D decomposition. For brevity we only discuss the computations used for the first mode. The other tensor modes are computed identically.

6.1.3 Distributed MTTKRP Operations

Process $p_{(x,y,z)}$ performs an MTTKRP operation with $\boldsymbol{\mathcal{X}}_{(x,y,z)}$. Any non-zeros in $\boldsymbol{\mathcal{X}}_{(x,y,z)}$ whose mode-1 indices are non-local will produce partial products that must be sent to other processes in the layer $p_{(x,:,:)}$. Likewise, $p_{(x,y,z)}$ will receive partial products from any processes in layer $p_{(x,:,:)}$ which output to rows in $\mathbf{K}_{(x,y,z)}$. The received partial products are then aggregated, resulting in the completed $\mathbf{K}_{(x,y,z)}$.

Cholesky Factorization

$\mathbf{B}^\intercal\mathbf{B}$ and $\mathbf{C}^\intercal\mathbf{C}$ are $F \times F$ matrices that comfortably fit in the memory of each process. Assume $\mathbf{B}^\intercal\mathbf{B}$ and $\mathbf{C}^\intercal\mathbf{C}$ are already resident in each process' memory. All processes redundantly compute the Cholesky factorization of $\mathbf{M} = (\mathbf{C}^\intercal\mathbf{C} * \mathbf{B}^\intercal\mathbf{B})$ in $\mathcal{O}(F^3)$ time, which is a negligible overhead for the low-rank problems that we are interested in. We perform the forward and backward substitutions in block form to exploit our row-wise distribution of $\mathbf{K}$:

$$\mathbf{A}^\intercal = \mathbf{M}^{-1}\mathbf{K}^\intercal = \begin{bmatrix} \mathbf{M}^{-1}\mathbf{K}_{p_1}^\intercal & \mathbf{M}^{-1}\mathbf{K}_{p_2}^\intercal & \dots & \mathbf{M}^{-1}\mathbf{K}_{p_p}^\intercal \end{bmatrix}$$

Forming the New Gram Matrix

Each process needs the updated $\mathbf{A}^\intercal\mathbf{A}$ factor in order to form $\mathbf{M}$ during the proceeding modes. We view the block matrix form of the computation to derive a distributed algorithm:

$$\mathbf{A}^\intercal\mathbf{A} = \begin{bmatrix} \mathbf{A}_{p_1}^\intercal & \mathbf{A}_{p_2}^\intercal & \dots & \mathbf{A}_{p_p}^\intercal \end{bmatrix} \begin{bmatrix} \mathbf{A}_{p_1} \\ \mathbf{A}_{p_2} \\ \dots \\ \mathbf{A}_{p_p} \end{bmatrix} = \sum_{i=1}^{p} \mathbf{A}_{p_i}^\intercal \mathbf{A}_{p_i}.$$

Each process first computes its local $\mathbf{A}_{p_i}^\intercal \mathbf{A}_{p_i}$. The 1D decomposition on the rows of $\mathbf{A}_{p_i}$ is used again to extract thread-level parallelism. We then perform an All-to-All reduction to find the final matrix and distribute it among all processes.

Updating Non-Local Rows

Processes with non-local rows of $\mathbf{A}$ must receive updated values before the next MTTKRP operation. This communication is a dual of exchanging partial products during the distributed MTTKRP operation. Any processes that sent partial MTTKRP products to process p_i now receive the updated rows of $\mathbf{A}_{p_i}$.

Residual Computation

Convergence is tested at the end of every iteration. In Section 2.3.1 we showed that residual computation cost is bounded by $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle$, which uses $4F \cdot \text{nnz}(\boldsymbol{\mathcal{X}})$ FLOPs. We observe that contributions from $\mathbf{B}$ and $\mathbf{C}$ with $\boldsymbol{\mathcal{X}}$ are already computed during the MTTKRP operation. Thus, we can cache $\mathbf{K}$ and rewrite Equation (2.4) as

$$\mathbf{1}^\intercal \begin{bmatrix} \mathbf{A}_{p_1} * \mathbf{K}_{p_1} \\ \mathbf{A}_{p_2} * \mathbf{K}_{p_2} \\ \cdots \\ \mathbf{A}_{p_p} * \mathbf{K}_{p_p} \end{bmatrix} \mathbf{1} = \sum_{i=1}^{p} \mathbf{1}^\intercal \left(\hat{\mathbf{A}}_{p_i} * \mathbf{A}_{p_i} \right) \mathbf{1}, \tag{6.1}$$

where $\mathbf{1}$ is the vector of all ones. This reduces the computation to $2IF$ FLOPs.

Each process computes its own local $\mathbf{1}^\intercal \left(\hat{\mathbf{A}}_{p_i} * \mathbf{A}_{p_i} \right) \mathbf{1}$. Thread-level parallelism is achieved via 1D row decompositions on $\hat{\mathbf{A}}_{p_i}$ and $\mathbf{A}_{p_i}$. Finally, we use a parallel reduction on each node's local result and form $\langle \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Z}} \rangle$.

6.1.4 Extensions to Higher-Order Tensors

Extending our distributed CPD-ALS algorithm to tensors with an arbitrary number of modes is straightforward. Suppose $\boldsymbol{\mathcal{X}}$ is a tensor with N modes and we wish to compute factors $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$.

Operation-efficient MTTKRP algorithms for a general number of modes are found in Chapter 4. Adding partial products from neighbor processes remains the same, with

the only consideration being that a *layer* is no longer a 2D group of processes, but a group of dimension $N-1$.

Residual computation again is easily extended. Generalized MTTKRP computes

$$\mathbf{K}(i_1, f) = \sum \boldsymbol{\mathcal{X}}(i_1, \ldots, i_N) \prod_{n=2}^{N} \mathbf{A}^{(n)}(i_n, f),$$

and so we can directly use Equation (6.1) to complete the residual calculation. Assuming $\mathbf{K}$ can be cached, our algorithm does not increase in cost as more modes are added.

6.1.5 Complexity Analysis

The cost of CPD-ALS is bounded by MTTKRP and its associated communication. Coarse-, medium-, and fine-grained distributed algorithms distribute work such that each process does $\mathcal{O}(F \cdot \text{nnz}(\boldsymbol{\mathcal{X}})/p)$ work. They differ, however, in the overheads associated with communication. In this section, we discuss the communication costs present in coarse- and medium-grained decompositions for a single mode. We define the communication cost as the number of words of $\mathbf{A}$ and $\mathbf{K}$ that must be communicated. The flexibility of fine-grained decompositions makes analysis difficult; both coarse- and medium-grained communication patterns are possible if the non-zeros are distributed appropriately. In our discussion we will use the personalized all-to-all collective communication. Derivation of its complexity can be found in [62].

Assume that $\boldsymbol{\mathcal{X}}$ has N modes, is of dimension $I \times \ldots \times I$, p processes are arranged in a $\sqrt[N]{p} \times \ldots \times \sqrt[N]{p}$ grid, and that messages require $\mathcal{O}(1)$ time to transfer per word. A medium-grained decomposition has two communication steps to consider: aggregating non-local rows during an MTTKRP operation and sending updated rows of $\mathbf{A}_{p_i}$ after an iteration.

In the worst case, every process has non-zeros in all $(I/\sqrt[N]{p})$ slices of the layer. A process must send (I/p) unique rows of $\mathbf{K}$ to each of its neighbors in the layer. Using a personalized all-to-all collective, this communication is accomplished in time

$$\frac{IF}{p}\left(p^{\frac{N-1}{N}} - 1\right) = \left(\frac{IF}{\sqrt[N]{p}} - \frac{IF}{p}\right) = \mathcal{O}\left(\frac{IF}{\sqrt[N]{p}}\right). \tag{6.2}$$

The worst case of the update stage is sending (I/p) rows to each of the $p^{\frac{N-1}{N}}$ neighbors in the layer. This operation is the dual of Equation (6.2) and has the same cost.

In comparison, a coarse-grained decomposition will send up to (I/p) rows to all p processes. The communication overhead is thus

$$\frac{IF}{p}(p-1) = \mathcal{O}(IF). \tag{6.3}$$

No partial results from an MTTKRP operation need to be communicated, however, so Equation (6.3) is the only communication associated with a coarse-grained decomposition. Comparing Equation (6.2) and Equation (6.3) shows that only the medium-grained decomposition can reduce communication costs by increasing parallelism. We experimentally evaluate this observation in Section 6.3.2.

6.1.6 Computing the Data Decomposition

Our discussion so far has provided an overview of our medium-grained data decomposition and a distributed algorithm for CPD-ALS. There are two forms of overhead that an ideal data decomposition will minimize: load imbalance and communication volume. Graph and hypergraph partitioners co-optimize these objectives, but can require significant pre-processing. We chose to optimize the objectives separately. We load balance the computation during the tensor decomposition because computational load is mostly a function of the number of non-zeros assigned to a process. Communication volume is optimized during the decomposition of the factor matrices because the assignment of rows directly impacts communication.

Finding a Balanced Tensor Decomposition

Our objective is to derive a load balanced $q \times r \times s$ decomposition of the modes of $\boldsymbol{\mathcal{X}}$. We begin by randomly permuting the each mode of the tensor. The purpose of the random permutation is to remove any ordering present from the data collection process that could result in load imbalance. Each mode is then partitioned independently.

The decomposition of the first mode into q parts is determined as follows: We greedily assign partition boundaries by adding consecutive slices until a partition has at least $\text{nnz}(\boldsymbol{\mathcal{X}})/q$ non-zeros. We call $\text{nnz}(\boldsymbol{\mathcal{X}})/q$ the *target* size of a partition because it will result in a load balanced partitioning of the mode. Slices can vary in density and adding a slice with many non-zeros can push a partition significantly over the target

size. Thus, after identifying the slice which pushes a partition over the target size we compare it to the slice immediately before and choose whichever leads to better balance.

Each of the independent mode decompositions is an instance of the chains-on-chains partitioning problem, for which there are fast exact algorithms [63]. We found that in practice, using optimal partitionings led to higher load imbalance than greedily choosing sub-optimal partitionings. Since we ultimately work with the *intersection* of the 1D partitionings, having optimality in each dimension does not guarantee optimality in the final partitioning.

Partitioning the Factor Matrices

A process may have non-zeros whose indices correspond to factor rows which are not owned by the process itself. These non-local rows must be communicated. Thus, the partitioning of rows during the sub-division of $\mathbf{A}_i$ directly affects the number of partial results which are exchanged during the MTTKRP operation. Our objective is to minimize the total number of communicated rows, or the *communication volume*. We adapt a greedy method of assigning rows developed for two-dimensional sparse matrix-vector multiplication [61]. We again partition each mode independently.

The sub-division of $\mathbf{A}$ is determined as follows: The q chunks of of $\mathbf{A}$ are partitioned independently. For each row i_r in chunk $\mathbf{A}_i$, processes count the number of tensor partitions (and thus, processes) that contain a non-zero value in slice $\boldsymbol{\mathcal{X}}(i_r, :, :)$. Any row that is found in only a single partition is trivially assigned to the owner because it will not increase communication volume. Next, the master process in the layer $p_{(i,:,:)}$ coordinates the assignment of all remaining rows. At each step it selects the processes with the two smallest communication volumes, p_j and p_k, with p_j having the smaller volume. The master process sends a message to p_j instructing it to claim rows until its volume matches p_k. Processes first claim indices which are found in their local tensor and only claim non-local ones when options are exhausted. The assignment procedure sometimes reaches a situation in which all processes have equal volumes but not all rows have been assigned. To overcome this obstacle we instruct the next process to claim a $1/(r \times s)$ fraction of the remaining rows.

These steps are then performed on the second and third tensor modes to complete the decomposition.

Algorithm 10 Deriving the decomposition shape

Input: dims, the dimensions of $\boldsymbol{\mathcal{X}}$; m, the number of modes in $\boldsymbol{\mathcal{X}}$; p, the number of processes.

Output: P, a vector storing the decomposition dimensions

$F \leftarrow$ the prime factors of p in non-increasing order
$P \leftarrow \mathbf{1}$, an m-dimensional vector of ones
▷ Find the optimal number of slices per process.
target $\leftarrow \left(\sum_{i=1}^{m} \text{dims}[i]\right) / p$
for all $f \in F$ **do** ▷ Assign a factor of p at a time
    distances $\leftarrow \mathbf{0}$, an m-dimensional vector of zeros
    ▷ Find the mode with the most work per process
    for $i \leftarrow 1$ to m **do**
        distances$[i] \leftarrow (\text{dims}[i]/P[i]) - \text{target}$
    end for
    furthest $\leftarrow \text{argmax}_i \text{distances}[i]$
    $P[\text{furthest}] \leftarrow P[\text{furthest}] \times f$ ▷ Give f processes
end for

Choosing the Shape of the Decomposition

Our decomposition does not require an equal number of processes along each mode. We select at runtime the number of processes that should be assigned to each mode. Most tensors will feature one or more modes that are significantly longer than the others. For example, the Netflix tensor described in Section 6.2 has over 20× more users than it does films. When choosing the dimensions for the decomposition, it is advantageous to assign more processes to the long modes than the short ones. The reasoning behind this decision is that short modes are likely to require storage and communication regardless of the decomposition and we should instead use more processes to further the decompose the modes which can benefit.

A constraint we impose when computing the decomposition is that the product of the dimensions must equal the number of processes, i.e., $q \times r \times s = p$. To achieve this, we break p into its prime factors and greedily assign them to modes. This process is detailed in Algorithm 10.

6.2 Experimental Methodology

6.2.1 Experimental Setup

We used SPLATT to implement three versions of distributed CPD-ALS. We refer to the collection of our implementations as DMS (distributed-memory SPLATT). The first version, DMS-CG, uses a coarse-grained decomposition and is a direct implementation of the algorithm used in SPLATT and adapted to distributed-memory systems. The second method uses a medium-grained decomposition described in Section 6.1 and is denoted DMS-MG. Our final implementation is DMS-FG, which follows the fine-grained tensor decomposition used in the evaluation of HyperTensor [27]. All three algorithms use the same computational kernels and only differ in decomposition and the resulting communications. DMS-CG and DMS-MG are implemented with personalized all-to-all collective operations, while DMS-FG uses point-to-point communications.

Zoltan [64] with PHG was used for hypergraph partitioning with LB_APPROACH set to "PARTITION". All hypergraphs were partitioned offline using 512 cores. Partitioning required between 1400 seconds on Netflix and 6400 seconds on Delicious.

DMS is implemented in C with double-precision floating-point numbers and 64-bit integers. DMS uses MPI for distributed memory parallelism and OpenMP for shared-memory parallelism. All source code is available for download[1] . Source code was compiled with GCC 4.9.2 using optimization level three.

We compare against DFacTo, which to our knowledge is the fastest publicly available tensor factorization software. DFacTo is implemented in C++ and uses MPI for distributed memory parallelism.

We used $F = 16$ for all experiments. Experiments were carried out on HP ProLiant BL280c G6 blade servers on a 40-gigabit InfiniBand interconnect. Each server had dual-socket, quad-core Xeon X5560 processors running at 2.8 GHz with 8MB last-level cache and 22 gigabytes of available memory.

6.2.2 Datasets

Table 6.1 is a summary of the datasets we used for evaluation. The Netflix dataset is taken from the Netflix Prize competition [44] and forms a *user-item-time* ratings tensor.

[1] `http://cs.umn.edu/~splatt/`

Table 6.1: Summary of datasets.

Dataset	I	J	K	nnz	storage (GiB)
Netflix	480K	18K	2K	100M	3.0
Delicious	532K	17M	3M	140M	4.2
NELL	3M	2M	25M	143M	4.3
Amazon	5M	18M	2M	1.7B	51.9
Random1	20M	20M	20M	1.0B	29.8
Random2	50M	5M	5M	1.0B	29.8

nnz is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively. **storage** is the amount of memory required to represent the tensor as $(i, j, k) = v$ tuples using 64-bit integers and 64-bit floating-point values.

NELL [45] is comprised of *noun-verb-noun* triplets. Amazon [49] is a *user-item-word* tensor parsed from product reviews. We used Porter stemming [50] on review text and removed all users, items, and words that appeared less than five times. Delicious is a *user-item-tag* dataset originally crawled by Görlitz et al. [47] and is also available for download. Random1 and Random2 are both synthetic datasets with non-zeros uniformly distributed. They have the same number of non-zeros and total mode length (i.e., output size), but differ in the length of individual modes.

6.3 Results

6.3.1 Effects of Distribution on Load Balance

Table 6.2 shows the load imbalance with 64 and 128 nodes. Load imbalance is defined as the ratio of the maximum amount of work (tensor non-zeros) assigned to a process to the average amount of work over all processes. DMS-CG suffers severe load imbalance on the Amazon tensor, with the imbalance growing from 2.17 with 64 nodes to 3.86 with 128 nodes. In contrast, DMS-MG has lower imbalance, with its largest ratios being only 1.08 with 64 nodes on Amazon. DMS-FG is the most balanced, with Zoltan reaching 1.05 on Delicious with 128 nodes and 1.00 on all other that datasets we could partition.

Table 6.2: Load imbalance with 64 and 128 nodes.

	DMS-CG		DMS-MG		DMS-FG	
Dataset	**64**	**128**	**64**	**128**	**64**	**128**
Netflix	1.03	1.18	1.00	1.00	1.00	1.00
Delicious	1.21	1.41	1.01	1.06	1.00	1.05
NELL	1.12	1.29	1.01	1.01	1.00	1.00
Amazon	2.17	3.86	1.08	1.08	part	part

Load imbalance is the ratio of the largest number of nonzeros assigned to a process to the average number of nonzeros per process. **part** indicates that we were unable to compute a hypergraph partitioning in the memory available on 64 nodes. Hypergraph partitioning was performed with the load imbalance parameter set to 1.10.

6.3.2 Effects of Distribution on Communication Volume

Table 6.3 presents results for communication volume with 128 nodes. We only count communication that is a consequence of the tensor decomposition, i.e., the aggregation of partial products during MTTKRP operations and exchanging updated rows. We report the average volume per MPI process as well as the maximum over all processes. We define the communication volume as the total number of rows sent and received per iteration, per MPI process. By measuring the total number of rows communicated, and not the number of *words*, our discussion is independent of the rank of the decomposition. When $F = 1$, the number of communicated rows is equal to the communicated words.

When the each process owns (I/p) rows of a factor, the worst case communication volume results from sending (I/p) rows to p processes and receiving $I - (I/p)$ rows for a total volume of $2I - (I/p)$. The maximum volume over all modes is

$$V_{max} = 2I + 2J + 2K - \frac{I + J + K}{p}.$$

DFacTo uses a pessimistic approach to communication and always has a communication volume of V_{max}. DMS-CG uses the same decomposition as DFacTo but instead utilizes an optimistic approach in which only the necessary factor rows are stored and communicated. Resultingly, DMS-CG has a smaller communication volume than V_{max} on all

Table 6.3: Communication volume with 128 MPI processes.

	DMS-CG		DMS-MG		DMS-FG	
Dataset	**max**	**avg**	**max**	**avg**	**max**	**avg**
Netflix	674.8K	616.9K	99.3K	56.8K	2.6M	210.5K
Delicious	2.8M	2.3M	2.5M	1.6M	4.2M	719.2K
NELL	3.8M	3.4M	2.5M	1.7M	6.0M	1.2M
Amazon	8.3M	7.3M	4.0M	2.5M	part	part
Random1	72.1M	72.1M	39.5M	39.3M	part	part
Random2	55.2M	55.2M	39.6M	23.5M	part	part

Table values are the communication volumes with 128 MPI processes. **max** is the maximum volume of any MPI process and **avg** is the average volume. **part** indicates that we were unable to compute a hypergraph partitioning in the memory available on 64 nodes.

datasets that we were able to collect results for. Despite the added communication step of aggregating partial results during the MTTKRP operations, DMS-MG and DMS-FG exhibit lower average communication volumes than DMS-CG on all datasets.

