CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms

Zachary Blanco *
Rutgers University
Piscataway, NJ, USA
zac.blanco@rutgers.edu

Bangtian Liu *
Rutgers University
Piscataway, NJ, USA
bangtian.liu@rutgers.edu

Maryam Mehri Dehnavi
Rutgers University
Piscataway, NJ, USA
maryam.mehri@rutgers.edu

ABSTRACT

Tensors, or N-dimensional arrays, are increasingly used to represent multi-dimensional data. Sparse tensor decomposition algorithms are of particular interest in analyzing and compressing big datasets due to the fact that most of real-world data is sparse and multi-dimensional. However, state-of-the-art tensor decomposition algorithms are not scalable for overwhelmingly large and higher-order sparse tensors on distributed platforms. In this paper, we use the MapReduce model and the Spark engine to implement tensor factorizations on distributed platforms. The proposed CSTF algorithm, *Cloud-based Sparse Tensor Factorization*, is a scalable distributed algorithm for tensor decompositions for large data. It uses the coordinate storage format (COO) to operate on the tensor nonzeros directly, thus, eliminating the need for tensor unfolding and the storage of intermediate data. Also, a novel queuing strategy (QCOO) is proposed to exploit the dependency and data reuse between a sequence of tensor operations in tensor decomposition algorithms. Details on the key-value storage paradigm and Spark features used to implement the algorithm and the data reuse strategies are also provided. The queuing strategy reduces data communication costs by 35% for 3rd-order tensors and by 31% for 4th-order tensors over the COO-based implementation respectively. Compared to the state-of-the-art work, BIGtensor, CSTF achieves 2.2× to 6.9× speedup for 3rd-order tensor decompositions.

ACM Reference Format:
Zachary Blanco, Bangtian Liu, and Maryam Mehri Dehnavi. 2018. CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225133

1 INTRODUCTION

Tensors, or multi-dimensional vectors, naturally lend themselves to representing multi-dimensional data. Tensor decomposition algorithms appear in numerous domains and applications such as data mining [11, 16], machine learning [1, 9], computer vision [27, 28], and quantum chemistry [15]. Recently, the size of tensor data has become overwhelmingly large, including tens to hundreds of millions and even billions of nonzeros in real tensor-based applications. This has demanded the need for developing novel algorithms and frameworks that implement tensor operations on parallel and distributed systems for better performance and scalability. Specifically, implementations of tensor factorization algorithms on fault-tolerant frameworks such as Hadoop [8] and Spark [30] are useful as they can execute in data-center settings.

Many successful advances have been made to scale large tensor decomposition algorithms to distributed platforms using the MapReduce paradigm [5] some of which are GigaTensor [11], HATEN2 [10], and BIGtensor [19]. The previous MapReduce implementations of tensor algorithms such as BIGtensor, use the Hadoop framework [8]. These implementations do not support higher-order tensors and are implemented with tensor unfolding which creates large memory footprints. The data-reuse and locality amongst different tensor operations in tensor factorization methods are not exploited efficiently in these implementations.

In this paper we propose CSTF, *Cloud-based Sparse Tensor Factorization*, a scalable distributed algorithm for tensor decompositions on large data. CSTF uses the open source Apache Spark [30] platform, an extension to the MapReduce [5] framework. Our work focuses on the performance optimization of the CANDECOMP/-PARAFAC (CP) decomposition on Spark. The running time of a typical CP decomposition on an N-order tensor is dominated by a sequence of Matricized Tensor Times Khatri-Rao product (MTTKRP) operations along each mode of the tensor. The mode-centric nature of the tensor computations is a major challenge when designing high-performance algorithms for higher-order sparse tensors. Previous implementations of tensor algorithms on distributed systems [4, 11–13] are mainly based on improving the performance of a single tensor operation such as the MTTKRP. In this work, we introduce algorithms to optimize the performance of an entire sequence of MTTKRP operations within the CP decomposition. This leads to a scalable implementation of higher-order tensor decompositions. To the best of our knowledge, we are the first to investigate sparse tensor CP decompositions for tensors of order 3 or higher on Spark. Major contributions of the work are as follow:

1. The CSTF-COO algorithm which uses the key-value storage paradigm to enable explicit computations on the tensor nonzeros. Our proposed algorithm uses the coordinate storage format (COO) to eliminate the need to unfold the tensor and to reduce redundant computation in tensor operations such as MTTKRP.
2. The CSTF-QCOO algorithm detects data reuse between different MTTKRP operations inside tensor factorization methods. CSTF-QCOO uses the key-value paradigm as well as data-persistence

*Both authors contributed equally to the paper

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225133

capabilities provided by Spark. As a result, data communication over the network is reduced in the tensor factorization algorithm.
3. We compare the performance of CSTF on up to 32 nodes with the state-of-the-art implementations of tensor algorithms using the MapReduce model, namely BIGtensor. CSTF-COO and CSTF-QCOO achieve up to 6.9× and 6.5× speedup for 3rd-order CP decompositions over BIGtensor respectively. The data-reuse strategy in the CSTF-QCOO algorithm reduces the amount of shuffled data by up to 35 percent compared to CSTF-COO.

The rest of this paper is organized as follows. Section II introduces notations and provides a brief introduction to tensor computations. Section III reviews state-of-the-art approaches to the optimization of tensor decompositions. Section IV describes the proposed CSTF algorithms, including CSTF-COO and CSTF-QCOO. Section V provides the experimental evaluations and results analysis. Section VI concludes the work.

2 BACKGROUND

This section presents preliminary definitions and notations for tensor computations and discusses the Spark framework and the MapReduce programing models.

2.1 Tensor Notation

A tensor can be thought of as a multidimensional array. The order of a tensor is the number of dimensions, also known as ways or modes. First-order tensors, vectors, are represented by lowercase letters, e.g. a. Second-order tensors, matrices, are shown with boldface capital letters, e.g. $\mathbf{A}$. Tensors of order three or higher are denoted by boldface Euler script letters, e.g. $\mathcal{X}$. Scalars are represented by lowercase letters, e.g. a, and the scalar element at position (i, j, k) of a third-order tensor $\mathcal{X}$ is denoted by $\mathcal{X}(i, j, k)$. We use the colon notation, where a colon represents all nonzeros in an index on that mode. For example, $\mathbf{A}(m, :)$ represents the m-th row of the matrix $\mathbf{A}$. Table 1 summarizes the notations used in this paper. We use I, J, K to represent the dimensions of a 3-order tensor.