DMS-FG has the lowest average volume on all datasets except Netflix. The discrepancy between mode lengths is largest on Netflix, resulting in DMS-MG using a $64\times2\times1$ decomposition of the tensor. By using most of processes to partition only the longest mode, the majority of the possible communication volume is constrained to the $p_{(i,:,:)}$ layers which have only two processes each. DMS-MG avoids partitioning the other tensor modes in exchange for greatly reducing the communication along the longest mode.

While the average communication volumes are lowest with DMS-FG, this method also sees the largest maximum volumes. Hypergraph partitioners optimize the total communication volume, not necessarily the maximum over any process. Additionally, with fine-grained decompositions a process may have to exchange rows with all other processes instead of being bounded by the size of a layer. Thus, some processes can exhibit large communication volumes in exchange for a lower average.

6.3.3 Strong Scaling

Table 6.4 shows the runtimes of our methods and DFacTo. We scale from 2 to 128 computing nodes and measure the time to perform one iteration of CPD-ALS averaged over 50 runs. Each node has eight processors available which we utilize. DMS is a hybrid MPI+OpenMP code and so we use one MPI process and eight OpenMP threads per node. DFacTo is a pure MPI code and so we use eight MPI processes per node.

The DMS methods are faster than DFacTo on all datasets. DMS-MG is 41× faster on Amazon and 76× faster on Delicious when both methods use 128 nodes (1024 cores). Our success is due to several key optimizations. The three DMS methods begin faster on small node counts due to an MTTKRP algorithm which on average is 5× faster (Chapter 4). As we increase the number of nodes, DMS methods out-scale DFacTo due to their ability to exploit parallelism in the dense matrix operations that take place after the MTTKRP operation. DMS methods also use significantly less memory than DFacTo, which is unable to factor some of our large datasets. This is due to a combination of our optimistic factor storage and our MPI+OpenMP hybrid code. DFacTo must replicate factors on every core to exploit multi-core architectures. Even in the worst case, the DMS methods only need one copy of each matrix factor (and in practice, almost always less than one copy).

DMS-FG was unable to partition the tensors with billions of non-zeros due to the overhead of hypergraph partitioning. It is important to note that fine-grained decompositions are not limited to only the hypergraph model, and non-zeros could instead be randomly assigned to processes. However, experimental results in [27] show that random assignment results in runtime performance that is comparable to a coarse-grained decomposition. On the tensors we were able to factor, its performance is comparable to DMS-CG on Netflix and Delicious, but DMS-CG is 1.7× faster on NELL.

DMS-MG is the fastest method among the DMS implementations. It ranges from 1.3× to 8.0× faster than DMS-CG and 1.5× to 5.0× faster than DMS-FG In many cases, DMS-MG is able to factor tensors when other methods cannot due to memory limitations or the hypergraph partitioning overhead. Figure 6.3 graphs the strong scaling results for the Netflix dataset. DMS-MG maintains near-perfect speedup through 512 cores. Between 16 and 128 cores, DMS-MG achieves speedups which are super-linear. We attribute this behavior to the decomposition shape that DMS-MG chooses.

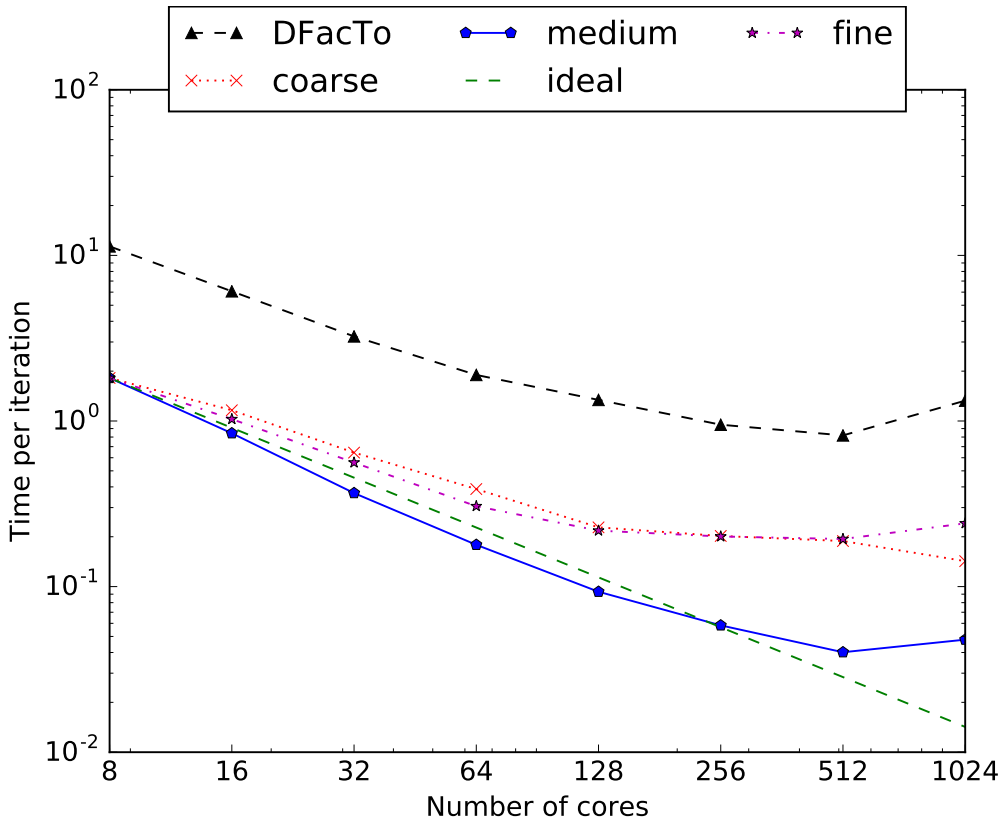


Figure 6.3: Average time per iteration in seconds on the Netflix dataset. **ideal-MG** indicates perfect scalability relative to **DMS-MG**.

As discussed in Section 6.3.2, almost all processes are assigned to the first mode of the tensor. In addition to decreasing the communication volume, this has the added effect of decreasing the amount of $\mathbf{A}$ that is stored and accessed on each node. As a result, the memory hierarchy is better utilized during the computational kernels. Interestingly, both DMS-MG and DMS-FG slow down between 512 and 1024 cores. This is a result of communication imbalance. While the *average* communication volume per node continues to decrease as we scale, we find that the *maximum* communication increases after 64 nodes (512 cores). DMS-CG is able to decrease both the average and maximum communication volume due to it having a larger amount of communication.

Table 6.4: Strong scaling results.

	Netflix				Delicious			
Nodes	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**
1	11.34	1.82	1.82	1.82	mem	7.90	7.90	7.90
2	6.07	1.16	0.84	1.03	mem	4.82	4.11	6.98
4	3.24	0.64	0.37	0.56	mem	3.08	2.23	4.43
8	1.90	0.39	0.18	0.31	28.01	1.88	1.25	2.16
16	1.34	0.23	0.09	0.22	25.54	1.26	1.04	1.35
32	0.95	0.20	0.06	0.20	24.93	0.86	0.59	0.96
64	0.82	0.19	0.04	0.19	25.15	0.81	0.37	0.66
128	1.33	0.14	0.05	0.24	24.34	0.42	0.32	0.48

	NELL				Amazon			
Nodes	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**
1	mem	10.82	10.82	10.82	mem	mem	mem	part
2	mem	6.66	6.01	9.14	mem	mem	mem	part
4	mem	4.06	3.32	5.24	mem	mem	mem	part
8	mem	2.55	2.02	3.46	mem	mem	8.34	part
16	mem	1.64	1.16	2.33	64.12	13.07	4.30	part
32	mem	1.09	0.82	1.74	50.92	10.06	2.19	part
64	mem	0.76	0.55	1.16	45.29	10.82	1.80	part
128	mem	0.53	0.35	0.92	40.20	7.82	0.97	part

	Random1				Random2			
Nodes	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**	**DFacTo**	**DMS-CG**	**DMS-MG**	**DMS-FG**
8	mem	mem	18.25	part	mem	mem	16.27	part
16	mem	12.80	11.42	part	mem	mem	9.61	part
32	mem	9.98	8.12	part	mem	10.61	6.25	part
64	mem	8.02	5.51	part	mem	7.86	4.06	part
128	mem	6.85	3.96	part	mem	5.53	2.81	part

Table values are seconds per iteration of CPD-ALS, averaged over 50 iterations. **mem** indicates the configuration required more memory than available. **part** indicates that we were unable to compute a data partitioning. Each node has eight cores which are fully utilized.

Chapter 7

Sparse Tensor Factorization with Constraints

We now present techniques for parallelizing and accelerating AO-ADMM (Algorithm 3). We begin with a parallelization strategy for ADMM (Algorithm 2) and then discuss a reformulation which improves convergence and computational efficiency. Next, we introduce a strategy for accelerating MTTKRP by exploiting the sparsity that naturally occurs in the factor matrices.

7.1 Accelerating ADMM

7.1.1 Parallelized ADMM

There is a wealth of research that focuses on accelerating the individual dense matrix kernels that constitute ADMM. These include matrix multiplication, Cholesky factorization, and forward/backward substitution [65]. Since the matrices of interest are tall and skinny, the kernels will ultimately be parallelized over the matrix rows while carefully optimizing for cache and other hardware features. Additionally, many popular constraints have proximity operators which are row separable. These include l_1 regularization for sparsity, non-negativity, and row simplex constraints.

A consequence of row separable computations is that Lines 6 through 9 in Algorithm 2, which comprise the bulk of the ADMM computation, can all be parallelized by

distributing rows of the tall and skinny matrices to threads. Lastly, convergence can be computed in parallel using any decomposition of the primal and dual variables. Each thread computes thread-local primal and dual norms (Lines 10 and 11) which are then aggregated.

7.1.2 Blocked ADMM

While the previous parallelization strategy offers a large amount of parallelism by focusing on the individual kernels, it does not take into account the iterative nature of the ADMM as a whole. We consider two challenges that emerge when viewing the ADMM algorithm beyond just a sequence of optimized kernels:

1. **Non-uniform convergence.** Real-world datasets often exhibit non-zeros which follow a power-law distribution. For example, a product rating tensor used by a recommender system will have some popular items and prolific users, while on average each item and user only have a few submitted ratings. It is natural to expect the rows corresponding to prolific users and items to carry much of the factorization's information. A consequence is that these "high-signal" rows may require many more iterations to converge than the average row. Since convergence criteria is an aggregation of all rows, this disparity not only decreases factorization quality by performing too few iterations on the high-signal rows, but also increases factorization time by performing additional iterations on the low-signal rows.

2. **Memory-bound computations.** Each step in Algorithm 2 involves a linear pass over the primal and dual matrices. Note that even the computationally intensive step, the forward/backward substitutions requiring $\mathcal{O}(F^2 I)$ operations, is linear in the large row dimension. If the size of the matrices exceed the size of the CPU cache, then we will access the matrices entirely from main memory instead of cache. Thus, the performance of Algorithm 2 will be determined by a machine's memory bandwidth instead of its compute capabilities.

We address both limitations by developing a blockwise reformulation of Equation 2.6.

If the proximity operator is row separable, we split the problem into B blocks of rows:

$$\begin{aligned} \underset{\{(\mathbf{H}_1,\widetilde{\mathbf{H}}_1),\ldots,(\mathbf{H}_B,\widetilde{\mathbf{H}}_B)\}}{\text{minimize}} \quad & \sum_{b=1}^{B} \frac{1}{2} \left\| (\mathbf{X}_{(1)})_b - \widetilde{\mathbf{H}}_b^T (\mathbf{C} \odot \mathbf{B})^T \right\|_F^2 + r(\mathbf{H}_b) \\ \text{subject to} \quad & \mathbf{H}_1 = \widetilde{\mathbf{H}}_1, \ldots, \mathbf{H}_B = \widetilde{\mathbf{H}}_B. \end{aligned}$$

Each of the blocks can then be optimized independently using Algorithm 2.

The blockwise reformulation accelerates convergence by allowing each block of rows to converge using different numbers of iterations. At one extreme, if $B = I$ then we separately optimize the solution for each row. In effect, the amount of work performed on each row is independent of all others. Therefore, the rows which take many iterations to converge will not be affected by those which converge early. Similarly, computation will not be wasted on rows which have already converged.

Likewise, the blockwise reformulation improves computational performance by creating temporal locality in the matrices. Since a block is processed until it has converged, a sufficiently small block size will allow of the necessary matrix data to be cache resident throughout the optimization procedure. Hence, we can expect to achieve better performance than the previously memory-bound formulation.

From a parallelization standpoint, this blockwise reformulation naturally provides an alternative decomposition of the computations. Instead of parallelizing the individual steps within Algorithm 2, we can simply distribute blocks to threads. This has the benefit of eliminating all synchronization overheads, as each block can be optimized totally in parallel. Even though blocks are equal in size, they may require different numbers of iterations. Thus, we cannot statically distribute blocks and instead dynamically load balance the optimization at block-level granularity. The work distribution is a simple loop over the B blocks and can be managed by the dynamic looping mechanism provided by many parallel frameworks such as OpenMP.

Selecting the number of blocks affects both convergence rate and execution performance. A natural first choice is to use $B{=}I$ and optimize over each row individually. In effect, the convergence benefits and temporal locality are maximized. Unfortunately, other overheads such as function calls and instruction cache misses are exaggerated when such a small amount of work is performed in each step. We empirically found that blocks of 50 rows offered a good trade-off between convergence and execution.

While we focus on shared-memory parallelism in this work, we note that the block-wise formulation also affords opportunities for distributed-memory parallelism. Since each block is processed independently, no communication needs to occur beyond the MTTKRP operation, which has efficient distributed-memory algorithms such as the fine-grained decomposition [27] and medium-grained decomposition (Chapter 6).

7.2 MTTKRP with Sparse Factors

Sparsity in the CPD factors is an attractive characteristic to practitioners. Intuitively, a sparse solution is a more simple one, and is thus easier to interpret and to gain insight from. Factor sparsity is also attractive from a performance perspective, as it affords opportunities for computational savings by avoiding operations with zero elements.

We focus our sparsity optimizations on the MTTKRP operation. Each tensor non-zero results in an access to the matrix factors, and thus MTTKRP can benefit greatly from sparsity. MTTKRP is primarily bound by memory bandwidth due to accesses to the factor matrices, and thus optimizations should reduce the volume of data fetched from the factors in order to achieve speedups. This is especially challenging when computing with a small rank, as each row has relatively few elements regardless, and thus memory savings are limited to a fraction of a cache line.

A first solution is to store a copy of the sparse factors in CSR format. CSR allows the factors to be randomly accessed by rows, which is required during MTTKRP. Since only the non-zero values and their indices are represented, the amount of data fetched from main memory scales with the matrix sparsity. Fortunately, the switch from a dense to compressed matrix format only requires minor changes to the MTTKRP algorithm. Whole rows of the factor are accessed at a time, and thus we only need to account for the difference between a dense and sparse row representation. In practice, the cost of accessing $\mathbf{C}$ dominates accesses to the other factor matrices due to it being accessed by every non-zero instead of fiber or slice. Therefore, we only represent $\mathbf{C}$ in CSR form and need only minor modifications to the MTTKRP implementation.

While a CSR representation *decreases* the effects of limited memory bandwidth, it *increases* the effects of memory latency. Consider the difference in implementation of dense and CSR matrices. A dense matrix will incur one latency cost when initially

accessing a row of $\mathbf{C}$, but the remaining entries exhibit spatial locality and will be fetched from main memory within the same cache line. Adjacent cache lines should be fetched by the hardware prefetching mechanisms. A CSR matrix, however, is implemented with three structures which encode the row length, the non-zero indices, and the non-zero values. The row length is required to index into the indices and values, and thus multiple latency costs will be incurred.

We address the challenge of memory latency costs by considering a hybrid combination of the dense and sparse matrix structures. Much like the distribution of tensor non-zeros, the sparsity patterns of the matrix factors are non-uniform. Importantly, $\mathbf{C}$ may have a few mostly-dense columns, with the remaining ones containing only a few non-zeros. We call a column "dense" when it contains more non-zeros than the average column density. When constructing the hybrid structure, we first sort the columns based on the number of non-zeros and place the dense columns first. The dense columns are represented with a simple dense matrix, and the remaining sparse columns are stored in CSR format. When a row of $\mathbf{C}$ is accessed during MTTKRP, we use software prefetching to begin fetching the CSR structure. During the data movement, we compute with the dense entries of the row. Finally, the row of the CSR structure is processed after the dense component.

Unlike the tensor which has a static sparsity pattern throughout the factorization, the sparsity patterns of the factors are dynamic. Therefore, techniques to exploit sparsity must be carefully vetted for efficiency, because their overheads are not amortized over multiple iterations. The potential gains that can be achieved by using a CSR representation to accelerate MTTKRP need to be balanced with the cost of constructing this CSR representation. Constructing the CSR matrix is an $\mathcal{O}(IF)$ operation due to it requiring a pass over the dense matrix to determine the sparsity pattern. Fortunately, this overhead is negligible when multiple ADMM iterations are performed, each taking $\mathcal{O}(F^2 I)$ time.

Table 7.1: Summary of datasets.

Dataset	NNZ	I	J	K
Reddit [57]	95M	310K	6K	510K
NELL [45]	143M	3M	2M	25M
Amazon [49]	1.7B	5M	18M	2M
Patents [58]	3.5B	46	240K	240K

NNZ is the number of nonzero entries in the dataset. **I**, **J**, and **K** are the dimensions of the datasets. *K*, *M*, and *B* stand for thousand, million, and billion, respectively.

7.3 Experimental Methodology

7.3.1 Experimental Setup

Experiments were performed on a workstation with 396GB of main memory and two ten-core Intel Xeon E5-2650v3 processors with 25MB of last-level cache. Our source code is modified from SPLATT version 1.1.1, a C library for high performance sparse tensor factorization [5]. Our source code is to be made part of the next SPLATT release. We use the Intel compiler version 17.0.1 and Intel MKL used for Cholesky factorization and forward/backward substitution. OpenMP is used for parallelism. Unless otherwise specified, we run with twenty OpenMP threads.

7.3.2 Datasets

Table 7.1 summarizes the tensors used in our evaluation. We selected four real-world tensors from various domains which are publicly available as part of the FROSTT collection [58]. NELL is a *noun-verb-noun* tensor from the Never Ending Language Learning project [45]. Reddit is a *user-community-word* tensor encoding a subset of comments on Reddit from 2007 to 2010 [57]. Amazon is a *user-item-word* tensor of product reviews [49]. Patents is a *year-word-word* tensor of pairwise co-occurrence probabilities from United States utilities patents.

7.3.3 Convergence Criteria

We follow the factorization community and use a normalized value of LS($\cdot$) to measure the quality of a factorization. Specifically, we measure the relative error between $\boldsymbol{\mathcal{X}}$ and its factored form:

$$\text{relative error} = \frac{\left\| \boldsymbol{\mathcal{X}} - \sum_{f=1}^{F} \mathbf{A}(:,f) \circ \mathbf{B}(:,f) \circ \mathbf{C}(:,f) \right\|_F^2}{\|\boldsymbol{\mathcal{X}}\|_F^2}.$$

Convergence is detected when the relative error improves less than 10^{-6} or if we exceed 200 outer iterations.

7.4 Results

7.4.1 Relative Factorization Costs

We first investigate the performance characteristics of a parallel implementation of AO-ADMM without blocking or sparsity optimizations. Figure 7.1 shows the fraction of factorization time spent in the main computational kernels of Algorithm 3 (i.e., MTTKRP and ADMM) during a rank-50 non-negative factorization.

Neither of the kernels consistently dominate the computation. NELL has both the longest modes of the datasets and is also the most sparse, and therefore spends most of the runtime in ADMM updating the factors. Amazon and Patents, on the other hand, are dominated by MTTKRP. These tensors have more non-zeros and are both more dense than NELL, which emphasizes the cost of MTTKRP. These results indicate that in order to achieve high performance, both the computations performed during ADMM and MTTKRP need to be optimized and parallelized effectively.

7.4.2 Parallel Scalability

We examine the parallel scalability of the baseline AO-ADMM algorithm in Figure 7.2. The baselines achieve speedups ranging from 5.4× on NELL to 12.7× on Patents. Note that the amount of achieved speedup is related to the amount of time spent on MTTKRP in Figure 7.1. The datasets which are dominated by the cost of MTTKRP exhibit the best scalability due to the already-optimized kernels provided by SPLATT. Using

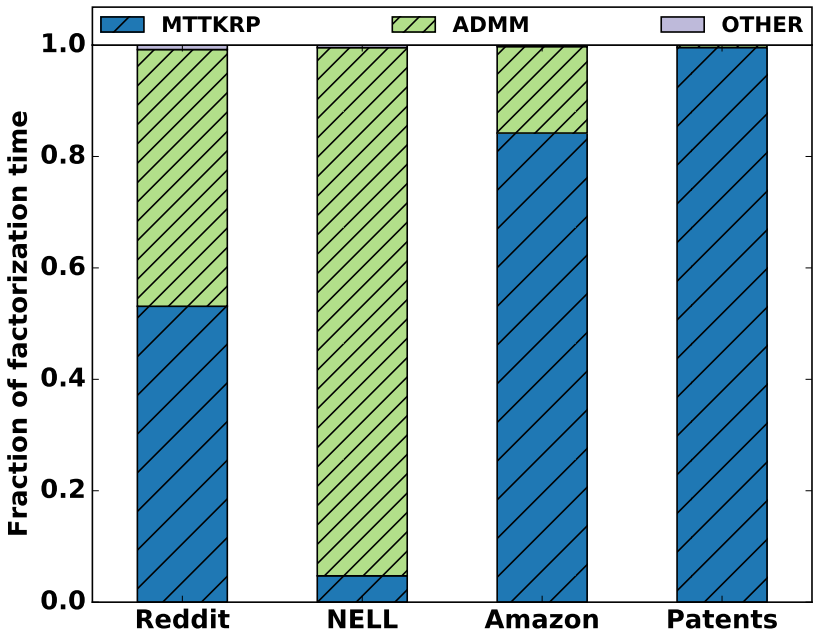
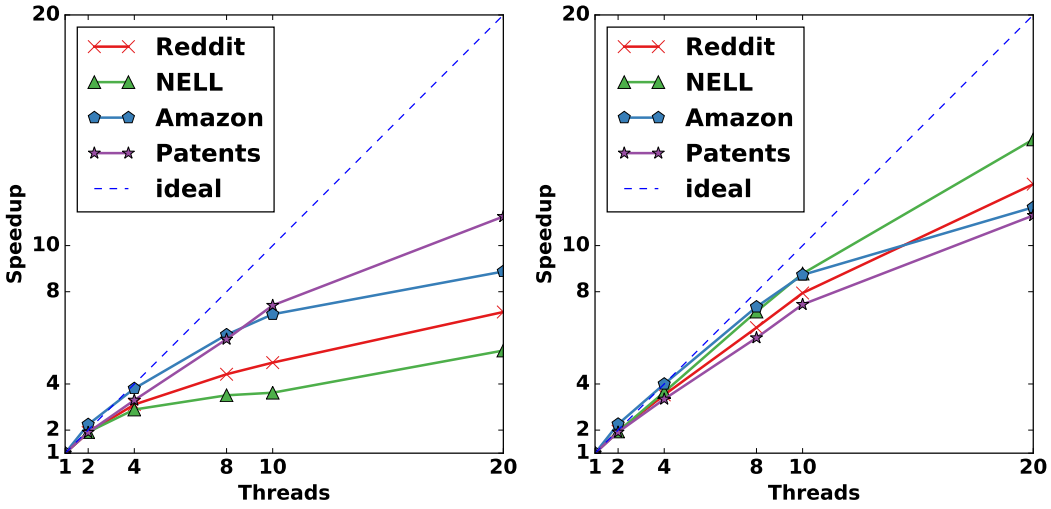


Figure 7.1: Fraction of time spent in MTTKRP and ADMM during rank-50 non-negative factorization.



(a) Baseline ADMM

(b) Blocked ADMM

Figure 7.2: Parallel speedup on rank-50 non-negative CPD.

the blocked formulation, speedups range from 12.7× on Patents to 14.6× on NELL. The trend observed on the baseline scalability is reversed: datasets which are dominated by ADMM runtime now achieve the best scalability. This is expected, as blocked ADMM features high temporal locality and minimal synchronization costs compared to its baseline counterpart.

7.4.3 Convergence Rate

We now evaluate the benefits of the blockwise formulation on the convergence of AO-ADMM. It is important to separate the speedups achieved by accelerated convergence and by faster execution rate. Figure 7.3 and Figure 7.4 show convergence on all datasets as both a function of time and the number of outer iterations. Including convergence as a function of outer iteration allows us to observe the effects of blocked ADMM without considering machine effects such as cache locality. When a solution of higher quality (i.e., lower error) is reached, or fewer outer iterations are performed, then we know that convergence has been improved.

Blocking improves the per-iteration convergence on every evaluated dataset. The positive benefits of blocking are observed in two forms: (i) reaching a higher-quality

Table 7.2: Effects of sparse matrix data structures on CPD runtime.

	Reddit			Amazon		
	$F = 50$ 3% dense	$F = 100$ 1% dense	$F = 200$ 2% dense	$F = 50$ 3% dense	$F = 100$ 3% dense	$F = 200$ 5% dense
DENSE	227.8	430.7	1774.9	305.9	715.3	18120.0
CSR	212.7	231.6	1000.5	<u>272.6</u>	<u>539.2</u>	<u>12535.6</u>
CSR-H	<u>199.4</u>	<u>186.3</u>	<u>903.5</u>	320.6	588.5	13476.5

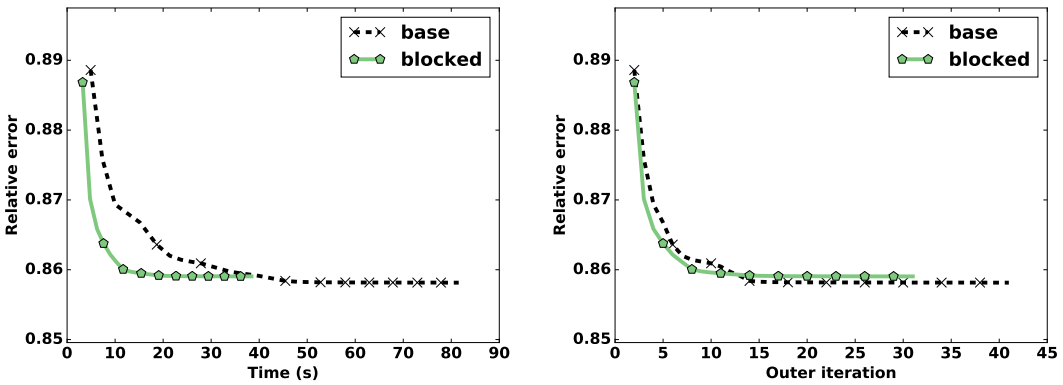
Values are the total time in seconds to compute the CPD. We impose a $10^{-1}||\cdot||_1$ regularization on all factors to promote sparsity. The density of each rank indicates the density of the longest factor matrix (i.e., the matrix that is stored in a sparse representation during MTTKRP). Density is computed via $\text{nnz}(\mathbf{C})/KF$. DENSE uses a baseline MTTKRP implementation with a dense matrix. CSR uses the compressed sparse row (CSR) format during MTTKRP. CSR-H uses the hybrid dense and CSR format.

solution in the same or less time, or (ii) converging to a comparable solution in fewer iterations. For example, NELL and Amazon both converge to lower errors than the baselines. This is most exemplified with NELL, which converges 3.7× faster and reaches a 3% lower error. The success of NELL is a result of the combination of both faster ADMM iterations and additional ADMM iterations being performed on the "high-signal" blocks. Reddit and Patents, on the other hand, converge in fewer iterations due to blocking and reach errors that are less than 1% higher than the baseline.

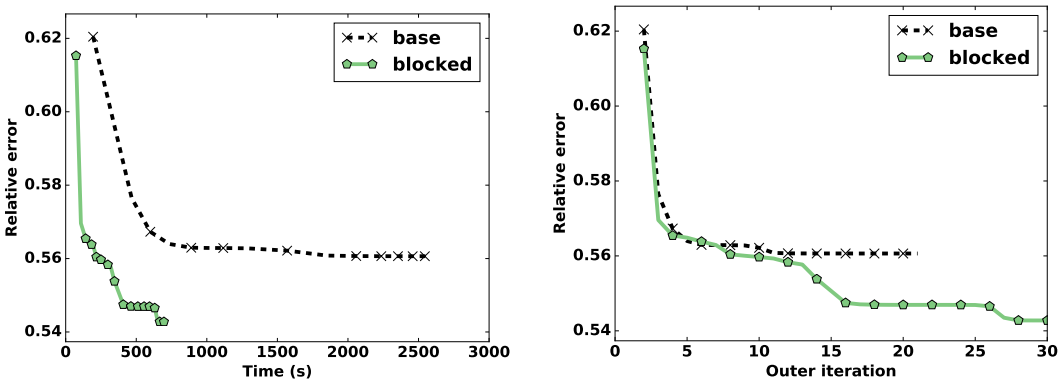
7.4.4 Accelerating MTTKRP with Factor Sparsity

We now evaluate the benefits of exploiting factor sparsity during MTTKRP. We compute l_1-regularized factorizations in order find to sparse solutions. For each factor, we set $r(\cdot) = 10^{-1}||\cdot||_1$. We omit NELL and Patents from this evaluation because they did not tend to exhibit sparsity and instead converged to either mostly dense or totally zero solutions as the regularization parameter was introduced.

Table 7.2 shows the time-to-solution for Reddit and Amazon on a variety of ranks. For each configuration, we include times for the baseline dense computation, CSR, and the hybrid dense and CSR computation. Notably, the complete factorization time is presented despite the evaluated algorithms only benefiting the MTTKRP portion of the



(a) Reddit

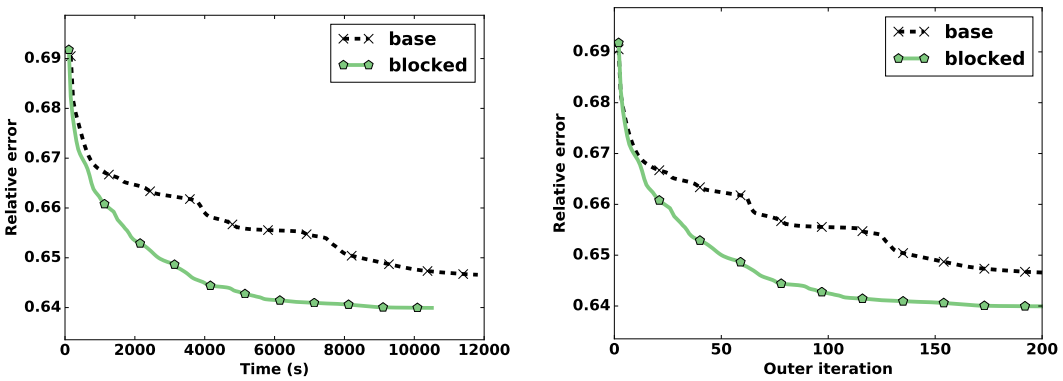


(b) NELL

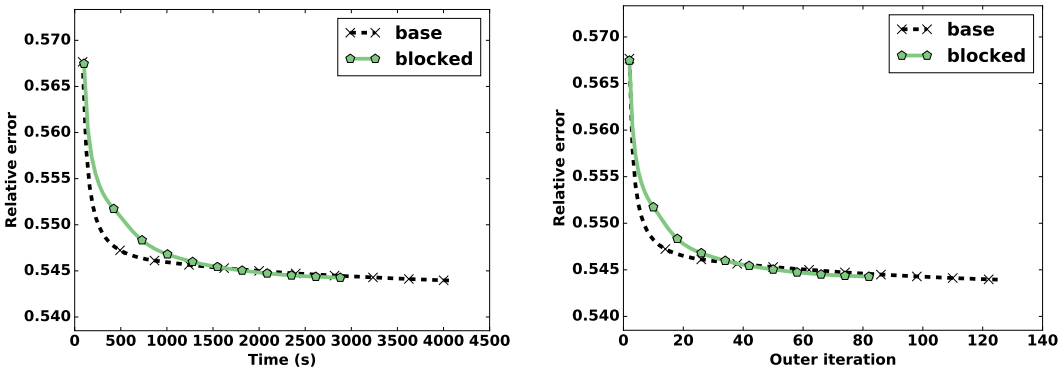
Figure 7.3: Effects of blockwise ADMM on rank-50 non-negative factorization on Reddit and NELL. **base** and **blocked** are the unblocked and blocked algorithms, respectively.

factorization. The time-to-solution more accurately portrays the benefits of sparsity as it accounts for conversion overheads and the early iterations in which the factors are not yet sparse. For our evaluation, we empirically determined that a factor can be gainfully treated as sparse when its density falls below 20%.

Exploiting sparsity outperforms the baseline dense computation in all cases. Speedups from sparse MTTKRP range from 1.1× to 2.3×. Interestingly, the hybrid CSR data structure is beneficial for Reddit but not Amazon. While the two datasets feature similar levels of sparsity in their largest factor matrix, the longest mode of Amazon is over thirty times longer than Reddit.



(a) Amazon



(b) Patents

Figure 7.4: Effects of blockwise ADMM on rank-50 non-negative factorization on Amazon and Patents. **base** and **blocked** are the unblocked and blocked algorithms, respectively.

Chapter 8

Large-Scale Tensor Completion

In this chapter, we develop scalable, high-performance algorithms for the three most popular algorithms for tensor completion: alternating least squares (ALS), coordinate descent (CCD++), and stochastic gradient descent (SGD). We detail optimizations for both shared-memory and distributed-memory systems and demonstrate scalability for each of the algorithms on a variety of datasets. Lastly, we introduce a source of randomization to ALS and CCD++ and experimentally demonstrate an improvement in solution time and quality.

8.1 Algorithms for High Performance Tensor Completion

8.1.1 Efficient Loss Computation with a Compressed Sparse Tensor

The choice of data structure for representing a sparse tensor affects performance in ways such as memory bandwidth, number of FLOPs performed, and opportunities for parallelism. The HPC formulations of the various optimization algorithms that we developed use the *compressed sparse fiber* (CSF) data structure (Chapter 4), illustrated in Figure 4.4. CSF can be thought of as a generalization of the compressed sparse row data data structure for matrices. The CSF data structure represents a tensor as a forest of I trees, with each mode recursively compressed one after the other and stored in the next level of the tree.

The evaluation of $\mathcal{L}(\cdot)$ in Equation (2.3) benefits from exploiting the CSF tensor

representation. Consider the computation of $\mathcal{L}(\cdot)$ associated with two successive non-zeros:

$$\mathcal{L}(i,j,k) = \boldsymbol{\mathcal{X}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f),$$

$$\mathcal{L}(i,j,k') = \boldsymbol{\mathcal{X}}(i,j,k') - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k',f).$$

We reuse partial results by storing the Hadamard (element-wise) product of $\mathbf{A}(i,:)$ and $\mathbf{B}(j,:)$ in a row vector $\mathbf{v}$:

$$\mathbf{v} \leftarrow \mathbf{A}(i,:) * \mathbf{B}(j,:)$$

$$\mathcal{L}(i,j,k) = \boldsymbol{\mathcal{X}}(i,j,k) - \sum_{f=1}^{F} \mathbf{v}(f)\mathbf{C}(k,f),$$

$$\mathcal{L}(i,j,k') = \boldsymbol{\mathcal{X}}(i,j,k') - \sum_{f=1}^{F} \mathbf{v}(f)\mathbf{C}(k',f).$$

This reduces the computation from $2F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ multiplications to $F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})+FP$, where P is the number of unique $\boldsymbol{\mathcal{X}}(i,j,:)$ fibers. The *matricized tensor times Khatri-Rao product* (MTTKRP) operation present in ALS uses the same technique for operation reduction (Chapter 4).

8.1.2 Parallel ALS

We follow the strategy of parallelizing over the rows of $\mathbf{A}$ for shared-memory parallel systems. Recall from Equation (2.8) that $\mathbf{H}_i$ has $|\boldsymbol{\mathcal{X}}(i,:,:)|$ rows and F columns. A major challenge when designing ALS algorithms is that if multiple rows of $\mathbf{A}$ are computed at once, the various $\mathbf{H}_i$ matrices must somehow be represented in memory. However, the collective storage for all $\mathbf{H}_i$ prohibitively requires $F\,\mathrm{nnz}(\boldsymbol{\mathcal{X}})$ storage. Another option, and the one used by existing work [13], is to aggregate rank-1 updates. For each non-zero $\boldsymbol{\mathcal{X}}(i,j,k)$, a rank-1 update with the row vector $(\mathbf{B}(j,:) * \mathbf{C}(k,:))$ is accumulated into $\mathbf{H}_i^T\mathbf{H}_i$. A naive implementation must aggregate the rank-1 updates for all possible i, requiring IF^2 storage. It was observed that this storage overhead can be reduced by sorting the tensor non-zeros before processing long modes [13].

We instead store N CSF representations of $\boldsymbol{\mathcal{X}}$, in which the nth CSF places the nth mode at the top level of the tree and sorts the remaining ones by length. Since the

mode ordering forces all non-zeros in $\boldsymbol{\mathcal{X}}(i,:,:)$ to be grouped into the same tree, we can process all of the non-zeros in $\boldsymbol{\mathcal{X}}(i,:,:)$ sequentially and thus only the $\mathcal{O}(F^2)$ memory associated with a single row is allocated. For multicore systems, we parallelize over the rows of $\mathbf{A}$ with a memory overhead of only $\mathcal{O}(F^2)$ per thread. The total storage overhead with P threads is $(N-1)\,\text{nnz}(\boldsymbol{\mathcal{X}}) + PF^2 = \mathcal{O}(N\,\text{nnz}(\boldsymbol{\mathcal{X}}))$, which is smaller than $F\,\text{nnz}(\boldsymbol{\mathcal{X}})$ and IF^2 for most problems.