Matricization, also known as *unfolding* or *flattening*, is the process of reordering the elements of an N-way array into a matrix. The mode-n matricization of a tensor $\mathcal{X}$ is shown with $\mathbf{X}_{(n)}$ and arranges the mode-n *fibers* to be the columns of the resulting matrix. A *fiber* of a tensor is defined by fixing every index but one. A three-way tensor has three kinds of *fibers*, denoted by $\mathcal{X}(:, j, k)$, $\mathcal{X}(i, :, k)$ and $\mathcal{X}(i, j, :)$. Given a three-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, $\mathbf{X}_{(1)}$ is the mode-1 matricization is of dimension $I \times JK$.

2.2 CANDECOMP/PARAFAC Decomposition

The CANDECOMP/PARAFAC (CP) decomposition algorithm factorizes a tensor into a sum of rank-one tensors. The most commonly used approach for computing the CP decomposition is the Alternating Least Squares (ALS) method. The ALS method for a 3rd-order tensor has three steps. Each step performs an update for one of the three factor matrices by keeping the other two matrices as shown in Algorithm 1. The result matrices from tensor factorization $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are called the *factor matrices.*

Table 1: Table of symbols

Symbol	Definition
$\mathcal{X}$	A tensor
$\mathbf{X}_{(n)}$	Mode-n matricization of a tensor
R	Rank of a tensor
N	Order of a tensor
nnz	nonzeros of a tensor $\mathcal{X}$
$\odot$	Khatri-Rao product
$\otimes$	Kronecker product
$*$	Hadamard product
$\mathbf{A}^T$	Transpose of matrix $\mathbf{A}$
$\mathbf{M}^\dagger$	Pseudoinverse of matrix $\mathbf{M}$
$bin()$	function that converts nonzeros elements of to 1

Algorithm 1 CP-ALS for a 3rd-order tensor

Require: $\mathcal{X}$: A 3rd order tensor R: The rank of factorization
Ensure: $[\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]$
repeat
  $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^\top \mathbf{B} * \mathbf{C}^\top \mathbf{C})^\dagger$
  Normalize columns of $\mathbf{A}$ and store the norms as λ
  $\mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{A}^\top \mathbf{A} * \mathbf{C}^\top \mathbf{C})^\dagger$
  Normalize columns of $\mathbf{B}$ and store the norms as λ
  $\mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{A}^\top \mathbf{A} * \mathbf{B}^\top \mathbf{B})^\dagger$
  Normalize columns of $\mathbf{C}$ and store the norms as λ
until stop criterion satisfied or maximum iterations reached

2.3 Matricized Tensor Times Khatri-Rao Product

MTTKRP is the key tensor operation and a compute-intensive operation in the CP decomposition algorithm. Equation 1 shows MTTKRP operations along the first mode of the tensor, meaning that the unfolded tensor along the first mode gets multiplied with the Khatri-Rao product of factor matricies $\mathbf{B}$ and $\mathbf{C}$:

$$\mathbf{M} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \tag{1}$$

If tensor $\mathcal{X}$ is of size $I \times J \times K$, then matrices $\mathbf{B}$ and $\mathbf{C}$ are of size $J \times R$ and $K \times R$. The result matrix of explicitly constructing the Khatri-Rao product $\mathbf{C} \odot \mathbf{B}$ is a dense matrix of size $J \times K \times R$, which is very large and is defined as the *intermediate data explosion* problem in [11].

2.4 Spark and the MapReduce Programming Model

MapReduce [5] is a programming model which defines *Map* and *Reduce* functions that are capable of processing records composed of *key-value* pairs. Key-value pairs represent data with an identifier, a *key*, and some associated data, the *value*. Spark [30] is an extension of the MapReduce programming model [5] which uses an abstraction called RDDs [29] to represent datasets. The dataset representation used by Spark, an RDD, is an immutable collection of records which may be composed of a key-value pair or simply a value which can have records partitioned across multiple processors connected through a network. Operations such as map, filter, join, reduce, and more are performed on an RDD. These RDD operations are classified as *transformations* and *actions*. Transformations apply a single function to many data items such as map, filter, and join. By contrast, *actions* require computations to be performed and data returned to the user, such as in the *reduce* function. Spark extends MapReduce by realizing a directed-acyclic-graph (DAG) representation for datasets. An RDD can be represented as a key-value pair and may be *partitioned* across processors based on the key in each record. Depending on the transformations performed, data from separate partitions may need to be *shuffled*. A *shuffle* is an operation which requires data from one or more partitions to be on the same processor to complete an operation. Examples of these types of operations are a *join* where records from two RDDs with similar keys are combined and *reduceByKey* which combines records with the same key in the same RDD together into a single record. Depending on which processor each record is located on, performing one of these operations will induce a *shuffle* where each processor will decide which records must be transmitted over the network. Shuffles are often expensive operations because they require data communication over the network.

3 RELATED WORK

A large class of previous work has proposed high-performance implementations of tensor algorithms on shared memory architectures [17], co-processors such as GPUs [18], and MPI-based implementations on distributed memory platforms. DfacTo [4] accelerates tensor decomposition methods on distributed platforms by using the Message Passing Interface (MPI). DfacTo lowers MTTKRP into two successive sparse matrix-vector multiplication (SpMV) operations to improve the performance of tensor decomposition methods. A hyper-graph partition-based method for the distributed implementation of tensor algorithms is presented by Kaya *et. al.* [13] to maintain an efficient trade-off between load balance and communication costs. DMS [23], based on SPLATT [25], proposes a new distributed CPD-ALS algorithm where a 3D decomposition is used to avoid complete factor replication and communication. A hybrid MPI+OpenMP implementation is used in DMS. Other work such as [24] and Shaden *et. al.* present decomposition techniques that avoid complete factor replication and communication in tensor computations and eliminate costly pre-processing steps. Kaya *et. al.* [14] proposes a novel computational scheme using dimension trees to effectively parallelize MTTKRPs in CP-ALS on shared and distributed memory architectures. Tensor computations have also been optimized for GPU architectures for key tensor operations in tensor factorization algorithms [18] and tensor contraction [21]. Liu *et al.* [18] proposed a new storage format called F-COO for optimizing sparse tensor computations on GPUs.

Distributed Computing on Cloud Platforms: Previous works have also provided implementations of tensor operations on distributed platforms using the MapReduce programming paradigm. MapReduce is a distributed programming model for processing massive datasets, which handles the problems of fault-tolerance, load balancing, and massive scaling automatically. Hadoop [8] is an open source implementation of MapReduc. Kang *et. al.* proposed GigaTensor [11], a large-scale tensor decomposition implementation on the Hadoop platform by utilizing the MapReduce framework. Park *et. al.* propose DBTF [20], a distributed algorithm and implementation for Boolean tensor factorization on Spark, in which Boolean tensors are composed of 0's and 1's. HATEN2 [10] is based on the MapReduce paradigm and supports two commonly used tensor factorization algorithms on Hadoop—PARAFAC and Tucker. DisTenC [6] implements a new CP-based tensor completion algorithm on Spark. In [2], a Spark+MPI system for integrating MPI-based programs with Spark is used to implement CP decomposition.

Recently, BIGtensor [19] has been introduced which is a large-scale tensor-mining library that handles a variety of tensor computations on Hadoop including tensor decomposition. BIGtensor is considered the state-of-the-art tool for distributed tensor factorizations. For distributed CP decomposition on MapReduce frameworks, BIGtensor uses a similar approach to GigaTensor [11]. Our work differs from the current implementations of tensor operations on distributed performs in that it uses Spark to provide implementations which eliminate the need to unfold the tensor, reduces the memory footprint, and also enables the reuse of factors across consecutive MTTRKP operations.

4 CLOUD-BASED SPARSE TENSOR FACTORIZATION

In this section, we introduce *Cloud-based Sparse Tensor Factorization (CSTF)*, which is a scalable algorithm for implementing tensor decompositions on distributed platforms with the MapReduce programming model using the Spark engine. CSTF optimizes tensor computations, specifically MTTKRP, to eliminate the need for tensor unfolding and to reduce the memory footprint of the implementation; we call this algorithm CSTF-COO. We also propose the CSTF-QCOO algorithm, which analyzes and exploits the dependency and locality between a sequence of tensor operations to enable efficient data reuse in distributed tensor factorizations. The following elaborates both algorithms and provides details on how the key-value storage paradigm is used in the spark engine to implement the methods efficiently on distributed systems.

4.1 CSTF-COO

CSTF-COO implements the MTTKRP operation with the COO storage format using the MapReduce programming model in Spark. In tensor decomposition algorithms, matricization across all modes of an N-order tensor requires N replications of the tensor to perform each MTTKRP. Also, constructing the Khatri-Rao product of dense matrices explicitly creates larger dense result matrices. CSTF-COO provides an implementation that eliminates explicit computation of these costly operations by fully exploiting the sparsity of the tensor. The main contributions in the CSTF-COO algorithm are the design of distributed data representations motivated from tensor computations and in-memory caching to reuse intermediate results in the MTTKRP. CSTF-COO uses the coordinate storage format to store the sparse tensor and then defines key-value pairs and operates on them to implement the MTTKRP operation.

In the COO storage format for a third-order tensor, each nonzero entry is stored with indices i, j, and k for three modes and the corresponding nonzero entries. In other words, COO stores a list of tuples including indices and values to represent all elements of the sparse tensor. Based on COO, a sparse tensor can be represented with an RDD where each element represents one nonzero entry. With a tensor stored in the COO format, MTTKRP operations can be performed as in Equations 2 and 3 derived from Equation 1. As shown in Equation 3, based on the nonzero indices and values, a row of **B** and a row of **C** are retrieved respectively and then their Hadamard product is computed and scaled with a tensor entry to update a row of **M**. This computation can be extended from 3-order tensors to N-order tensors.

$$\mathbf{M}(i,r) = \sum_{z=1}^{JK} \mathbf{X}_{(1)}(i,z)(\mathbf{C}(z/J,r)\mathbf{B}(z\%J,r)) \tag{2}$$

$$\begin{aligned}\mathbf{M}(i,:) &= \sum_{z=1}^{JK} \mathbf{X}_{(1)}(i,z)(\mathbf{C}(z/J,:) * \mathbf{B}(z\%J,:)) \\ &= \sum_{k=1}^{k=K}\sum_{j=1}^{j=J} \mathcal{X}(i,j,k)(\mathbf{C}(k,:) * \mathbf{B}(j,:))\end{aligned} \tag{3}$$

Algorithm 2 CSTF-COO mode-1 MTTKRP for a 3-order tensor

Require: :
$\mathcal{X} \in \mathbb{R}^{I\times J\times K}$: A 3-order tensor
$\mathcal{X}(i,j,k)$: A nonzero element of $\mathcal{X}$ at position (i,j,k)
B: Factor matrix of second mode
C: Factor matrix of third mode
Ensure: :
M: The result matrix of MTTKRP
$\mathbf{M} \leftarrow \mathbf{0}$
for $\mathcal{X}(i,j,k) \in \mathcal{X}$ **do**
  $\mathbf{M}(i,:) \leftarrow \mathbf{M}(i,:) + \mathcal{X}(i,j,k)[\mathbf{C}(k,:) * \mathbf{B}(j,:)]$
end for

Implementation workflow. Table 2 illustrates the workflow of implementing MTTKRP on mode-1 ($\mathbf{M} = \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$) in CSTF-COO. The basic idea of MTTKRP for a 3rd-order tensor is to fix two matrices and update the remaining matrix. For MTTKRP on mode-1, matrices **B** and **C** are fixed and the result of MTTKRP, **M**, is used to update matrix **A**. STAGE 1 in Table 2 shows transformations of the RDD based on the dependency between matrix **C** and the tensor $\mathcal{X}$ along mode-3 to perform the computation: $\mathcal{X}(i,j,k) * \mathbf{C}(k,:)$; STAGE 2 shows transformations of the RDD based on the dependency between matrix **B** and tensor $\mathcal{X}$ along mode-3 to perform the computation with the intermediate result from STAGE 1: $\mathcal{X}(i,j,k) * \mathbf{C}(k,:) * \mathbf{B}(j,:)$. STAGE 3 of CSTF-COO applies transformations to RDDs based on the dependency between the output matrix $\mathbf{M}_1$ and the tensor $\mathcal{X}$ along mode-1 to update the result matrix with intermediate results from STAGE-2: $\mathbf{M}(i,:) = \mathbf{M}(i,:) + \mathcal{X}(i,j,k) * \mathbf{C}(k,:) * \mathbf{B}(j,:)$.

Caching. As shown in Algorithm 1, tensor factorization is performed using the alternative least squares method (ALS). Factor matrices **A**, **B**, and **C** are updated repeatedly in the ALS procedure in each iteration until a stopping criterion is satisfied. Keeping the tensor in memory can improve the performance significantly since the tensor data is reused across iterations. RDDs in Spark can be *cached* by specifying a storage strategy and the data formats between intermediate stages of the DAG. Data may be cached in a *serialized* or *raw* format while choosing memory or disk as the storage strategy [31]. Serialized formats convert the internal objects into a stream of bytes which typically take up less space compared to the raw Java or Scala object representation in memory. While serialization takes less space, more CPU cycles are needed to convert the data representation. Raw caching typically requires more space but is faster at reading objects into memory when a transformation must be performed. We cache the tensors using the raw format since it leads to better performance benefits in iterative tensor algorithms such as tensor factorizations mainly due to the faster data accesses.