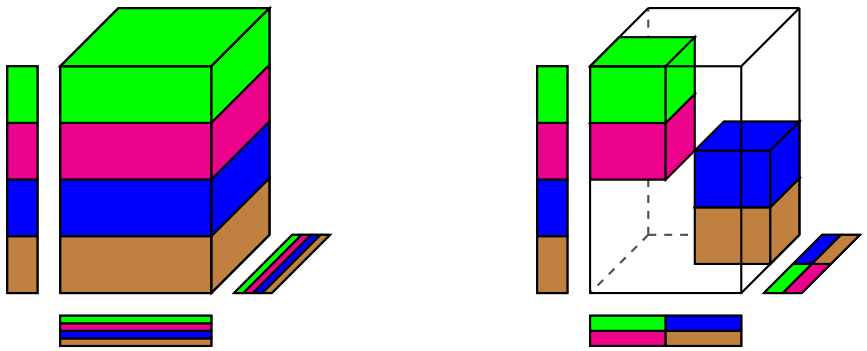
The construction of $\mathbf{H}_i^T\mathbf{H}_i$ and $\mathbf{H}_i^T\,\text{vec}(\boldsymbol{\mathcal{X}}(i,:,:))$ are performed together during a single pass over the sparsity structure of $\boldsymbol{\mathcal{X}}$ using similar strategies as presented in Section 4.3. Interestingly, the expression $\mathbf{H}_i^T\,\text{vec}(\boldsymbol{\mathcal{X}}(i,:,:))$ is equivalent to computing one transposed row of the MTTKRP operation, for which operation-efficient CSF algorithms exist.

Accumulating rank-1 updates into $\mathbf{H}_i^T\mathbf{H}_i$ causes $\mathcal{O}(F^2)$ data to be accessed for each non-zero and leads to memory-bound computation. We collect the intermediate Hadamard products formed during MTTKRP into a thread-local matrix of fixed size. When that matrix fills, or all non-zeros in $\boldsymbol{\mathcal{X}}(i,:,:)$ are processed, the thread performs one rank-k update, where k is the number of rows in the thread-local matrix. We empirically found that a matrix of size $2048\times F$ is sufficient to see significant benefits from the BLAS-3 performance. For typical values of F, this equates to storage overheads of up to a few megabytes per thread.

We follow existing work and use a coarse-grained decomposition in the distributed-memory setting [14]. Coarse-grained decompositions impose separate 1D decompositions on each tensor mode, eliminating the need to communicate and aggregate partial computations. As a result, the worst-case communication volume is reduced from $\mathcal{O}(IF^2)$ to $\mathcal{O}(IF)$ per process, which is the cost of exchanging updated rows of $\mathbf{A}$. Our coarse-grained decomposition uses chains-on-chains partitioning to optimally load-balance the slices of each mode [63]. After partitioning and distributing non-zeros, we use our shared-memory ALS kernels without modification. After a factor matrix is updated, we use an all-to-all collective communication to exchange the new rows.

8.1.3 Parallel SGD

Processing non-zeros in a totally random fashion requires $\mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{X}}))$ work per epoch to shuffle the non-zeros and results in random access patterns to the matrix factors. We



(a) (b)

Figure 8.1: Asynchronous SGD strategies for P=4 processes. (a) S=1 stratum layers. The longest mode is partitioned and other modes are updated asynchronously. (b) S=2 stratum layers. The longest mode is partitioned and S teams of P/S asynchronously update at the end of each stratum.

instead use a more coarse-grained approach and randomize only one mode of the tensor, i.e., we randomize the processing order of the trees in the CSF structure (Figure 4.4) but sequentially access non-zeros within each tree. This approach reduces the cost of shuffling to $\mathcal{O}(I)$ and retains the cache-friendly access pattern to the matrix factors that is provided by the sorted indices in CSF. Our shared-memory SGD algorithm uses a Hogwild approach and processes the trees in parallel without synchronization constructs [36]. Choosing which mode to randomize requires careful consideration, because SGD may not converge if updates are not sufficiently stochastic. This is demonstrated in Section 8.4.1.

In a distributed-memory setting, we refer to a group of blocks in the P^N stratification grid (Figure 3.2) that share a coordinate as a *stratum layer*. For example, blocks of coordinate $(i, :, :)$ are in the ith stratum layer along the first mode. We partition the grid by first selecting the longest mode and decomposing it in a 1D fashion. Each process is assigned to a unique stratum layer in the longest mode. This privatizes the largest matrix factor, meaning no processes can produce updates which overlap with the another process' local portion. Thus, there will not be any communication associated with the largest mode during the factorization.

The number of strata increases exponentially with the number of modes. Assuming the number of non-zeros is constant, and therefore the work per epoch is constant, the average work per stratum that can be processed in parallel decreases exponentially. Many real-world tensors have modes with skewed dimensions. For example, a tensor of health records will have significantly more unique patients than unique medical procedures or doctors. The lengths of the dense modes can be much smaller than the number of available cores. In this case, the amount of parallelism is limited by the shortest mode (the number of blocks in a stratum equals $\min(I_1, ..., I_N)$, where I_n is the length of nth mode).

In order to address these parallelism challenges, we extend ASGD to tensors, which allows multiple processes to update the same row of a factor matrix with their local copies. ASGD can trade-off the staleness of factor matrices for increased parallelism by adjusting the number of copies and the frequency of synchronizing the copies. Our implementation is parameterized with the number of stratum layers, S, which determines the number of strata as S^{N-1}. We can set $S = P$ (which reduces ASGD to the usual stratified SGD) for tensors with a few modes or set $S < P$ for tensors with more modes. When $S < P$, since each stratum has only S independent blocks, P/S processes need to update the same range of factor matrices simultaneously, resulting in up to P/S copies of a factor matrix row. Specifically, we partition an N-mode tensor into a $P \times S^{N-1}$ grid and assign P mode-1 layers to each process. Then, we group every P/S processes as a *team* with total S teams. This process is shown in Figure 8.1.

We partition each factor matrix among P processes, aligning with the grid used for the tensor partitioning. At the beginning of a stratum, each process sends the rows of the factor matrices that it owns to the other processors that need them. By our construction, a factor matrix row will be sent to one team, thus limiting the number of copies to P/S. Then, each process goes through the non-zeros of the current stratum it owns, updating the corresponding rows of factor matrices. After the update, processes send the updated rows back to their owners. Finally, processes compute weighted sums of the received updated rows, where the weights are the number of non-zeros which updated the particular row. For example, for a 3-mode tensor $\boldsymbol{\mathcal{X}}$ and a given stratum, suppose process p_1 processes 2 non-zeros in a mode-2 slice $\boldsymbol{\mathcal{X}}(:, i, :)$ and p_2 processes 1 non-zeros in the same slice. At the end of the stratum, the owner of ith row computes

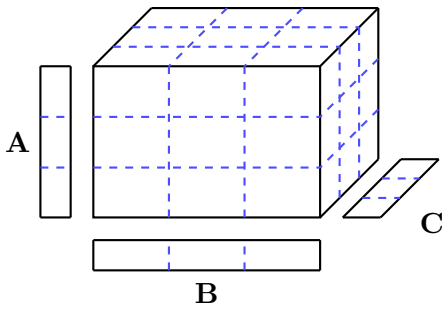


Figure 8.2: The P^N-way tiling scheme used in CCD++ for P=3 threads.

a weighted sum of the local copies of ith row of mode-2 factor matrix from p_1 and p_2, using 2/3 and 1/3 as their weights, respectively. Since we synchronize the factor matrices every stratum, the number of synchronizations per epoch is S^{N-1}.

ASGD allows us to alleviate the limited amount of parallelism and frequent communication, the primary challenges of SGD, especially for high-mode tensors. Still, compared to ALS and CCD++, SGD has higher communication volume, which can be analyzed as follows. For each stratum, a process receives the rows of factor matrices that correspond to the non-zeros it needs to process, which equals the sum of number of non-empty slices of each mode except for the first mode (which is completely privatized). In a worst case (and not an uncommon case for highly sparse tensors), we have only a few non-zeros per slice, leading to receiving $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}_{p,s}))$ floating-point numbers for each mode for the sub-tensor $\boldsymbol{\mathcal{X}}_{p,s}$ processed by process p at stratum s. Summing over all processes and strata results in $\mathcal{O}(NF \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ total communication volume. It is important to receive only the required factor matrix rows corresponding to non-empty slices to be processed. Otherwise, the total communication volume will be $\mathcal{O}\left(S^{N-2}F\sum_{n=2}^{N} I_n\right)$ as analyzed by Shin and Kang [14].

8.1.4 Parallel CCD++

As discussed in Section 3.2.3, CCD++ implementations follow a similar parallelization strategy as ALS on shared-memory systems. Following Equation (2.9), all α_i's and β_i's are independent subproblems and can be computed in parallel. However, unlike ALS, it is not advantageous to use separate CSF representations for each mode in order to

extract coarse-grained parallelism. This is due to the added $(N-1)\,\text{nnz}(\boldsymbol{\mathcal{X}})$ operations that would be required for updating multiple residual tensors. Therefore, we restrict ourselves to a single tensor and turn to other decomposition strategies.

We leverage the P^N-way tiling strategy developed for parallelizing MTTKRP with a single CSF (Chapter 4). Shown in Figure 8.2, an N-dimensional grid is imposed on $\boldsymbol{\mathcal{X}}$, with each dimension having P chunks. This allows P threads to partition any mode of $\boldsymbol{\mathcal{X}}$ into P independent chunks, each consisting of P^{N-1} tiles. Each mode of $\boldsymbol{\mathcal{X}}$ can thus be updated without parallel overheads such as reductions or synchronization.

Conveniently, because threads access whole layers of tiles at a time, we do not need each of the P^N tiles to have a balanced number of non-zeros. Instead, only the layers themselves need to be balanced. We use chains-on-chains partitioning to determine the layer boundaries in each mode, resulting in load-balanced parallel execution.

In a distributed-memory setting, the communication requirements of CCD++ closely follow those of MTTKRP. A minor variation comes from CCD++ being a column-major method, and thus we must exchange partial results and updated columns F times per mode instead of exchanging full rows of size F once per mode. Unlike ALS, exchanging partial results does not cause a prohibitive amount of communication. We can therefore choose from the recently proposed fine-grained [27] and medium-grained (Chapter 6) decompositions for MTTKRP. We opt for the medium-grained decomposition, which imposes an N dimensional grid over $\boldsymbol{\mathcal{X}}$. The medium-grained decomposition varies from our tiling strategy in that there are only as many cells in the grid as processes. After distributing $\boldsymbol{\mathcal{X}}$ over a grid, our shared-memory parallelization strategy is applied by each process independently.

8.1.5 Dense Mode Replication

ALS and CCD++ parallelize over the dimensions of $\boldsymbol{\mathcal{X}}$. As discussed in Section 8.1.3, real-world tensors have skewed mode lengths. Simply parallelizing over the short, dense modes is insufficient because the number of threads can easily outnumber the slices. Additionally, the dense modes often have non-zeros that are not uniformly distributed, leading to further load imbalance.

The issue of dense modes was first addressed by Shao [12] for shared-memory ALS when $I < P$. Non-zeros are instead divided among threads and each thread computes a

local set of $\mathbf{H}_i^T\mathbf{H}_i$ and $\mathbf{H}_i^T \text{vec}(\boldsymbol{\mathcal{X}}(i,:,:))$. A parallel reduction is then used to combine all partial results before the Cholesky factorization. We adopt this solution in our own ALS implementation and parallelize directly over non-zeros when the mode is dense. We use the tiling mechanism employed by CCD++ to load balance non-zeros. The tensor is still stored using CSF, and thus the optimizations to achieve BLAS-3 performance can still be employed.

CCD++ uses a similar mechanism for handling dense modes. CCD++ only constructs one representation of $\boldsymbol{\mathcal{X}}$ which is already tiled for parallelism. There is no advantage to tiling the dense modes because they will not be partitioned, and so we use a P^{N-d}-way tiling on a tensor with d dense modes. Each thread then computes local α_i and β_i and we aggregate them with a parallel reduction when the mode is dense.

The same decomposition strategies apply to distributed-memory computation. The factors representing dense modes are replicated on all processes. Instead of doing a coarse- or medium-grained decomposition to establish communication patterns, a simple all-to-all reduction is used to aggregate partial computations.

8.2 Improving Convergence of ALS and CCD++ via Randomization

Non-convex optimization problems such as Section 2.3 are susceptible to converging to local minima. Algorithms such as SGD and randomized block coordinate descent [66, 67] have been used with great success because they have the ability to escape local minima when a deterministic descent algorithm would instead converge. While randomized algorithms typically provide no guarantees on converging to a globally optimal solution, in practice they often converge faster and arrive at higher quality local minima than their deterministic counterparts.

The success of these randomized methods motivates the addition of randomization to ALS and CCD++. Careful attention must be paid in order to avoid introducing additional computation or communication and to maintain the degree of available parallelism. In the following discussion we refer to the factor matrices as $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$ for notational convenience.

8.2.1 Randomized ALS

An epoch of ALS traditionally updates $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$ in a cyclic fashion. A cyclic scheme means that $\mathbf{A}^{(n)}$ will always be updated using the latest $\mathbf{A}^{(n-1)}$, which in turn was updated using the latest $\mathbf{A}^{(n-2)}$, and so on. Thus, after the factor matrices are initialized, the optimization process is entirely deterministic. Instead, we propose to introduce *mode-level* randomization and apply a random permutation to the tensor modes each epoch $\pi : \{1, \ldots, N\} \to \{1, \ldots, N\}$. This randomization scheme allows $\mathbf{A}^{(\pi(n))}$ to be updated using the latest state of a different mode each epoch. Pseudocode is presented in Algorithm 11.

Mode-level randomization can be implemented with negligible overhead. If each process uses the same reproducible pseudo-random number generator, then π can be constructed locally by each process and does not need to be communicated. Thus, the only cost of randomization is the construction of π, which is negligible.

8.2.2 Randomized CCD++

CCD++ presents two opportunities for low-overhead randomization: *rank-level* and *mode-level*. Rank-level randomization updates the F rank-one tensors in a random order instead of cyclically. Within each rank-one optimization, CCD++ can leverage mode-level randomization by altering the order of the updates to the N columns being updated. Pseudocode is presented in Algorithm 12. Like ALS, the only overheads associated with randomization are the construction of two permutations of size F and N, both of which are negligible.

Additional randomized behavior could conceivably be incorporated into ALS and CCD++ by making multiple passes over the factors and updating random subsets of the rows each time. However, we do not explore this option because it reduces the degree of available parallelism from the length of the mode to the number of rows in the current block. Moreover, the already-unstructured communication pattern becomes dynamic due to different sets of rows being updated and communicated each pass.

Algorithm 11 Mode-randomized ALS

while not converged **do**
  Form random permutation $\pi_N : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$.
  for $n \in \pi_N$ **do**
    for $i \in I_n$ in parallel **do**
      Update $\mathbf{A}^{(n)}(i, :)$ using Equation (2.8).
    end for
  end for
end while

Algorithm 12 Rank- and mode-randomized CCD++

while not converged **do**
  Form random permutation $\pi_F : \{1, \ldots, F\} \rightarrow \{1, \ldots, F\}$.
  for $f \in \pi_F$ **do**
    Form random permutation $\pi_N : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$.
    for $n \in \pi_N$ **do**
      for $i \in I_n$ in parallel **do**
        Update $\mathbf{A}^{(n)}(i, f)$ using Equation (2.9).
      end for
    end for
  end for
end while

8.3 Experimental Methodology

8.3.1 Experimental Setup

We use the Cori supercomputer at NERSC. Each compute node has 128 GB of memory and is equipped with two sockets of 16-core Intel Xeon E5-2698 v3 that has 40 MB last-level cache. The compute nodes are interconnected via Cray Aries with Dragonfly topology. Our ALS, SGD, and CCD++ implementations are made part of the open source tensor factorization library, SPLATT [5]. We use double-precision floating-point numbers and 64-bit integers. We use the Intel compiler version 16.0.0 with the

`-xCORE-AVX2` option for instructions available in the Haswell generation of Xeon processors, Cray MPI version 7.3.1, and Intel MKL version 11.3.0 for LAPACK routines used in ALS. Compute jobs are scheduled with Slurm version 16.05.03. We run one MPI rank per socket (two ranks per node) and one OpenMP thread per core for SGD and CCD++, and one MPI rank per node for ALS. We use the *bold driver* heuristic [68] for a dynamic step size parameter in SGD with an initial value of 10^{-3}.

We follow the recommender systems community and use *root-mean-square error* (RMSE) as a measure of factorization quality. RMSE was the metric used by the Netflix Prize [44], where the first algorithm to improve the baseline RMSE by 10% was awarded one million dollars. RMSE is defined as

$$\text{RMSE} = \sqrt{\frac{\sum_{\boldsymbol{\mathcal{X}}(:,:,:)} \mathcal{L}(i,j,k)^2}{\text{nnz}(\boldsymbol{\mathcal{X}})}}.$$

Datasets are split into 80% *training*, 10% *validation*, and 10% *test* sets. The training set is used to compute the factorization. RMSE is computed each epoch using the validation set, and convergence is detected when the RMSE does not improve for twenty epochs. The final factorization quality is determined by the test set. All algorithms are given the same random initialization for fairness.

8.3.2 Datasets

Table 8.1 summarizes the tensors used for evaluation. The reported non-zeros and memory requirements reflect only that of the training data, because that is the portion of the computation that we focus on in this work. We work with real-world datasets coming from a variety of domains. Movielens, Netflix and Yahoo! are *(user, item, month)* product rating tuples. Values in Movielens and Netflix range from 1-5 and Yahoo! values range from 1-100. Amazon is formed from *(user, item, word)* tuples taken from product reviews. Patents is formed from *(year, word, word)* pairwise co-occurrences taken from United States utility patents from years 1969 through 2014. Non-zero $\boldsymbol{\mathcal{X}}(i,j,k)$ is equal to $\log(f_{jk})$, where f_{jk} is the number of times terms j and k appeared in a seven-word window during year i. Outpatient is a six-mode tensor of *(patient, institution, physician, diagnoses, procedure, day)* tuples formed from synthetic outpatient Medicare claims. We selected non-zeros from the original data in order to

Table 8.1: Summary of training datasets for tensor completion.

Dataset	NNZ	Dimensions	Mem. (GB)
Movielens [69]	20M	138K, 27K, 234	0.5
Netflix [44]	80M	480K, 17K, 73	2.4
Outpatient [55]	87M	1.6M, 6K, 13K, 6K, 1K, 192K	4.5
Yahoo! [56]	210M	1M, 625K, 133	6.3
Amazon [49]	1.4B	4.8M, 1.7M, 1.8M	41.5
Patents [58]	2.9B	46, 240K, 240K	85.7

K, **M**, and **B** stand for thousand, million, and billion, respectively. **NNZ** is the number of non-zeros in the training dataset. **Mem.** is the memory required to store the tensor as a list of *(coordinate, value)* tuples, measured in gigabytes.

have three-, four-, five-, and six-mode versions of this dataset with the same number of non-zeros. The Amazon, Patents, and Outpatient datasets are publicly available in the FROSTT tensor collection [58].

Throughout the remaining discussion, we will use appropriate subsets of our datasets based on the subject of the experiment. For example, when discussing convergence properties of the various optimization algorithms, we will focus on the ratings tensors (i.e., Movielens, Netflix, and Yahoo!) due to them requiring many iterations to converge and their prevalence in the matrix and tensor completion community. As another example, the synthetic Outpatient tensor converges too quickly for a convergence study, but has a large number of modes and allows for an evaluation on the scalability for higher order tensors.