4.2 CSTF-QCOO

CSTF-QCOO improves upon CSTF-COO by reusing data between MTTKRP operations to reduce the amount of shuffling required. The proposed improvement is useful when the MTTKRP operation is used in a tensor factorization algorithm such as the CP-ALS where multiple MTTKRP operations are computed in each iteration and between iterations. One of the main issues of using the CSTF-COO algorithm in CP-ALS is the communication required for every MTTKRP operation because of the volume of data shuffled. With CSTF-COO, every MTTKRP for an N-order tensor requires N shuffle operations which reduces the performance of the algorithm. Algorithm 3 demonstrates CP-ALS for an N-order tensor using the queuing process.

Algorithm 3 CP-ALS for an N-way tensor with QCOO

Require: $\mathcal{X} \in \mathbb{R}^{\mathbb{I}_1\times\cdots\times\mathbb{I}_N}$: A N^{th} order sparse tensor
R: The rank of approximation
Ensure: CP decomposition $[\lambda; \mathbf{A}_1, \cdots, \mathbf{A}_N]$
$V \leftarrow \text{Queue}\{\mathbf{A}_1^T\mathbf{A}_1, \ldots, \mathbf{A}_{N-1}^T\mathbf{A}_{N-1}\}$
$Z \leftarrow \text{Queue}\{\mathbf{A}_1, \ldots, \mathbf{A}_{n-1}\}$
repeat
  for $n = 1 : N$ **do**
    dequeue(V)
    dequeue(Z)
    if $n = 1$ **then**
      $V \leftarrow \text{enqueue}(\mathbf{A}_N^T\mathbf{A}_N)$
      $Z \leftarrow \text{enqueue}(\mathbf{A}_N)$
    else
      $V \leftarrow \text{enqueue}(\mathbf{A}_{n-1}^T\mathbf{A}_{n-1})$
      $Z \leftarrow \text{enqueue}(\mathbf{A}_{n-1})$
    end if
    $\widehat{V} \leftarrow \text{reduce}(V, v_1, v_2 \rightarrow v_1 * v_2)$
    $\widehat{Z} \leftarrow \text{reduce}(Z, z_1, z_2 \rightarrow z_1 \odot z_2)$
    $\mathbf{A}_n \leftarrow \mathbf{X}_{(n)}\widehat{Z}\widehat{V}^\dagger$
    Normalize the columns of A_n and store the norms as λ
  end for
until no improvement or maximum iterations reached

Table 2: Workflow comparison between BIGtensor, CSTF-COO, and CSTF-QCOO on a 3rd-order mode-1 MTTKRP $\mathbf{M} \Leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$

Stage	BIGtensor	CSTF-COO	CSTF-QCOO
1	**Map** $(i, j_0, \mathbf{X}_{(1)}(i, j_0))$ on $\lceil \frac{j_0}{J} \rceil$, and $(k, r, \mathbf{C}(k, r))$ on k. **Reduce**: $(i, j_0, \mathbf{X}_{(1)}(i, j_0)\mathbf{C}(k, r))$	**Map** COO on k to get $(k, (i, j, k, \mathcal{X}(i, j, k)))$ **Join** $C(k, :), (k, (i, j, k, \mathcal{X}(i, j, k)), C(k, :))$	**Join** $(k, ((i, j, k, \mathcal{X}(i, j, k)),$ Queue$(A(i, :), B(j, :)))$ with $C(k, :)$
2	**Map** $(i, j_0, bin(\mathbf{X}_{(1)}(i, j_0)))$ on $(j_0 \bmod J)$ and $(j, r, \mathbf{B}(j, r))$ on j **Reduce**: $(i, j_0, bin(\mathbf{X}_{(1)}(i, j_0))\mathbf{B}(j, r))$	**Map**$(k((i, j, k\mathcal{X}(i, j, k)), C(k, :)))$ on j to get $(j(i, j, k, \mathcal{X}(i, j, k)), C(k, :)))$, **Join** $B(j, :), (j, ((i, j, k, \mathcal{X}(i, j, k), C(k, :)), B(j, :))$	**Map** Add $C(k, :)$ to the queue, dequeue $A(i, :)$ from the queue and move to the next key. **Emit** $(i, (i, j, k, \mathcal{X}(i, j, k)),$ Queue$(B(j, :), C(k, :)))$
3	**Map** $(i, j_0, \mathbf{X}_{(1)}(i, j_0)\mathbf{C}(j, r))$. and $(i, j, bin(\mathbf{X}_{(1)}(i, j))\mathbf{B}(j, r))$ on i, **Reduce**: $(i, j_0, \sum_j \mathbf{X}_{(1)}(i, j_0)\mathbf{B}(j, r)\mathbf{C}(k, r))$ **Update**: sum up columns and emit $\mathbf{M}(i, r)$.	**Map**: $(j, (i, j, k, \mathcal{X}(i, j, k), C(k, :)), B(j, :))$ on i and do $B(j, :) * C(k, :) * \mathcal{X}(i, j, k)$. **ReduceByKey**: on $(i, B(j, :) * C(k, :) * \mathcal{X}(i, j, k))$. **Update**: $M(i, :)$ with $(B(j, :) * C(k, :) * \mathcal{X}(i, j, k))$.	**MapValues**: $(i, (i, j, k, \mathcal{X}(i, j, k)),$ Queue$(B(j, :), C(k, :)))$ reduce the queue to $B(j, :) * C(k, :)$. **ReduceByKey**: on $(i, B(j, :) * C(k, :) * \mathcal{X}(i, j, k))$ **Update**: $M(i, :)$ with $B(j, :) * C(k, :) * \mathcal{X}(i, j, k)))$.