8.4 Results

8.4.1 Intra-Method Evaluation

ALS

Figure 8.3 shows the effects of BLAS-3 routines and dense mode replication while factoring the Yahoo! tensor with F=10. With neither optimization, each ALS epoch takes

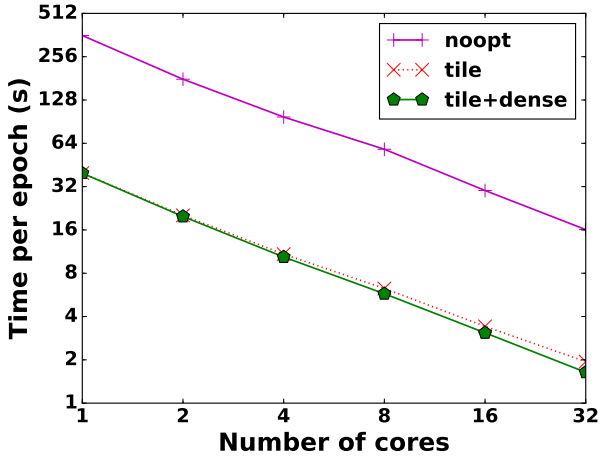


Figure 8.3: Effects of ALS optimizations during a rank-10 factorization of the Yahoo! tensor. **noopt** is a baseline ALS implementation with a CSF data structure. **tile** delays rank-1 updates to use the BLAS-3 *dsyrk* routine. **tile+dense** includes **tile** and also dense mode replication.

359 seconds on average and we achieve a 22.3× speedup on 32 cores. BLAS-3 routines improve the runtime by 9×, but speedup is reduced to 20.5×. Finally, by replicating the dense mode across cores and using a parallel reduction on partial products, we achieve 24.2× speedup with 32 cores and a resulting 219× improvement over the original serial implementation.

CCD++

Figure 8.4 shows the effects of dense mode replication on CCD++. Dense mode replication will not affect serial runtime, and so we show speedup as we scale the number of threads. Speedup is improved from 4.6× to 16.2× due to improved load balance from mode-replication and also from temporal locality. Note that super-linear speedup is observed, reaching 2.3× with two cores. The $p \times p$ tiling of $\boldsymbol{\mathcal{X}}$ for p threads results in a smaller portion of the factor matrices accessed at a time, improving temporal locality. Interestingly, super-linear speedup was also observed in the original CCD++ evaluation for matrices [15].

Despite the improved speedup, CCD++ sees little improvement after 16 cores (one

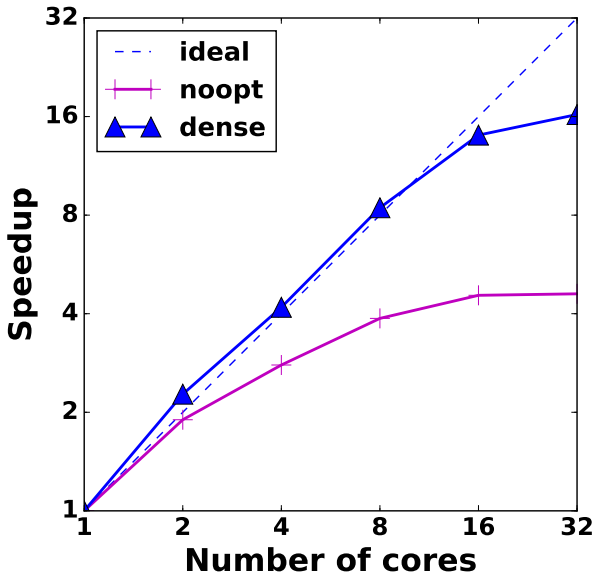


Figure 8.4: Effects of CCD++ optimizations during a rank-10 factorization of the Yahoo! tensor. **noopt** is a baseline CCD++ implementation with the CSF data structure. **dense** uses dense mode replication.

full socket) due to NUMA effects and the memory-bound nature of the algorithm. Compared to ALS, CCD++ performs a factor of F fewer FLOPs on every non-zero that is accessed. Additionally, CCD++ being a column-oriented method requires NF passes per epoch over the sparsity structure of $\boldsymbol{\mathcal{X}}$, compared to N times for ALS and one time for SGD. We show in Section 8.4.6 that this limitation can be solved by simply using one MPI process per socket.

SGD

Figure 8.5 shows the effects of coarse-grained randomization on SGD convergence. We use a full random traversal of the tensor as a baseline and compare against two CSF configurations. The first configuration, CSF-S, sorts the modes in non-decreasing order with the smallest mode placed at the top of the data structure. CSF-S is the default mode ordering used by SPLATT due to it typically resulting in the highest level of compression. CSF-L sorts the modes in non-increasing order, with the longest mode

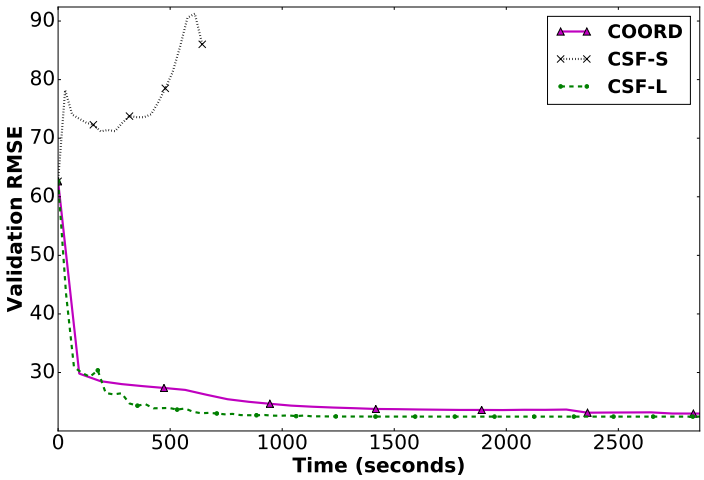


Figure 8.5: Effects of randomization strategy on SGD convergence during a serial rank-40 factorization of the Yahoo! tensor. **COORD** uses complete randomization on a coordinate form tensor. **CSF-S** randomizes over the shortest mode. **CSF-L** randomizes over the longest mode.

placed at the top of the CSF structure. CSF-L effectively trades additional storage for increased randomization and higher degrees of parallelism. CSF-S has the fastest per-epoch runtime but fails to converge. CSF-L exhibits similar per-epoch convergence compared to the baseline, but the computational savings afforded by the CSF data structure results in a faster time-to-solution. We therefore use CSF-L as the default in the remaining experiments.

Figure 8.6 shows the effects of stratification on convergence. We evaluate three algorithms across two factorizations: fully asynchronous, fully stratified, and a hybrid algorithm with 16 stratum layers, totaling 256 strata. The hybrid configuration outperforms both baselines for the rank-10 factorization and converges to a higher quality solution in less time. All three algorithms are competitive for rank-40, but the hybrid ultimately reaches a lower RMSE.

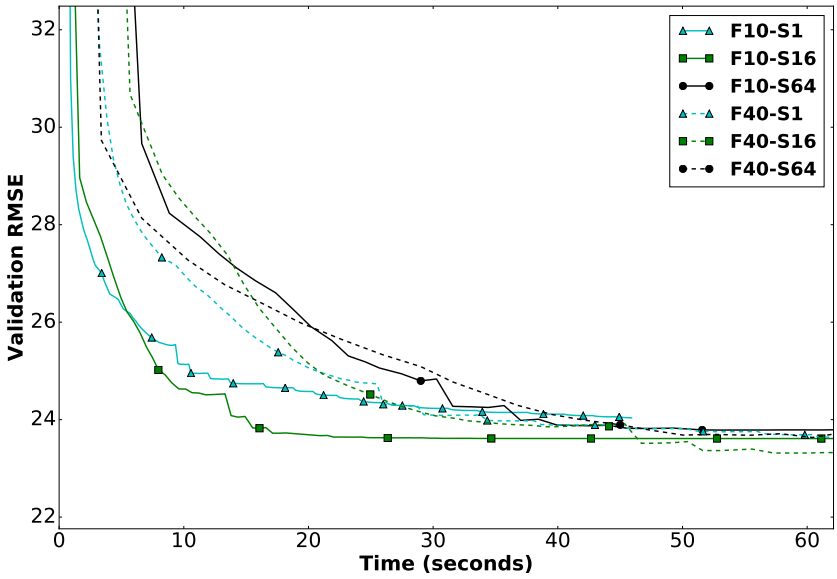


Figure 8.6: Convergence rates for SGD parallelization strategies using 32 compute nodes on the Yahoo! dataset. **F** denotes the rank of the factorization. **S** denotes the number of stratum layers, scaling from fully asynchronous (S1) to fully stratified (S64).

8.4.2 Communication Volume

Figure 8.7 shows the average communication volume per epoch. CCD++ consistently has a lower communication volume than SGD and ALS due to its medium-grained decomposition. The communication volume for SGD increases until four nodes (eight ranks) are used, and then sharply decreases. The communication volume of SGD scales with the number of strata used, which we limit to 64. Recall that SGD uses S^{N-1} strata for an N-mode tensor. Therefore, we see volume increase until we reach the maximum number of strata. From that point communication is limited to within strata, and we see communication volume decrease. The communication volume of ALS increases until eight nodes and then stays near constant. ALS uses a coarse-grained decomposition which in the worst case requires each communication of entire factors per epoch.

8.4.3 Strong Scaling

Figure 8.8 shows strong scaling results. For SGD, we again limit the number of strata to 64 which provides a good trade-off between convergence rate and parallelism. As

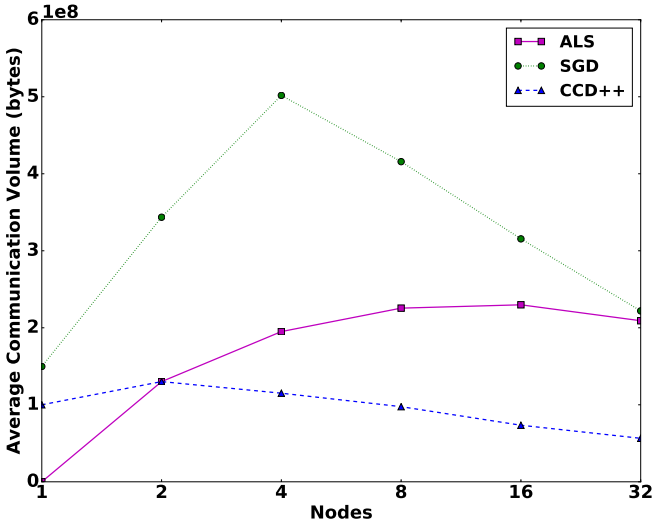
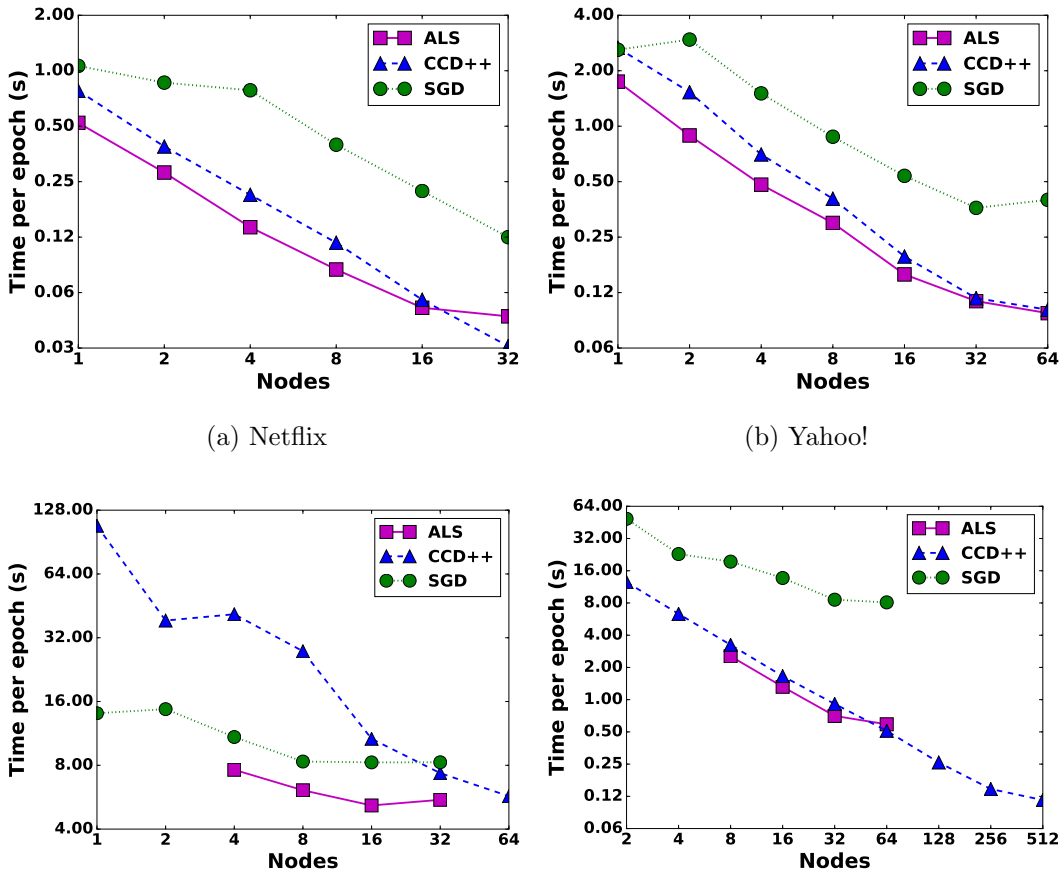


Figure 8.7: Average communication volume per node on the Yahoo! dataset. CCD++ and SGD use two MPI ranks per node and ALS uses one.

discussed in the evaluation of communication volume, SGD primarily introduces overhead until the maximum number of strata is reached, and after that point we begin to overcome the communication overheads. SGD only scales to eight nodes on the Amazon tensor. This is because Amazon is significantly more sparse than Netflix and Yahoo! and as a result SGD performs less work per stratum, resulting in high communication costs.

ALS is unable to process the Amazon and Patents tensors on as few nodes as SGD and CCD++ due to it requiring three copies of $\boldsymbol{\mathcal{X}}$ during factorization. Neither ALS nor CCD++ are consistently faster per epoch at 32 nodes. CCD++ begins slower on all datasets but out-scales ALS on all but Yahoo!, which does not have enough non-zeros to effectively parallelize over 64 nodes (2048 cores). The disparity between CCD++ and other algorithms on Amazon is due to its large size being more taxing on memory bandwidth. CCD++ is able to scale through 512 nodes (16384 cores) on Patents due to its high density, resulting in a small amount of communication and memory bandwidth relative to the computational load. ALS is unable to scale past 32



(a) Netflix

(b) Yahoo!

(c) Amazon

(d) Patents

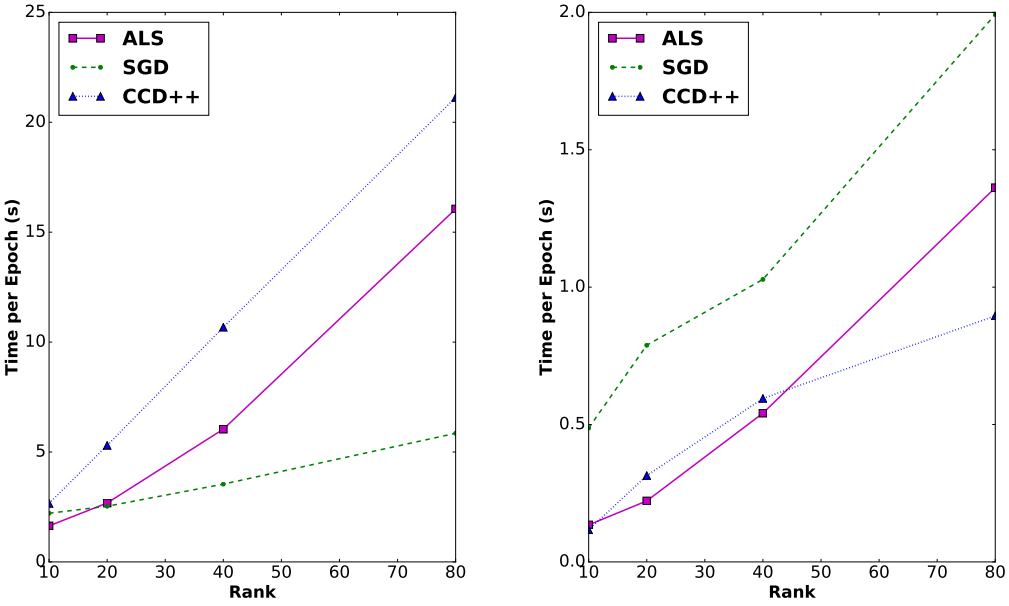
Figure 8.8: Strong scaling the optimized ALS, SGD, and CCD++ algorithms. Each node has 32 cores.

nodes on Patents due to its short mode lengths, which result in a load-imbalanced coarse-grained decomposition. The second and third modes are too long to treat as dense (i.e., the storage costs of normal equations are prohibitive). In contrast, the medium-grained decomposition used by CCD++ is able to find a load balanced distribution on Patents.

8.4.4 Rank Scaling

Figure 8.9 shows the effects varying the rank of the factorization. We scale from rank 10 to 80 on the Yahoo! dataset. CCD++ and SGD both have $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$ complexity, so we expect the runtime to increase by $8\times$ as we scale F. ALS, on the other hand, has complexity $\mathcal{O}(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{X}}) + IF^3)$. The $F^2 \operatorname{nnz}(\boldsymbol{\mathcal{X}})$ term will dominate in most scenarios because users are interested in low rank factorizations and because $I \ll \operatorname{nnz}(\boldsymbol{\mathcal{X}})$ for most tensors. Under this assumption, we expect the runtime of ALS to increase by a factor of $80^2/10^2 = 64$.

CCD++ sees the expected linear increase in runtime on both 32 and 1024 cores: $7.9\times$ and $7.6\times$, respectively. SGD scales sub-linearly and only sees $2.4\times$ and $4.2\times$ increases on 32 and 1024 cores, respectively. The sub-linear effects are due to the way SGD accesses the matrix factors. SGD only ever accesses entire rows of the factors, leading to spatial locality and vectorized inner loops. We do not see the same effects for CCD++ because it accesses the factors in a strided manner that is dependent on the sparsity pattern. Additionally, CCD++ must traverse the sparsity pattern of the tensor F times for each mode, compared to once for SGD. Surprisingly, ALS only sees $9.8\times$ and $10.1\times$ increase in runtime at 32 and 1024 cores, respectively. While the work does increase quadratically, all of the quadratic functions are performed by BLAS-3 routines on small, dense matrices. The work that depends on the sparsity pattern of $\boldsymbol{\mathcal{X}}$ is an MTTKRP operation, which has the same spatial locality as SGD and a cheaper complexity of $\mathcal{O}(F \operatorname{nnz}(\boldsymbol{\mathcal{X}}))$. The BLAS-3 routines will eventually out-scale the cost of the MTTKRP operation, but factorizations of such a high rank are unlikely to be useful to a domain expert.



(a) 32 cores (1 node)

(b) 1024 cores (32 nodes)

Figure 8.9: Effects of increasing factorization rank on the Yahoo! dataset.

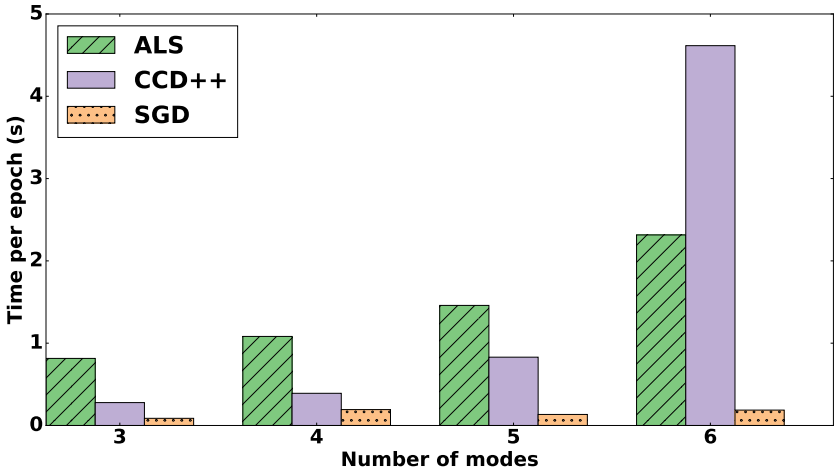


Figure 8.10: Average time per epoch with 16 nodes while scaling the number of modes using the Outpatient dataset.