Table 3: Representation of Data as Spark RDDs

Dataset	Type	Spark RDD abstract	Element example	Implementation
$\mathcal{X}$	Sparse tensor	*RDD[Vector]*	$(i, j, k, \mathcal{X}(i, j, k))$	COO
$\mathcal{X}_Q$	Sparse tensor	*RDD[(Vector, Queue[Vector])]*	$((i, j, k, \mathcal{X}(i, j, k)), Queue(\mathbf{A}(i, :), \mathbf{B}(j, :), \mathbf{C}(k, :)))$	QCOO
$\mathbf{A}, \mathbf{B}, \mathbf{C}$	Dense factor matrices	*IndexedRowMatrix*	$(index, \mathbf{A}(index, :))$	COO, QCOO

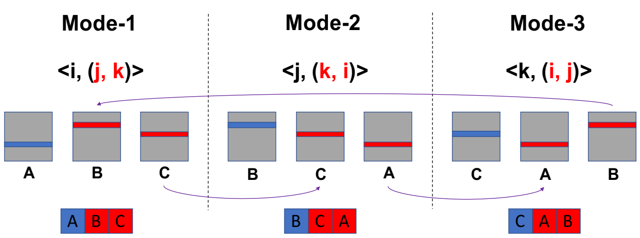


Figure 1: The data reuse pattern among three MTTKRP operations along different modes in CP decomposition. The color blue represents an index and row of the matrix that has to be updated. The color red shows the rows of factor matrices which are fixed and are used to perform the MTTKRP operation. A line with an arrow marks the reuse flow from one MTTKRP to another.

$$\widehat{\mathbf{A}} \leftarrow \mathbf{X}_{(1)}(\underbrace{\mathbf{D} \odot \mathbf{C}}_{M_1} \odot \mathbf{B})(\underbrace{\mathbf{D}^\top \mathbf{D} * \mathbf{C}^\top \mathbf{C}}_{N_1} * \mathbf{B}^\top \mathbf{B})^\dagger \tag{4}$$

$$\widehat{\mathbf{B}} \leftarrow \mathbf{X}_{(2)}(\underbrace{\mathbf{D} \odot \mathbf{C}}_{M_1} \odot \widehat{\mathbf{A}})(\underbrace{\mathbf{D}^\top \mathbf{D} * \mathbf{C}^\top \mathbf{C}}_{N_1} * \widehat{\mathbf{A}}^\top \widehat{\mathbf{A}})^\dagger \tag{5}$$

$$\widehat{\mathbf{C}} \leftarrow \mathbf{X}_{(3)}(\mathbf{D} \odot \underbrace{\widehat{\mathbf{B}} \odot \widehat{\mathbf{A}}}_{M_2})(\mathbf{D}^\top \mathbf{D} * \underbrace{\widehat{\mathbf{B}}^\top \widehat{\mathbf{B}} * \widehat{\mathbf{A}}^\top \widehat{\mathbf{A}}}_{N_2})^\dagger \tag{6}$$

$$\widehat{\mathbf{D}} \leftarrow \mathbf{X}_{(4)}(\widehat{\mathbf{C}} \odot \underbrace{\widehat{\mathbf{B}} \odot \widehat{\mathbf{A}}}_{M_2})(\widehat{\mathbf{C}}^\top \widehat{\mathbf{C}} * \underbrace{\widehat{\mathbf{B}}^\top \widehat{\mathbf{B}} * \widehat{\mathbf{A}}^\top \widehat{\mathbf{A}}}_{N_2})^\dagger l \tag{7}$$

In subsequent MTTKRP operations in CP-ALS there is a reuse of factor matrices as shown in Equations 4, 5, 6, and 7. For example, we see that the Khatri-Rao product of D and C is used in the update of both $\widehat{A}$ in equation 4 and $\widehat{B}$ in equation 5. Furthermore, the computation for N_1 and N_2 in Equations 4 and 6 can be omitted for Equations 5 and 6 by reusing the intermediate results.

As shown in Algorithm 2, for each nonzero $\mathcal{X}(i, j, k)$ at (i, j, k), MTTKRP along mode-1 is performed as the j−th row of matrix $\mathbf{B}$ and the k−row of matrix $\mathbf{C}$ are retrieved based on indices j and k. Their Hadamard product is scaled with the tensor entry at (i, j, k) to update the i−th row of matrix $\mathbf{A}$. When performing the MTTKRP along mode-2, the k-th row of $\mathbf{C}$ and the i-th row of $\mathbf{A}$ are retrieved to update the j−th row of matrix $\mathbf{B}$. Therefore, between mode-1 MTTKRP and mode-2 MTTKRP, the k−th row of the matrix $\mathbf{C}$ can be reused directly (marked by the arrow in Figure 1). The i−th row of A is updated during the mode-1 MTTKRP computation and remains in the same partition without introducing more communication for the computations in the mode-2 MTTKRP. Likewise, data reuse exists between mode-2 and mode-3 MTTKRPs as well as mode-3 and mode-1 MTTKRPs in the next iteration as shown via the arrows in Figure 1.

Implementation workflow: Algorithm 3 describes how the CSTF-QCOO algorithm is implemented. Between consecutive MTTKRP operations, the matrices used to calculate $\widehat{V}$ and $\widehat{Z}$ are all the same except for one matrix. The implementation of CP-ALS benefits from this data reuse in Spark to reduce the number of shuffle operations. CSTF-QCOO utilizes the same storage format as the CSTF-COO algorithm. By using the COO format for the implementation it is able to take full advantage of the sparsity of the tensor in the same way that CSTF-COO does. The first step of the CSTF-QCOO algorithm creates the queues V and Z in lines 1 and 2 of Algorithm 3 and enqueues each factor matrix for the first MTTKRP. The resulting RDD is in the form $\mathcal{X}_Q$, shown in Table 3. Afterwards, STAGE 1 of CSTF-QCOO in Table 2 is applied. The data representation for Stage 1 is similar to the COO format except that a key-value scheme is applied as shown in Table 2 and instead of a single vector a queue is used.

After performing the join transformation in STAGE 1, a map transformation is applied to the RDD to switch to the right key shown in STAGE 2 of CSTF-QCOO in Table 2. During the same map operation, the joined vector is added to the queue and a dequeue operation is performed which drops the oldest vector from the queue. The resulting RDD from STAGE 2 can then be used to perform the first join operation for the next MTTKRP. After STAGE 2, the resulting RDD has its values mapped. The map function reduces the queue by performing an element-wise multiplication with each row vector and finally multiplying it by the tensor value. The entire RDD is then transformed via *ReduceByKey* in order to add all vectors which correspond to the same row of the matrix. This step is shown in STAGE 3 in Table 2. This entire operation corresponds to line 14 in Algorithm 3.

Caching. The queue of the RDD is updated for each MTTKRP, thus, no more than two operations are performed on the same RDD. Because RDDs are immutable many new RDDs are created throughout the CP-ALS algorithm. Each tensor RDD that is used before a join in STAGE 1 of Table 2 is cached to memory. The RDD from the previous MTTKRP iteration is removed from the cache by explicitly asking Spark to unpersist the old RDD. CSTF-QCOO also exploits the reuse of data related to computations of the gram matrices from one MTTKRP update to the next. An entire update of a matrix row consists of an MTTKRP operation followed by the psuedo-inverse of the multiplication of all but one of the gram matrices. Because the matricized modes of the tensor are large and distributed, the gram matrix for each factor is only computed once per CP-ALS iteration. By computing the gram matrix only once per iteration in CSTF-QCOO, the algorithm eliminates the need to perform extra reduce operations.