8.4.5 Mode Scaling

Figure 8.10 shows the scalability of our algorithms on the Outpatient dataset as we increase the number of tensor modes while keeping the number of non-zeros constant. ALS sees a roughly linear increase in runtime, matching the computational complexity in Table 2.1. The runtime of the sixth mode increases super-linearly, which we attribute to the sixth mode being longer than most others and ALS having a $\mathcal{O}(IF^3)$ complexity component. CCD++ exhibits severe slowdown as the number of modes is increased. This is due to CCD++ doing NF passes over $\boldsymbol{\mathcal{X}}$ per epoch and performing only $\mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{X}}))$ work per pass. The memory-bound nature of CCD++ is exaggerated as the number of modes increases. SGD has a nearly constant runtime due to it only performing one pass over $\boldsymbol{\mathcal{X}}$ per epoch, regardless of the number of modes. Additionally, higher-order tensors such as Outpatient have several dense modes which will exhibit high temporal locality, leaving the system's memory bandwidth free for streaming through the single representation of $\boldsymbol{\mathcal{X}}$. SGD appears to be an attractive choice for higher-order tensors. We cautiously recommend it, however, because situations may arise in which many stratum layers are required to maintain convergence, negatively impacting performance.

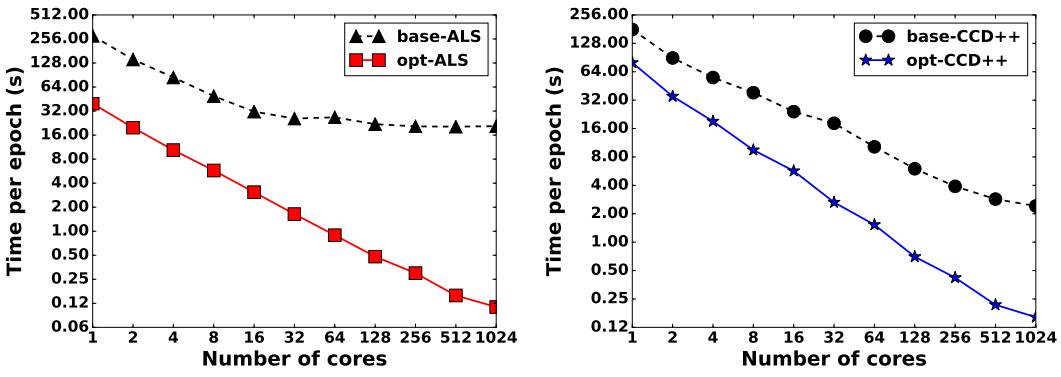
8.4.6 Comparison Against the State-of-the-Art

In Figure 8.11 we compare our ALS and CCD++ algorithms against the state-of-the-art MPI implementations [13]. We scale from 1 to 1024 cores on the Yahoo! tensor with F=10. We use one MPI rank per node for opt-ALS due to the high speedup it achieves on 32 cores and also due to the high communication volume that comes with a coarse-grained decomposition. Throughout the comparison we refer to the existing implementations as "base-ALS" and "base-CCD++" and our own as "opt-ALS" and "opt-CCD++".

On one core, opt-ALS is 7× faster than base-ALS due to the BLAS-3 performance. opt-ALS then scales to achieve 353× speedup at 1024 cores, compared to the 13.5× speedup of base-ALS. The improvements in speedup are due to the coarse-grained decomposition used by opt-ALS which reduces the communication volume from $\mathcal{O}(IF^2)$ to $\mathcal{O}(IF)$ words. The difference in communication requirements is observed in the ratio of communication to computation: base-ALS spends 95% of the total runtime communicating, compared to opt-ALS which spends 39% of its runtime communicating. opt-ALS is 185× faster than base-ALS when both use 1024 cores.

Serial opt-CCD++ is 2.2× faster than base-CCD++ due to the operation reduction and improved cache locality resulting from the CSF data structure. The locality improvements are also present in the ALS results, but due to ALS being a compute-bound algorithm they are not observed except for very small values of F. On 1024 cores, opt-CCD++ and base-CCD++ achieve 685× and 74.2× speedup, respectively. The disparity in speedups is attributed to the large amount of communication performed by base-CCD++, in which 69% of the total runtime is spent in communication routines. In comparison, opt-CCD++ spends 25% of the total runtime communicating. The medium-grained decomposition used by opt-CCD++ results in a smaller communication volume than the arbitrary decomposition used by base-CCD++. Communication volume is further reduced by utilizing *sparse* communication and only sends an updated value to the processes that require it.

We also note that with the configuration using one MPI rank per socket, opt-CCD++ achieves a 30× speedup on 32 cores compared to 16.2× with a pure OpenMP configuration.



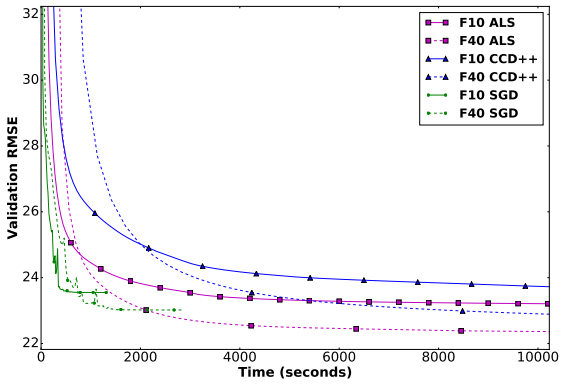
(a) ALS (b) CCD++

Figure 8.11: Comparison of the presented ALS and CCD++ algorithms (prefixed **opt**) against the state-of-the-art MPI implementations (prefixed **base**) on a rank-10 factorization of the Yahoo! tensor.

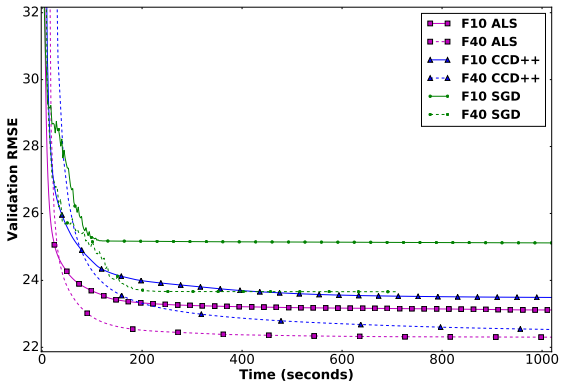
8.4.7 Convergence

We now evaluate the time-to-solution of ALS, CCD++, and SGD. Figure 8.12 shows convergence of our optimized algorithms using 1, 32, and 1024 cores. SGD is the most successful algorithm in a serial setting. For F=10, SGD converges within 1500 seconds and achieves a quality that takes ALS over twice as long to reach. SGD sees a similar advantage when F=40, but ALS ultimately reaches a higher quality solution shortly after SGD converges.

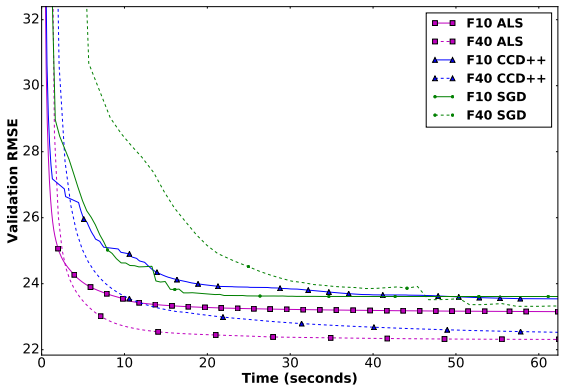
On a single node, ALS outperforms SGD and CCD++ for both F=10 and F=40. Since ALS has the fastest per-epoch times in addition to making the most progress per epoch, it is the recommended algorithm for small-to-moderate node counts. CCD++ is also competitive in the multi-core environment and has the added benefit of having a smaller memory footprint due to only storing a single copy of $\mathcal{X}$. Trends continue as we move to a large-scale distributed system and ALS narrowly bests CCD++ for F=40. While ALS still converges faster than SGD and CCD++, CCD++ is more scalable in distribute-memory environments due to its lower communication volume.



(a) Serial



(b) 32 cores (1 node)



(c) 1024 cores (32 nodes)

Figure 8.12: Convergence rates for parallel methods on the Yahoo! dataset with factorization ranks 10 (F10) and 40 (F40). Ticks are placed every five epochs.

8.4.8 Effects of Randomization on ALS and CCD++

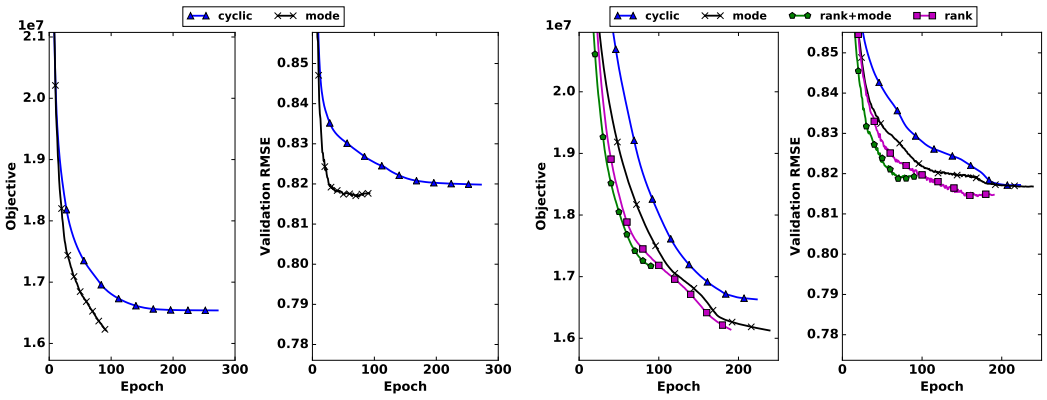
We now explore the convergence benefits offered by randomization during ALS and CCD++. Figure 8.13 shows the convergence per epoch when applying mode-level randomization to ALS and combinations of mode- and rank-level randomization to CCD++. In each case, we show both the value of the objective function and the validation RMSE. Observing the objective function offers insights into the effects on the actual optimization algorithm, whereas observing the validation RMSE measures the predictive abilities of the resulting model (i.e., its usefulness to a domain specialist).

Both algorithms benefit from randomization on the Movielens dataset. Randomized ALS requires 66% fewer epochs to arrive at a validation RMSE which is 0.3% lower than cyclic. Likewise, all variations of randomization improve the validation RMSE of CCD++. The best performance is achieved by rank-level randomization, which converges 16% faster than cyclic and improves the solution again by 0.3%. The objective function is improved in all cases, and therefore the gains are attributed to an optimization algorithm which converges more quickly to a better local minimum.

Netflix likewise benefits from randomization in terms of both convergence rate and validation RMSE. Interestingly, mode-level randomization achieves the best validation RMSE for both ALS and CCD++, but at the same time arrives at the worst objective value. Randomization allows the optimization algorithm to learn a more general, albeit less optimized solution.

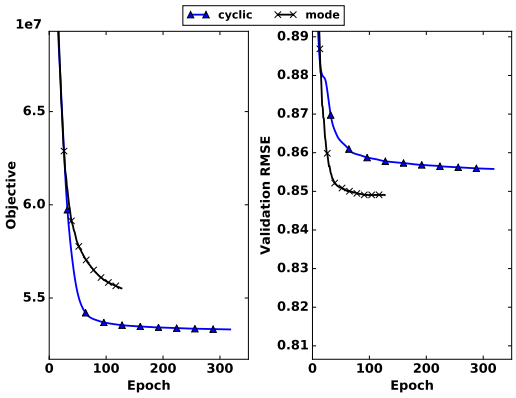
We find that algorithms which use mode-level randomization tend to learn models with empty columns, reducing the rank of the factorization. We explore this phenomenon in Figure 8.14. We plot the number of non-zero rank-one components, which refer to as the *effective rank* of the factorization. ALS and CCD++ learn models with ranks 32 and 25, respectively. The lower-rank models do not fit the training data as well, and thus have a higher objective value. However, due to their lower complexity and sufficiently good model of the data, they are able to better predict unknown entries in the validation set.

Lastly, Yahoo! does not benefit from randomization and the algorithms which use mode-level randomization achieve notably worse objective values and validation RMSEs. The lower-rank factorizations learned via randomization are not able to capture the training data sufficiently well to result in a predictive model.

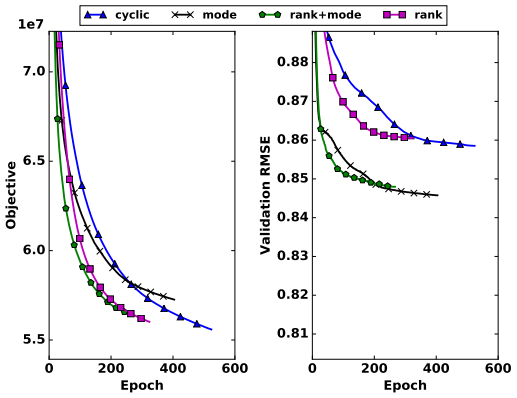


(a) Movielens - ALS

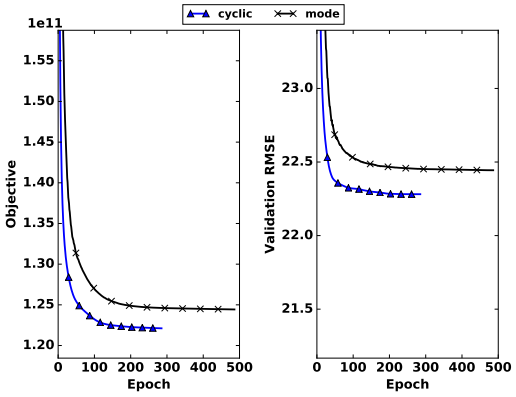
(b) Movielens - CCD++



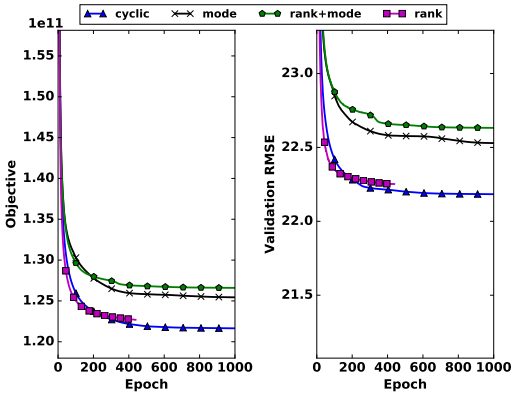
(c) Netflix - ALS



(d) Netflix - CCD++

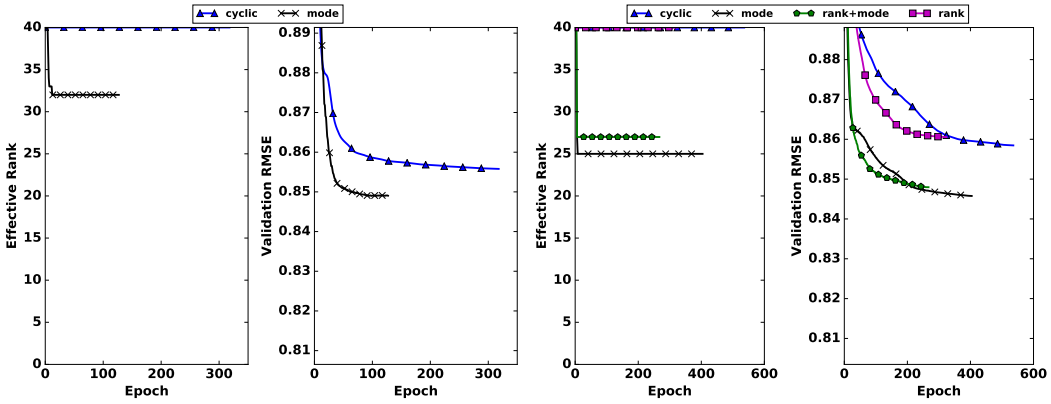


(e) Yahoo! - ALS



(f) Yahoo! - CCD++

Figure 8.13: Effects of randomization on ALS and CCD++ convergence during rank-40 factorizations of three datasets. **cyclic** uses no randomization, **mode** uses mode-level, **rank** uses rank-level, and **rank+mode** uses both rank- and mode-level randomization.



(a) Netflix - ALS

(b) Netflix - CCD++

Figure 8.14: The number of non-zero rank-one components (**effective rank**) during a rank-40 factorization of the Netflix dataset.

Chapter 9

Accelerating the Tucker Decomposition

The TTMc operation introduced in Section 2.4 is the key computation to consider when computing the Tucker decomposition. Existing strategies for performing TTMc either rely on memoizing intermediate results to save computation [39, 40] or operating in a memory-efficient manner at the expense of additional FLOPs [38]. The memory overhead of memoization is closely tied to the dimensionality and the sparsity pattern of the tensor, and can result in significant memory overhead. Meanwhile, the memory-efficient strategies require orders of magnitude more computation and are often impractical for large and sparse tensors.

This chapter details our operation- and memory-efficient parallel algorithm for TTMc. We first perform a reformulation of the underlying computations in order to remove redundancies and then describe a parallel algorithm which uses CSF to exploit these redundancies. We then analyze the computational complexity of our algorithm.

9.1 TTMc with a Compressed Sparse Tensor

9.1.1 Operation-Efficient Formulation

We work from Equation (3.3) which processes individual non-zeros. Each non-zero contributes

$$\mathbf{Y}_{(1)}(i_1,:) \leftarrow \mathbf{Y}_{(1)}(i_1,:) + \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\left[\mathbf{A}^{(2)}(i_2,:) \otimes \cdots \otimes \mathbf{A}^{(N)}(i_N,:)\right].$$

There are two forms of arithmetic redundancies that we eliminate during TTMc:

Distributive Kronecker Products. Consider two adjacent non-zeros in a three-mode tensor. Performing a TTMc operation for the first mode results in the following computations:

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\left[\mathbf{A}^{(2)}(j,:) \otimes \mathbf{A}^{(3)}(k_1,:)\right], \tag{9.1a}$$

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\left[\mathbf{A}^{(2)}(j,:) \otimes \mathbf{A}^{(3)}(k_2,:)\right]. \tag{9.1b}$$

The Kronecker product (KP) is a distributive operation, and so we combine Equation (9.1a) and Equation (9.1b) to eliminate a KP and reach a more efficient update:

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \mathbf{Y}_{(1)}(i,:) + \mathbf{A}^{(2)}(j,:) \otimes \left[\boldsymbol{\mathcal{X}}(i,j,k_1)\mathbf{A}^{(3)}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\mathbf{A}^{(3)}(k_2,:)\right].$$

This can be exploited for any set of non-zeros that reside in the same fiber. For each fiber, we accumulate all of the linear combinations of rows of $\mathbf{A}^{(3)}$ into a row vector, followed by a single KP:

$$\mathbf{Y}_{(1)}(i,:) \leftarrow \sum_{\boldsymbol{\mathcal{X}}(i,:,:)} \mathbf{A}^{(2)}(j,:) \otimes \left(\sum_{\boldsymbol{\mathcal{X}}(i,j,:)} \boldsymbol{\mathcal{X}}(i,j,k)\mathbf{A}^{(3)}(k,:)\right).$$

This eliminates the construction and accumulation of $\text{nnz}(\boldsymbol{\mathcal{X}}(i,j,:))-1$ KPs, resulting in a reduction of $2F_2F_3\,(\text{nnz}(\boldsymbol{\mathcal{X}}(i,j,:))-1)$ FLOPs. This strategy generalizes to any number of modes:

$$\mathbf{Y}_{(1)} \leftarrow \sum_{\boldsymbol{\mathcal{X}}(i_1,:,\ldots,:)} \mathbf{A}^{(2)}(i_2,:) \otimes \left(\sum_{\boldsymbol{\mathcal{X}}(i_1,i_2,:,\ldots,:)} \mathbf{A}^{(3)}(i_3,:) \otimes \ldots \left(\sum_{\boldsymbol{\mathcal{X}}(i_1,\ldots,i_{N-1},:)} \boldsymbol{\mathcal{X}}(i_1,\ldots,i_N)\mathbf{A}^{(N)}(i_N,:)\right)\right).$$

Redundant Kronecker Products. Consider the case of performing mode-3 TTMc:

$$\mathbf{Y}_{(3)}(k_1,:) \leftarrow \mathbf{Y}_{(3)}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\left[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)\right],$$
$$\mathbf{Y}_{(3)}(k_2,:) \leftarrow \mathbf{Y}_{(3)}(k_2,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\left[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)\right].$$

Note that $[\mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:)]$ appears in the processing of both non-zeros. We eliminate operations by reusing the KP for both non-zeros:

$$\mathbf{s} \leftarrow \mathbf{A}^{(1)}(i,:) \otimes \mathbf{A}^{(2)}(j,:),$$
$$\mathbf{Y}_{(3)}(k_1,:) \leftarrow \mathbf{Y}_{(3)}(k_1,:) + \boldsymbol{\mathcal{X}}(i,j,k_1)\mathbf{s},$$
$$\mathbf{Y}_{(3)}(k_2,:) \leftarrow \mathbf{Y}_{(3)}(k_2,:) + \boldsymbol{\mathcal{X}}(i,j,k_2)\mathbf{s}.$$

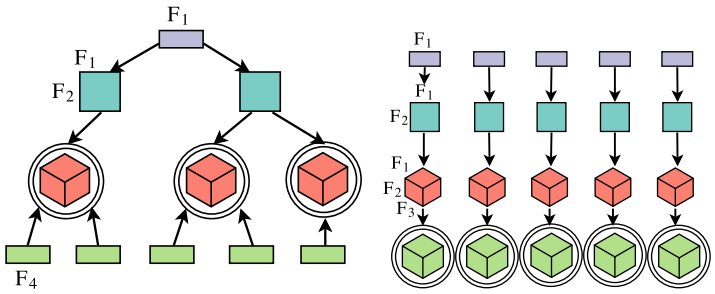
Reusing the shared KP for an entire fiber saves $F_1F_2(\text{nnz}(\boldsymbol{\mathcal{X}}(i,j,:))-1)$ FLOPs. As before, this process can be generalized to any number of tensor modes.

Operation-Efficient Algorithm.

Using the two previous optimizations, we can devise an algorithm which uses the CSF data structure to eliminate redundant operations. A branch in the tree structure at the ith level represents a set of non-zeros which overlap in the previous $i-1$ indices, which is precisely the scenario that the previous optimizations target. Our TTMc algorithm is described in Algorithm 13 and illustrated in Figure 9.1. Intuitively, partial computations begin at the root and leaf levels of the tree and grow inward towards the level representing the mode of computation. Algorithm 13 avoids intermediate memory blowup by processing the tree depth-first, which limits the intermediate memory to a single row of $\mathbf{Y}_{(n)}$.

Parallelism.

Algorithm 13 is parallelized by distributing the I_1 trees to threads. Each thread performs a depth-first traversal, and thus the thread-local storage overhead is asymptotically limited to a single row of $\mathbf{Y}_{(n)}$. A consequence of this distribution is the potential for write conflicts when updating any modes other than the first. This can be observed in Figure 4.4, in which node IDs are only unique within the root-level nodes. The same synchronization challenges are present when performing MTTKRP (see Chapter 4). We



(a) CSF (mode-1) (b) CSF (mode-3) (c) Coordinate (mode-4)

Figure 9.1: TTMc with CSF and coordinate data structures. The number of FLOPs performed on a node is equal to its volume. Circled nodes produce updates to the output.

present synchronization using mutexes for simplicity, but note that the algorithm can benefit from other mechanisms such as tiling or transactional memory (see Section 5.2).

9.1.2 Complexity Analysis

We now analyze the computational complexity of Algorithm 13. Let nodes(i) be the number of nodes present in the ith level of a CSF structure (by convention, the 1st level is the root level). The number of FLOPs required to perform TTMc for the nth mode is $\sum_{i=1}^{N} \text{nodes}(i) \times \text{cost}(i, n)$, where "cost" is defined as

$$\text{cost}(i,n) = \begin{cases} \prod_{j=1}^{i-1} F_j & \text{if } i < n, \\ 2\prod_{j=i}^{N} F_j & \text{if } i > n, \\ 2\prod_{\substack{j=1 \\ j\neq i}}^{N} F_j & \text{if } i = n. \end{cases} \tag{9.2}$$

Intuitively, the cost of a node above level-n is the cost of constructing a KP, and the cost at or below level-n is the cost of constructing *and* accumulating a KP.

When computing for the leaf mode of the tensor, Algorithm 13 assembles KPs and pushes them down the tree from root to leaves. The complexity grows with each level of the tree, with the final level having the same asymptotic complexity as Equation (3.4),

the baseline coordinate approach. At the other extreme, when $n = 1$, the computation moves upwards from leaves to root. Interestingly, the dimensionality of the KPs is non-decreasing, and at the same time the number of nodes in each level is non-increasing. In the worst case, non-zeros have no overlapping indices and the algorithm is equivalent to operating with a tensor stored in coordinate format. However, lower complexities are possible under some assumptions on the CSF structure and the ranks of the factorization. To see, compare the costs of levels i and $i-1$:

$$\frac{\text{nodes}(i) \times 2\prod_{j=i}^{N} F_j}{\text{nodes}(i-1) \times 2\prod_{j=i-1}^{N} F_j} = \frac{\text{nodes}(i)}{\text{nodes}(i-1)F_{i-1}}.$$

Suppose that the cost of the ith mode always exceeds mode $i-1$:

$$\text{nodes}(i) > \text{nodes}(i-1)F_{i-1}, \qquad i = 2, \ldots, N$$

then the Nth mode dominates the computation, arriving at a reduced complexity of $\mathcal{O}(\text{nodes}(N)F_N) = \mathcal{O}(\text{nnz}(\boldsymbol{\mathcal{X}})F_N)$.

9.2 Utilizing Additional CSF Representations

Section 9.1.2 showed that Algorithm 13 has the potential for an asymptotic speedup over the competing memory-efficient approaches. This depends on the costs of the lower levels of the tree dominating those at the top, which is possible if: (i) the branching factor at each level is larger than the corresponding rank; and (ii) the mode on which we are operating is found at or near the top of the tree. Fortunately, CSF places no restriction on the ordering of modes. Indeed, constructing a unique CSF representation for each mode of the tensor was used in other kernels to expose parallelism and to reduce communication costs (see Chapter 4 and Chapter 8).

9.2.1 Allocating Multiple CSF Representations

We can construct multiple CSF representations in order to minimize the required number of operations. Utilizing multiple CSF representations allows computations to occur near the roots of the tree structures while also favoring mode orderings which result in large branching factors.

There are $N!$ possible orderings of the tensor modes. To evaluate the cost of a representation, we must sort the non-zeros in order to inspect the tree structure and count the number of nodes. Thus, an exhaustive search is impractical for even small values of N. We begin from an existing heuristic: sort the modes by their lengths, with the shortest mode placed at the top level (see Chapter 4). The intuition behind this heuristic is that ordering shorter modes prior to longer ones discovers indices with high levels of overlap, resulting in a large branching factor.

Suppose there is memory available for up to K representations of the tensor data, denoted $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_K$. We select $\boldsymbol{\mathcal{X}}_1$ by sorting the modes as previously discussed. The remaining $K-1$ representations are selected in a greedy fashion: at step k, use Equation (9.2) to examine the costs associated with TTMc for each mode when provided with $\boldsymbol{\mathcal{X}}_1, \ldots, \boldsymbol{\mathcal{X}}_{k-1}$. The mode with the highest cost is placed at the top level of $\boldsymbol{\mathcal{X}}_k$, and the remaining modes are sorted by increasing length. At the end of this procedure, each mode has the choice of K representations to use for TTMc computation. We assign each mode to the representation with the lowest cost, and use that representation for TTMc. Importantly, if ties are broken in a consistent manner, then it happens in practice that several modes can be assigned to the same $\boldsymbol{\mathcal{X}}_k$, meaning that fewer than K representations need be kept in memory for computation.

9.2.2 Reconstructing CSF Representations

It may not be feasible to store multiple representations of the input data in a memory-constrained environment. When the savings from a dedicated CSF representation are greater than the cost of reconstructing the CSF, then speedups can still be achieved. Instead of pre-allocating K CSF representations of the data, we can simply reconstruct a dedicated CSF before each TTMc operation. We denote this reconstruction strategy as CSF-R.

The total memory needed for CSF-R is the maximum size of the N CSF representations and the size of the original uncompressed data. This closely matches the cost of only using one compressed representation, which requires the memory for the uncompressed representation and the smallest CSF variant.

Algorithm 13 TTMc with a CSF Tensor.

function TTMc($\boldsymbol{\mathcal{X}}, mode$)
  for $i_1 = 1, \ldots, I_N$ in parallel **do**
    CONSTRUCT($\boldsymbol{\mathcal{X}}(i_1, :, \ldots, :), mode, \mathbf{1}$)
  end for
end function
▷ Construct Kronecker products and *push* them down to level $mode-1$.
function CONSTRUCT($node, mode, above$)
  $d \leftarrow$ level($node$) ▷ The level in the tree (i.e., distance from the root).
  $i_d \leftarrow$ node_id($node$) ▷ The partial coordinate of a non-zero.

  if $d <$ mode **then**
    $above \leftarrow above \otimes \mathbf{A}^{(d)}(i_d, :)$
    for $c \in$ children($node$) **do**
      CONSTRUCT(c, mode, $above$)
    end for

  else if $d =$ mode **then**
    $below \leftarrow \sum_{c \in \text{children}(node)}$ ACCUMULATE(c)
    Lock mutex i_d.
    $\mathbf{Y}_{(d)}(i_d, :) \leftarrow \mathbf{Y}_{(d)}(i_d, :) + (above \otimes below)$ ▷ Update $\mathbf{Y}_{(d)}$.
    Unlock mutex i_d.
  end if
end function
▷ *Pull* Kronecker products up from the leaf nodes.
function ACCUMULATE($node$)
  $i_d \leftarrow$ node_id($node$)
  if level($node$) $= N$ **then**
    return $\boldsymbol{\mathcal{X}}(i_1, \ldots, i_d) \cdot \mathbf{A}^{(N)}(i_d, :)$
  else
    return $\mathbf{A}^{(d)}(i_d, :) \otimes \sum_{c \in \text{children}(node)}$ ACCUMULATE(c)
  end if
end function

9.3 Experimental Methodology

9.3.1 Experimental Setup

Experiments are conducted on the Mesabi supercomputer at the Minnesota Supercomputing Institute. Compute nodes have two twelve-core Intel Haswell E5-2680v3 processors and 256GB of RAM. Our source code is written in C and parallelized with OpenMP. All source code is configured to use double-precision floating-point numbers and 32-bit integers. We compile with the Intel compiler version 16.0.3 and Intel MKL for BLAS/LAPACK routines. We bind threads to cores via the Intel OpenMP configuration `KMP_AFFINITY=granularity=fine,compact,1`.

Reported runtimes are the arithmetic mean of twenty iterations. Unless otherwise noted, we measure only the time spent on TTMc, as that is the focus of this study and the remaining computational steps do not differ between the implementations. Reported times and speedups are based on performing all of the required computations for TTMc over a full HOOI iteration. Measuring a full HOOI iteration instead of individual kernels allows us to compare memoized and non-memoized algorithms.

We compare against two algorithms implemented in the C++ library HyperTensor [40], the state-of-the-art parallel software for the Tucker decomposition. HyperTensor uses MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism. The efficient distributed-memory algorithm used by HyperTensor combines the communication steps associated with the TTMc and the following truncated SVD, preventing us from measuring the runtime corresponding to only TTMc. Thus, we run HyperTensor with one MPI rank and twenty-four OpenMP threads. We denote the two algorithms as HT-FLAT, which is a direct implementation of the elementwise formulation in Equation (3.3), and HT-BTREE, which uses memoization via binary dimension trees.

9.3.2 Datasets

Table 9.1 provides an overview of the datasets used in our evaluation. NELL-2 is from the Never Ending Language Learning project [45] and its modes represent *entities*, *relations*, and *entities*. Netflix [44] is constructed from movie ratings and has modes representing *users*, *movies*, and *dates*. Enron [70] is parsed from an email corpus spanning

Table 9.1: Summary of datasets.

Dataset	Modes	Non-zeros	Dimensions
NELL-2 [45]	3	77M	12K, 9K, 29K
Netflix [44]	3	100M	480K, 18K, 2K
Enron [70]	4	54M	6K, 6K, 244K, 1K
Alzheimer [71]	5	6.27M	5, 1K, 156, 1K, 396
Poisson3D, Poisson4D [72]	3,4	100M	3K, . . . , 3K

K and **M** stand for thousand and million, respectively.

three years. Its non-zero values are word frequency and its modes represent *senders*, *receivers*, *words*, and *dates*. Alzheimer is constructed from public gene expression data related to Alzheimer's disease, provided by MSigDB [71]. Its values are binary and its five modes represent cell type, drug, binned dosage, gene, and binned amplitudes. Poisson is a set of synthetically-generated tensors whose values follow a Poisson distribution. We generated tensors following the method of Chi and Kolda [72] with three and four modes of length 3000 and 100-million non-zeros. All tensors except Netflix and Alzheimer are freely available as part of the FROSTT collection [58].

9.4 Results

9.4.1 Operation Efficiency

Figure 9.2 shows the number of FLOPs required to perform TTMc. HT-FLAT (coordinate format) is used as a baseline because a CSF tensor will match its complexity if it achieves no compression.

A single CSF representation (CSF-1) reduces computational costs by 59%−83% compared to the baseline. Interestingly, CSF-1 is nearly identical in cost to the memoized HT-BTREE algorithm on the three-mode datasets. This is due to the limited amount of memoization possible for a three-mode tensor: one TTMc is computed at full cost and is used to optimize the remaining two operations. This matches the limitation of CSF-1, in which the leaf-level mode must still be computed at full cost. Optimizing for the leaf mode by using CSF-2 is sufficient to achieve the best-possible FLOP performance on

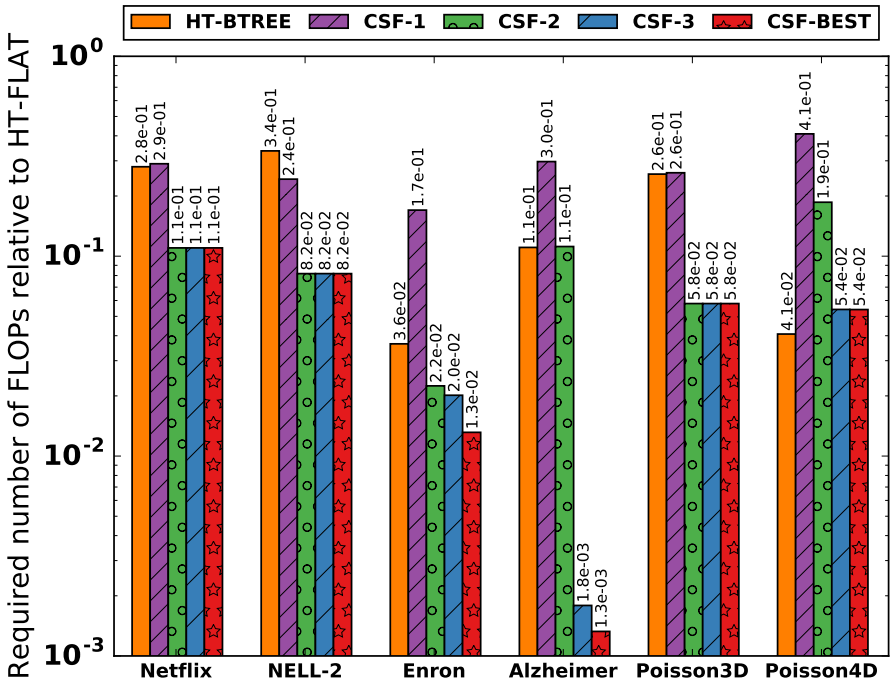


Figure 9.2: The number of required FLOPs for rank-20 TTMc on all modes, relative to HT-FLAT (i.e., coordinate form). **CSF-K** is the solution found using K CSF representations. No dataset utilized more than four CSF representations. **CSF-BEST** is the optimal configuration using multiple CSF representations, found by exhaustive search.

all three-mode tensors.

Both HT-BTREE and the CSF variants improve over HT-FLAT as the number of modes increase, because additional tensor modes bring additional TTMc operations which can be optimized. The benefits of CSF are most apparent on the five-mode Alzheimer tensor, in which the greedily-selected CSF-A requires 555× fewer FLOPs than HT-FLAT and 61× fewer FLOPs than HT-BTREE.

Observe that HT-BTREE is more operation-efficient than CSF-based methods on the synthetic Poisson4D tensor. The number of $\boldsymbol{\mathcal{X}}(i_1, i_2, :, \ldots, :)$ sub-tensors is 88% of the total number of non-zeros, meaning that the redundancies that CSF exploits do not exist in the lower levels of the tree.

9.4.2 Parallel Scalability

Figure 9.3 shows speedup as we scale from 1 to 24 cores. We include results for CSF-A which dedicates a CSF representation for each mode of the tensor, despite fewer representations being sufficient in terms of FLOP efficiency. CSF-A allows us to measure performance without fine-grained synchronization overheads because there are no race conditions to consider when the output mode is located at the root level of the tree.

Synchronization overheads prevent CSF-1 from scaling beyond one CPU socket, whereas additional CSF representations achieve near-linear scaling. The cost of synchronization dominates when computing for the bottom levels of the CSF structure: there are more nodes present in the tree (i.e., more synchronizations) and also the amount of work performed during synchronizations exponentially increases.

All methods exhibit poor scalability on the Alzheimer tensor. This is attributed to its unusually short dimensions; the presented methods parallelize over the outer dimensions of the tensor and thus have idle threads when the outer dimension is small. This limitation has also been observed in other tensor kernels [51], and has been remedied via alternative parallel decompositions [39].

9.4.3 Runtime and Memory Trade-Offs

Figure 9.4 shows the memory costs and average runtime for TTMc. We measure memory consumption via instrumented source code which tracks the storage used for the tensor structure, thread-local storage, and memoization. We omit the storage dedicated to the factor matrices and output because they are the same between methods.

Despite CSF-A not providing additional computational savings, we can see that it always achieves the best runtime across all datasets and algorithms. This is expected due to its lack of synchronization overheads and structured writes to memory. CSF-A ranges from $1.5\times - 20.7\times$ faster than HT-BTREE, and also uses less memory for four of the six datasets. We note that while Poisson4D is the only tensor for which memoization achieves a better operation reduction than the CSF variants, but CSF-A is $1.5\times$ faster in runtime.