4.3 Comparing the CSTF Workflow to BIGtensor

In the following we first elaborate the workflow in BIGtensor and then compare it to the CSTF workflow. As shown in Table 2, to perform a mode-1 MTTKRP operation the tensor data is matricized in mode-1 at STAGE-1 of the BIGtensor workflow. It is then joined with the factor matrix $\mathbf{C}$ along mode-3. Both the tensor and the factor matrix $\mathbf{C}$ are shuffled between nodes which leads to data communication. In STAGE-2, the bin function $bin()$ is used to preserve the sparsity of the tensor $\mathcal{X}$, which keeps the nonzero indices of the mode-1 matricized tensor without storing the nonzero values. Then the tensor and factor matrix $\mathbf{B}$ are joined along mode-2. In STAGE-3, BIGtensor combines the results from STAGE-1 and STAGE-2 using the Hadamard product and adds each row to obtain the final result. In this stage, double the number of tensor nonzeros are shuffled. The workflow of BIGtensor from STAGE-1 to STAGE-3 is based on matricization of the tensor, however, CSTF uses key-values to directly operate on nonzeros. Also, CSTF-QCOO utilizes a queue strategy in MTTKRP operations to look for data reuse between MTTKRP operations while BIGtensor optimizes for a single MTTKRP. Finally, at STAGE-2 of BIGtensor, the $bin()$ function is used to preserve the sparsity of the tensor and to execute STAGE-1 and STAGE-2 simultaneously. The $bin()$ function is an expensive operation and requires a full pass over the tensor data.

5 COMPLEXITY ANALYSIS

In this section, we analyze the complexity of the MTTKRP implementation in BIGtensor, CSTF-COO, and CSTF-QCOO demonstrated in Table 4. nnz represents the number of nonzeros in the sparse tensor, R is the rank of the tensor decomposition, and *flops* represents the number of floating point operations. Intermediate data is the size of data stored to complete a single MTTKRP. *Shuffles* represents the number of shuffle operations caused by an MTTKRP operation.

BIGtensor. At STAGE-1 of Table 2, the amount of data communicated is $nnz \times R$ to join tensor $\mathbf{X}_{(1)}$ with matrix $\mathbf{C}$ with one shuffle. At STAGE-2 in Table 2, the communicated data is also $nnz \times R$ to join tensor $bin(\mathbf{X}_{(1)})$ with matrix $\mathbf{B}$ in one shuffle. STAGE-3 combines the intermediate result from STAGE-1 and STAGE-2 with two shuffles. In total, BIGtensor performs four shuffles for one MTTKRP operation. The total amount of communicated data is $4 \times nnz \times R$ (nnz is the number of nonzeros of the tensor $\mathcal{X}$). For intermediate data BIGtensor uses the tensor and one column from the factor matrix B or C for computations at each task. The intermediate data size is $max(J + nnz, K + nnz)$. BIGtensor requires $5 \times nnz \times R$ to perform one MTTKRP, including $3 \times nnz \times R$ for 3 Hadamard products at each STAGE and $2 \times nnz \times R$ for the final multiplication at STAGE-3. Performance analysis of BIGtensor's MTTKRP algorithm is provided in more detail in [11].

CSTF-COO. As shown in Table 4, CSTF-COO requires $3nnz \times R$ flops to perform $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. This includes $nnz \times R$ flops to compute $\mathcal{X}(i,j,k)\mathbf{C}(k,:)$ in STAGE 1 and $nnz \times R$ to compute $\mathcal{X}(i,j,k)\mathbf{C}(k,:) * \mathbf{B}(j,:)$ in STAGE 2. Then finally another $nnz \times R$ to perform the *ReduceByKey* operation in STAGE 3. Every nonzero entry is associated with vector of size R related to a tensor entry. Thus, the intermediate data is $nnz \times R$. The size of the tensor entry itself is not included because these values must be stored with each record. Three shuffle operations are required for a 3rd order tensor. There will be two *joins* and one *ReduceByKey* which is shown in table 2.

A N-order tensor will require up to N shuffles for N MTTKRP operations in CSTF-COO because a *join* must be performed for every dimension of the tensor except for one. The join is followed by a *ReduceByKey* which requires a shuffle operation. For an entire iteration of CP decomposition, N^2 data shuffles with intermediate data of size $nnz \times R$ occur. Thus, the maximum amount of data communicated during shuffles for a single CP iteration is $N^2 \times nnz \times R$. Since the RDD size does not depend on the tensor order, the intermediate data remains the same ($nnz \times R$).

CSTF-QCOO: The required number of flops for CSTF-QCOO is the same as CSTF-COO because both algorithms perform the same number of vector operations. For a 3rd-order tensor up to 2 vectors are required for each tensor entry. Thus, the intermediate data is $2nnz \times R$. However, only a *join* operation is required for each MTTKRP. The total shuffles counting the final *reduceByKey* is 2 while the amount of data required to perform the second shuffle is less than the intermediate data size shown in Table 4 and is only $nnz \times R$. In the CP decomposition algorithm with CSTF-QCOO, an overhead of N shuffles exists before the first MTTKRP operation.

Table 4: Cost comparison of BIGtensor, CSFT-COO, and CSTF-QCOO for 3rd-order mode-1 MTTKRP.

Algorithm	Flops	Intermediate Data	Shuffles
BIGtensor	$5nnz \times R$	$max(J + nnz, K + nnz)$	4
CSTF-COO	$3nnz \times R$	$nnz \times R$	3
CSTF-QCOO	$3nnz \times R$	$2nnz \times R$	2

This overhead occurs because the first N initial vectors must be joined and added to the queue of each record. The queue holds a vector for each dimension of the tensor leading to an increase in the intermediate data. The intermediate data size is $(N-1) \times nnz \times R$ where N is the dimension of the tensor. For a single CP iteration, the maximum communication cost is $N \times (N-1) \times nnz \times R$ for join operations. While this leads to a small decrease in the overall communication costs compared to CSTF-COO, for real world tensors of orders of 3, 4, or 5, CSTF-QCOO reduces communication costs up to 33%, 25%, and 20% respectively.

6 EXPERIMENTS

This section presents the implementation results for the CSTF-COO and CSTF-QCOO algorithms. The performance of the algorithms are compared to the state-of-the-art framework BIGtensor and demonstrated for multidimensional tensors.

6.1 Experimental Setup

The experiments are ran on the Comet cluster provided by the XSEDE [26] project in which each node is equipped with an Intel Xeon E5-2680v3 processor (24 cores per node with clock speed of 2.5GHz), a 128GB RAM, and 320GB of SSD local scratch space. The cluster runs Spark v1.5.2 and Hadoop 2.6.0, and consists of a driver node and up to 32 worker nodes. All the experiments are performed in double precision. We implement the CSTF algorithms on Spark written in Scala.

6.2 Datasets

Datasets used for the experiments vary in density and size. They are standard datasets generated from real applications obtained from FROSTT [22] except for the synthetically generated *synt3d*. Nell1 comes from the Never Ending Language Learning (NELL) project [3]. The nell1 tensor represents *noun-verb-noun* triplets. Delicious4d is a *user-item-tag-date* tensor crawled from tagging systems [7], where date is at the granularity of day. The 3rd-order tensor delicious3d is the same data with dates removed. The *synt3d* is a synthetically generated random 3rd-order tensor. The detailed configurations of these datasets are shown in Table 5.

Table 5: Summary of datasets.

Dataset	Order	Max mode size	nnz	Density
dellicious3d	3	17.3M	140M	$6.5e-12$
nell1	3	25.5M	144M	$9.3e-13$
synt3d	3	$15M$	200M	$5.3e-12$
flickr	4	28M	112M	$1.1e-14$
delicious4d	4	$17.3M$	140M	$4.3e-15$

6.3 Reference Algorithms

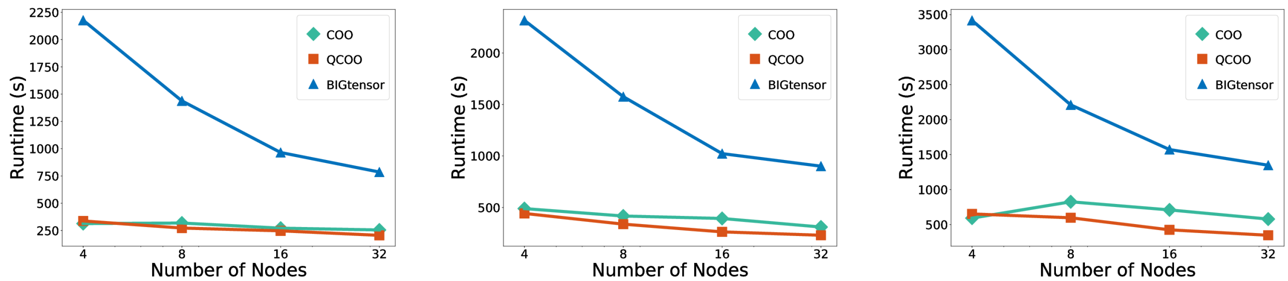
To evaluate the performance of the CSTF algorithms on the 3rd-order CP decomposition algorithm we use BIGtensor [19], a recent large-scale tensor mining library which runs on Hadoop. BIGtensor supports various tensor operations and factorizations including the 3rd-order CP routine. The BIGtensor library uses the implementation for the distributed CP algorithm from GigaTensor [11]. To provide a fair and comprehensive comparison, we execute all the three algorithms, BIGtensor-CP, CSTF-COO, and CSTF-QCOO, for 20 iterations on the same cluster with the *Rank* of tensor factorization fixed to 2. The worker nodes are from 4 to 32 for all three algorithms on the datasets (*nell1*, *delicious3d*, and *synt3d*). We report the average execution time for a CP-ALS iteration of the three algorithms. To evaluate the performance provided by CSTF-QCOO for 4th-order CP decompositions, CSTF-COO is chosen as the baseline for comparison because BIGtensor only supports 3rd-order tensors.

6.4 Performance Results and Analysis

CSTF versus BIGtensor on 3rd-order tensors. Figure 2 compares the performance of CSTF-COO and CSTF-QCOO to BIGtensor. Both the CSTF-COO and CSTF-QCOO algorithms show performance improvements over the BIGtensor library by a large margin. For delicious3d, CSTF-COO achieves 3.0× to 6.9× speedup while CSTF-QCOO achieves 3.8× to 6.5× speedup as shown in Figure 2(a). For nell1, CSTF-COO achieves 2.6× to 4.7× speedup while CSTF-QCOO achieves 3.9× to 5.2× speedup as shown in Figure 2(b). Figure 2(c) shows CSTF-COO achieves 2.2× to 5.8× speedup while CSTF-QCOO achieves 3.7× to 5.2× speedup.

Unfolding or the matricization operations on the input tensor in BIGtensor increases communication overheads on a distributed platform. The *bin*() function used in BIGtensor also increases communication. However, the implementations of CSTF-COO and CSTF-QCOO avoid expensive and unnecessary unfolding operations and explicit generation of the Khatri-Rao product by fully exploiting the sparsity of tensor. Also, the in-memory caching provided by Spark enables faster data access throughout the tensor factorization in CSTF. Through this combination of algorithm improvements the CSTF algorithms are able to achieve better performance compared to the reference implementation—BIGtensor. However, as shown in Figure 2 the scalability of the CSTF algorithms is not better than BIGtensor on 4 to 32 nodes because of the overhead of generating more intermediate data as shown in Table 4.

CSTF-COO versus CSTF-QCOO for 3rd-order and 4th-order tensors. When comparing the performance across different number of nodes, the running time of CSTF-QCOO and CSTF-COO are relatively close for small clusters but the gap widens as the number of nodes and the tensor dimension increases. QCOO performs around 1.1× worse than CSTF-COO for a 4-node cluster on the delicious3d tensor but then improves as more nodes are added. As shown in Figure 2(a), CSTF-QCOO improves the performance from 0.92× to 1.24× on delicious3d. For nell1, the performance improvements range from 1.1× to 1.49× as shown in Figure 2(b). For synt3d, CSTF-QCOO achieves 0.90× to 1.7× speedup over COO as shown



(a) Runtime on delicious3d (b) Runtime on nell1 (c) Runtime on synt3d

Figure 2: Runtime for CSTF-COO, CSTF-QCOO, and BIGtensor CP-ALS on 3rd-order tensors.

in Figure 2(c). The performance gains range from 0.98× to 1.27× for flickr as shown in Figure 3(b). On delicious4d. the speedups range from 1.06× to 1.67× as shown in Figure 3(a). The difference between CSTF-COO and CSTF-QCOO performances are because of the reuse provided by the Queue strategy.

6.5 Communication Cost

Data communication is a major performance bottleneck in large-scale tensor decomposition algorithms. In the following section, we discuss our metrics for measuring this communication and show that CSTF reduces data communication through data-reuse, caching, and an overall reduced number of transformations. We use Spark's [30] built-in metrics collection service to collect the data on shuffle communication costs while running CSTF-COO and CSTF-QCOO. We measure the number remote and local bytes read. Remote bytes are the bytes read from all remote processors across all shuffle phases in Spark. Lower amounts of remote bytes are indicative of reduced network traffic. Local bytes represent the total number of bytes read from a partition without communication during the Spark shuffle phases for all processors. Figure 4 displays the information that was collected for a single CP-ALS iteration.