We can see the benefit of supporting a flexible number of CSF representations. CSF-1 is always the most space-efficient, while CSF-A is always the fastest algorithm. CSF-2

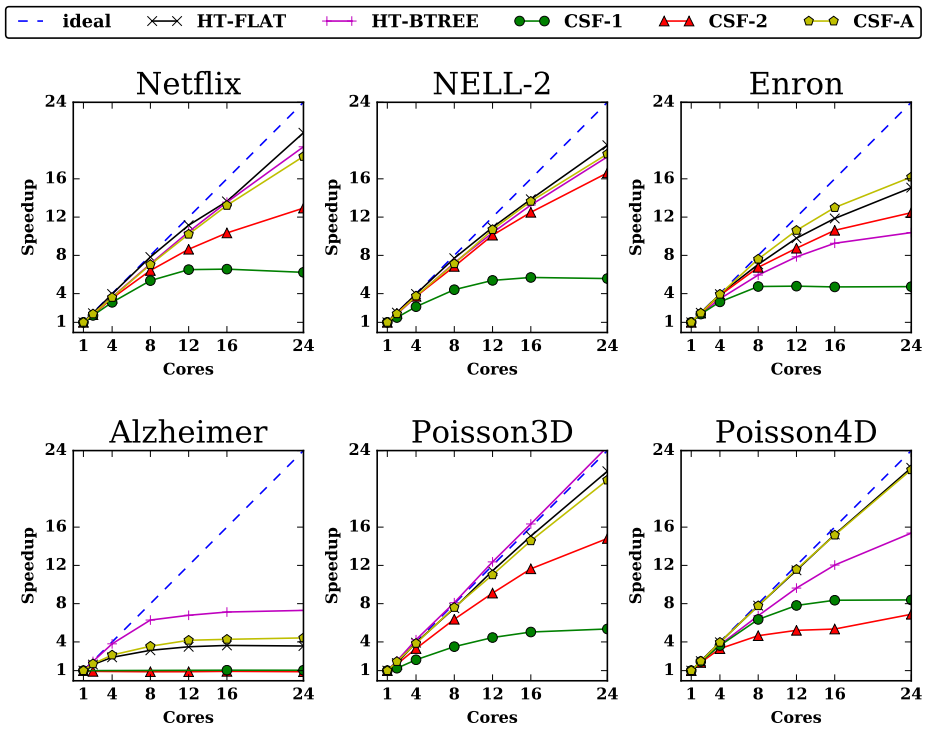


Figure 9.3: Parallel speedup for rank-20 TTMc. **CSF-A** denotes dedicating one CSF representation for each mode of the tensor.

provides a reasonable trade-off when time and space are both limited by dedicating a special CSF representation to the most expensive mode which will also exhibit the highest synchronization costs.

9.4.4 Reconstructing CSF Representations

Figure 9.5 evaluates reconstructing a CSF representation prior to each TTMc kernel instead of precomputing K representations. Runtimes for CSF-R include both TTMc computation and CSF reconstruction. We omit the CSF construction time for the variants that store multiple CSF representations, as the cost is amortized over multiple iterations.

We first note that CSF-R improves the total runtime over CSF-1 for all datasets while having a similar memory footprint. However, the CSF construction overhead

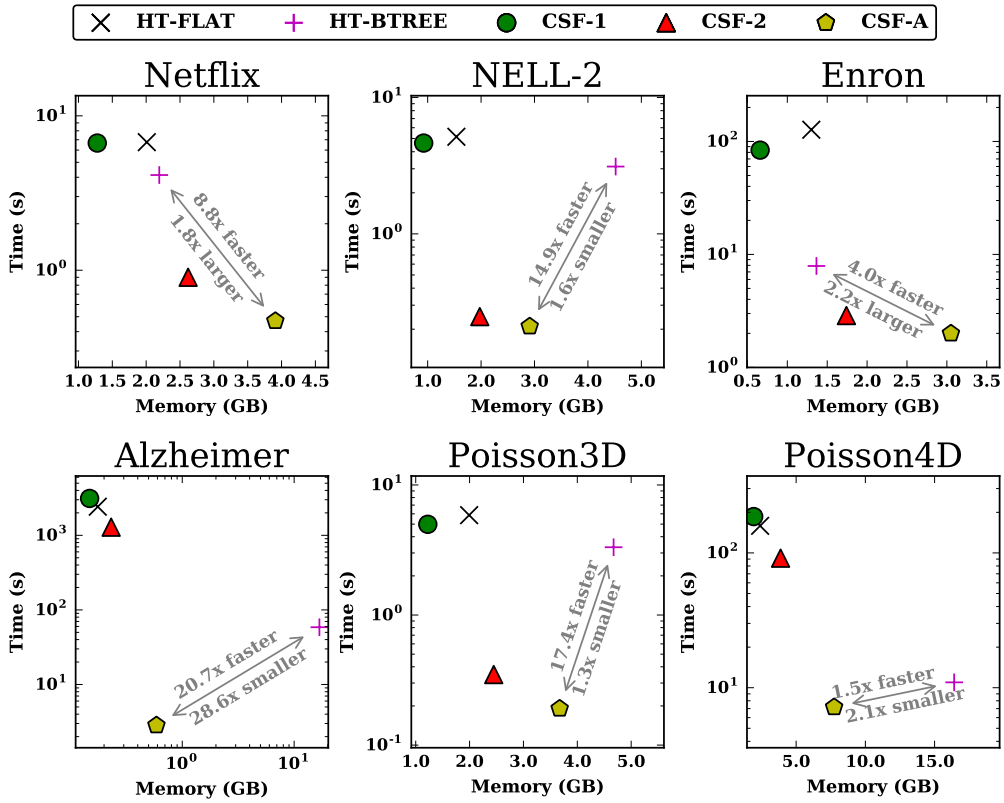


Figure 9.4: Time and space trade-offs for rank-20 TTMc on 24 cores. **Time** is the mean number of seconds spent on TTMc during a full iteration of HOOI. **Memory** is the storage required for the tensor memoization, and structures for parallelism.

consumes more time than TTMc, and thus the total runtime is generally not improved over schemes that use at least two CSF representations. CSF-R is more competitive as the number of tensor modes is increased and serves as a good default selection for users that wish to have moderate performance and memory consumption.

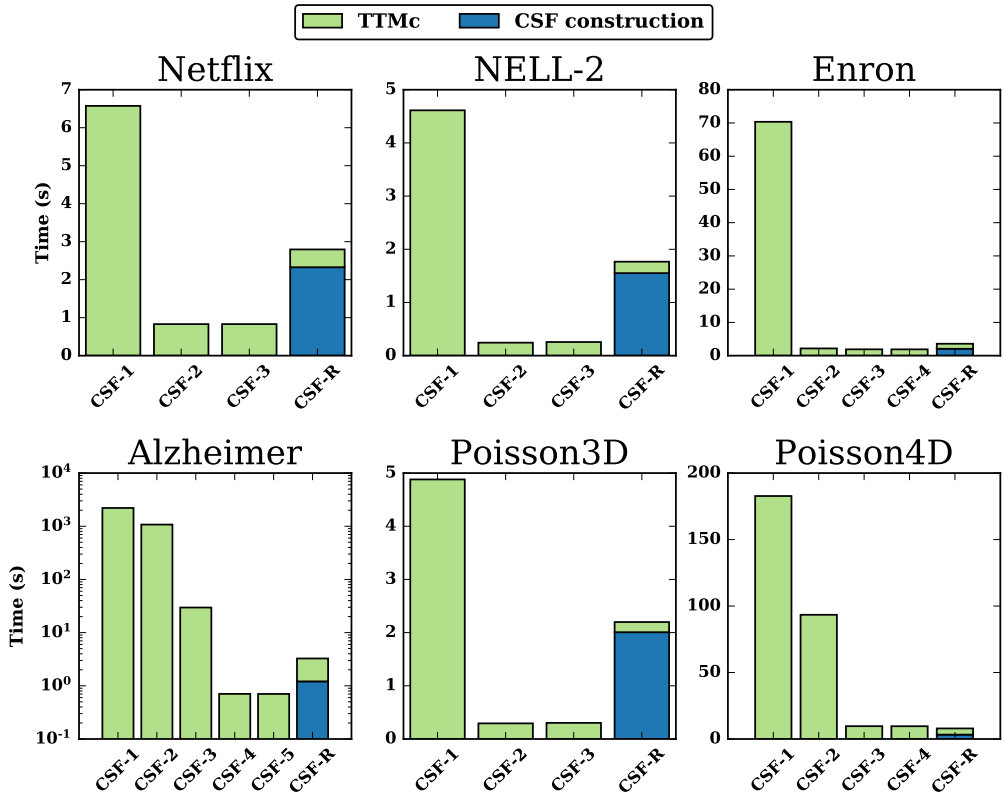


Figure 9.5: Evaluation of CSF reconstruction for memory-efficient TTMc during a rank-20 factorization with 24 threads. **CSF-K** denote using K CSF representations following the greedy algorithm presented in Section 9.2.1. **CSF-R** denotes reconstructing a specialized CSF representation before each TTMc operation.

Chapter 10

Conclusion

Multi-dimensional arrays, or *tensors*, are increasingly found in fields such as cybersecurity, social network analysis, and recommender systems. Real-world tensors can be enormous in size and often sparse. There is a need for efficient, high-performance tools capable of processing the massive sparse tensors of today and the future. In this thesis, we presented various algorithms for enabling and accelerating sparse tensor computations on modern parallel machines.

In Chapter 4, we presented an algorithm that reduces the number of operations and memory accesses required to perform MTTKRP by exploiting the tensor structure of the data. We then introduced further optimizations including a method of reordering sparse tensors and cache tiling to improve data locality. We developed a data structure for representing sparse tensors that exposes the opportunities for reducing the number of required operations. The data structure enables the operation-efficient algorithms and also has memory-efficient variants that eliminate the need for multiple representations of the tensor.

In Chapter 5, we presented an exploration of sparse tensor factorization on a many-core processor, using the Xeon Phi Knights Landing processor as a case study. We addressed challenges such as exploiting multiple levels of parallelism, managing high-bandwidth memory, load balancing hundreds of threads, and reducing fine-grain synchronization. We showed that no parallelization or synchronization strategy works consistently across datasets and provided guidelines for deciding which strategies to employ that take into account various tensor properties. Our evaluation highlighted the need

for improved hardware atomics on many-core architectures.

In Chapter 6, we introduced a medium-grained decomposition for sparse tensor factorization on distributed-memory systems. We presented a method of distributing the sparse tensor and the factor matrices and developed efficient algorithms for factoring tensors distributed in this fashion. The medium-grained decomposition addresses the limitations of coarse-grained methods by avoiding complete replication and communication of the factor matrices, while also improving load balance. The medium-grained decomposition addresses the limitations of fine-grained methods by trading off additional communication volume for fewer exchanged messages, which in practice can be the performance limiter on modern interconnects. Lastly, the medium-grained decomposition does not require computationally expensive pre-processing such as hypergraph partitioning to have a low communication volume.

In Chapter 7, we studied the acceleration and high performance implementation of AO-ADMM, a recent framework for constrained tensor factorization. We presented a form of parallelization and two optimizations which together accelerate the complete factorization process. First, a blockwise reformulation improves performance by creating temporal cache locality, eliminating parallel synchronization costs, and accelerating convergence. Second, we exploit the sparsity that dynamically emerges in the factorization output to reduce operations and memory bandwidth in the primary tensor kernel, achieving additional speedup.

In Chapter 8, we explored the design and implementation of three optimization algorithms for tensor completion: ALS, SGD, and CCD++. We focused on modern architectures with shared- and distributed-memory parallelism. We solved issues such as memory- and operation-efficiency, cache locality, load balance, and communication. We further improved the convergence rates of ALS and CCD++ by introducing randomization during the optimization procedure. When comparing algorithms for tensor completion, time-to-solution is the most important detail for end users. We compared convergence rates in three configurations: serial, a multi-core system, and a large-scale distributed system and showed that no algorithm performs best in all three environments. SGD is most competitive in a serial environment, ALS is recommended for shared-memory systems, and both ALS and CCD++ are competitive on distributed systems.

In Chapter 9, we presented an operation- and memory-efficient algorithm for computing the Tucker decomposition. The algorithm uses the compressed data structure that was presented in Chapter 4 to eliminate redundant computations while improving parallelism and memory access patterns. Furthermore, we presented a method of tuning the trade-off between the time and memory footprint of the computation. Thus, users can have either the fastest execution, the smallest memory footprint, or in-between the two extremes.

References

[1] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Alan Hanjalic, and Nuria Oliver. Tfmap: optimizing map for top-n context-aware recommendation. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 155–164. ACM, 2012.

[2] Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C. Denny, Abel Kho, You Chen, Bradley A. Malin, and Jimeng Sun. Rubik: Knowledge guided tensor factorization and completion for health data analytics. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1265–1274. ACM, 2015.

[3] Hadi Fanaee-T and João Gama. Tensor-based anomaly detection: An interdisciplinary survey. *Knowledge-Based Systems*, 98:130–147, 2016.

[4] Fatemeh Sheikholeslami and Georgios B. Giannakis. Overlapping community detection via constrained parafac: A divide and conquer approach. In *International Conference on Data Mining*. IEEE, 2017.

[5] Shaden Smith and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. `http://cs.umn.edu/~splatt/`, 2016.

[6] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[7] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, 2017.

[8] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[9] Kejun Huang, Nicholas D. Sidiropoulos, and Athanasios P. Liavas. A flexible and efficient algorithmic framework for constrained matrix and tensor factorization. *IEEE Transactions on Signal Processing*, 64(19):5052–5065, 2016.

[10] Kejun Huang, Nicholas D. Sidiropoulos, and Athanasios P. Liavas. Efficient algorithms for 'universally' constrained matrix and tensor factorization. In *Signal Processing Conference (EUSIPCO), 2015 23rd European*, pages 2521–2525. IEEE, 2015.

[11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[12] Weijia Shao. Tensor completion. Master's thesis, Universität des Saarlandes Saarbrücken, 2012.

[13] Lars Karlsson, Daniel Kressner, and André Uschmajew. Parallel algorithms for tensor completion in the CP format. *Parallel Computing*, 2015.

[14] Kijung Shin and U Kang. Distributed methods for high-dimensional and large-scale tensor factorization. In *Data Mining (ICDM), 2014 IEEE International Conference on*, pages 989–994, Dec 2014.

[15] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, 41(3):793–819, 2014.

[16] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. Cubesvd: a novel approach to personalized web search. In *Proceedings of the 14th international conference on World Wide Web*, pages 382–390. ACM, 2005.

[17] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[18] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[19] Woody Austin, Grey Ballard, and Tamara G. Kolda. Parallel tensor compression for large-scale scientific data. In *International Parallel & Distributed Processing Symposium (IPDPS'16)*, pages 912–922. IEEE, 2016.

[20] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, , and Dheeraj Sreedhar. On optimizing distributed tucker decomposition for dense tensors. *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*, 2017.

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

[22] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 2.5. `http://www.sandia.gov/~tgkolda/TensorToolbox/`, January 2012.

[23] Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. Tensorlab v2.0. `http://www.tensorlab.net/`, January 2014.

[24] U Kang, Evangelos E. Papalexakis, Abhay Harpale, and Christos 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*, pages 316–324. ACM, 2012.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] Joon Hee Choi and S Vishwanathan. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*, pages 1296–1304, 2014.

[27] Oguz Kaya and Bora Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 77. ACM, 2015.

[28] Qiang Zhang, Michael W. Berry, Brian T. Lamb, and Tabitha Samuel. A parallel nonnegative tensor factorization algorithm for mining global climate data. In *Computational Science–ICCS 2009*, pages 405–415. Springer, 2009.

[29] Athanasios P. Liavas and Nicholas D. Sidiropoulos. Parallel algorithms for constrained tensor factorization via alternating direction method of multipliers. *IEEE Transactions on Signal Processing*, 63(20):5450–5463, 2015.

[30] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. A high-performance parallel algorithm for nonnegative matrix factorization. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 9. ACM, 2016.

[31] Guoxu Zhou, Andrzej Cichocki, Qibin Zhao, and Shengli Xie. Nonnegative matrix and tensor factorizations: An algorithmic perspective. *IEEE Signal Processing Magazine*, 31(3):54–65, 2014.

[32] Andrzej Cichocki, Rafal Zdunek, Anh Huy Phan, and Shun-ichi Amari. *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation.* John Wiley & Sons, 2009.

[33] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

[34] Mark Gates, Hartwig Anzt, Jakub Kurzak, and Jack Dongarra. Accelerating collaborative filtering using concepts from high performance computing. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 667–676. IEEE, 2015.

[35] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis 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*, pages 69–77. ACM, 2011.

[36] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[37] Fabio Petroni and Leonardo Querzoni. GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 241–248. ACM, 2014.

[38] Tamara G. Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 363–372. IEEE, 2008.

[39] Muthu Baskaran, Benoît Meister, Nicolas Vasilache, and Richard Lethin. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6. IEEE, 2012.

[40] Oguz Kaya and Bora Uçar. High-performance parallel algorithms for the tucker decomposition of higher order sparse tensors. Technical report, 2015.

[41] Oguz Kaya and Bora Uçar. Parallel candecomp/parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018.

[42] Bruce Hendrickson and Tamara G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[43] Umit V. Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse matrix vector multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 10(7):673–693, 1999.

[44] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[45] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka, and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *In AAAI*, 2010.

[46] Tom M. Mitchell, Svetlana V. Shinkareva, Andrew Carlson, Kai-Min Chang, Vicente L. Malave, Robert A. Mason, and Marcel Adam Just. Predicting human brain activity associated with the meanings of nouns. *science*, 320(5880):1191–1195, 2008.

[47] Olaf Görlitz, Sergej Sizov, and Steffen Staab. Pints: peer-to-peer infrastructure for tagging systems. In *IPTPS*, page 19, 2008.

[48] Julian McAuley, Jure Leskovec, and Dan Jurafsky. Learning attitudes and attributes from multi-aspect reviews. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 1020–1025. IEEE, 2012.

[49] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[50] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[51] Thomas B. Rolinger, Tyler A. Simon, and Christopher D. Krieger. Performance evaluation of parallel sparse tensor decomposition implementations. In *Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 2016.

[52] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights

Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[53] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[54] Hansang Bae, James Cownie, Michael Klemm, and Christian Terboven. A user-guided locking API for the OpenMP* application program interface. In *International Workshop on OpenMP*, pages 173–186. Springer, 2014.

[55] Center for Medicare and Medicaid Services. CMS data entrepreneurs synthetic public use file (DE-SynPUF), 2010.

[56] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The yahoo! music dataset and kdd-cup'11. In *KDD Cup*, pages 8–18, 2012.

[57] Jason Baumgartner. Reddit comment dataset. `https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/`, 2015.

[58] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools. `http://frostt.io/`, 2017.

[59] Daniël M. Pelt and Rob H. Bisseling. A medium-grain method for fast 2d bipartitioning of sparse matrices. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 529–539. IEEE, 2014.

[60] Umit V. Catalyurek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 32(2):656–683, 2010.

[61] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.

[62] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[63] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[64] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. Parallel hypergraph partitioning for scientific computing. IEEE, 2006.

[65] Gene H. Golub and Charles F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[66] Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[67] Olivier Fercoq and Peter Richtárik. Accelerated, parallel, and proximal coordinate descent. *SIAM Journal on Optimization*, 25(4):1997–2023, 2015.

[68] Roberto Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex systems*, 3(4):331–342, 1989.

[69] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[70] Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, 4, 2004.

[71] Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.

[72] Eric C Chi and Tamara G Kolda. On tensors, sparsity, and nonnegative factorizations. *SIAM Journal on Matrix Analysis and Applications*, 33(4):1272–1299, 2012.