As shown in Figure 4(a), CSTF-QCOO reads a total of 20.8 GB remote data from other nodes while CSTF-COO reads 31.9 GB remote data for the delicious3d tensor. CSTF-QCOO reduces the shuffle cost by 35% compared to CSTF-COO. Figure 4(a) also shows that CSTF-QCOO reads a total of 23.8 GB data remotely from other nodes while CSTF-COO reads 34.4 GB remote data for the flickr tensor. CSTF-QCOO reduces the shuffle cost by 31% compared to CSTF-COO. These values show that our QCOO algorithm is able to decrease the overall network communication overhead.

The experiments demonstrating the local data read are ran on 8 nodes. As shown in Figure 4(b), CSTF-COO reads 4.68 GB from local processors while CSTF-QCOO consumes 3.0 GB for the delicious3d tensor. CSTF-QCOO reduces the local data read by 36.0% for delicious-3d. Figure 4(b) also shows that CSTF-COO reads 5.13 GB from the file-system while CSTF-QCOO uses 3.34 GB for the flickr tensor. CSTF-QCOO reduces the local communication cost by 35.0% for flickr.

The saved shuffle cost in experiments fits well with the theoretical analysis demonstrated in Section 5. Because of data reuse provided by CSTF-QCOO, less transformations (e.g., map, filter, and join) are performed for a sequence of MTTKRP operations which can be seen in Table 2. The reduction in transformations is reflected in 4(b). If there are less transformations being performed on each RDD then there should also be less overall bytes read from local partitions. By reducing local and global communication, the performance of tensor decomposition algorithms is enhanced significantly.

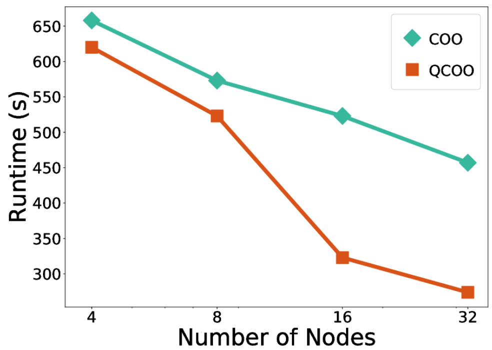
6.6 Mode Behavior

For MTTKRP operations along different modes for the nell1 tensor, as shown in Figure 5(a), CSTF-COO achieves 4.0× to 6.1× speedup over BIGtensor; CSTF-QCOO achieves 4.3× to 6.3× speedup over BIGtensor. For MTTKRP operations along different modes for delicious3d, shown in Figure 5(b), CSTF-COO achieves 5.6× to 6.3× speedup over BIGtensor; CSTF-QCOO achieves 4.3× to 9.5× speedup over BIGtensor.

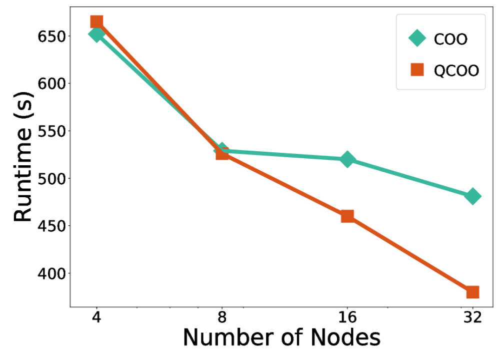
Figure 5 shows that the CSTF algorithm delivers relatively similar performance benefits for all modes because it partitions and parallelizes the nonzeros of the tensor. This is specifically more evident for delicious which is an "oddly" shaped tensor; CSTF-QCOO is able to achieve up to 9.5× speedup. As shown in 5, the runtime for MTTKRP along mode-1 in CSTF-QCOO exceeds CSTF-COO by 30% for nell1 and 35% for delicious3d tensor. This extra overhead comes from initialization of the Queue data structure in the CSTF-QCOO algorithm.

7 CONCLUSION

In this paper, we propose CSTF, which is composed of two highly efficient distributed algorithms for the sparse and higher-order tensor CP decomposition on distributed platforms. CSTF-COO is proposed to decompose large-scale sparse tensors stored in the COO format based on the MapReduce paradigm using the Spark engine. We also present CSTF-QCOO which introduces a queuing strategy to reduce data communication in the CP-ALS algorithm. The experiments show that our proposed CSTF-QCOO algorithm can outperform BIGtensor (the state-of-the-art tensor decomposition tool based on Hadoop) on tested 3rd-order sparse tensor datasets with a 3.9× to 6.5× speedup. For higher-order sparse tensors across

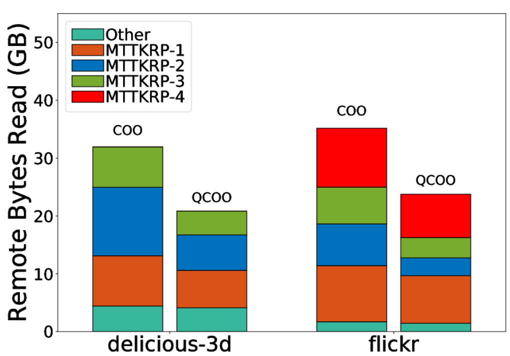


(a) Runtime on delicious4d

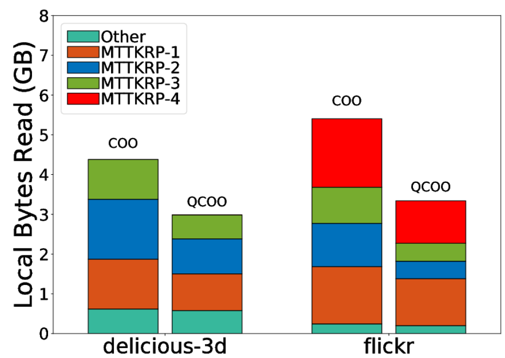


(b) Runtime on flickr

Figure 3: Runtime for CSTF-COO and CSTF-QCOO for CP-ALS on 4th-order tensors.

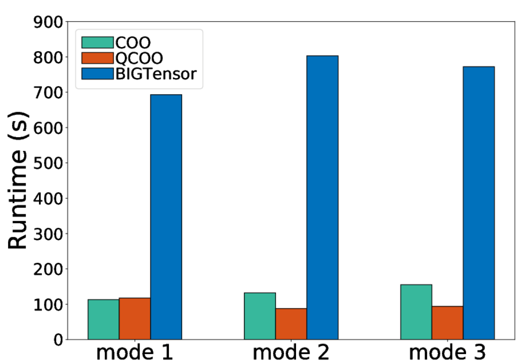


(a) Shuffle data read from remote processors

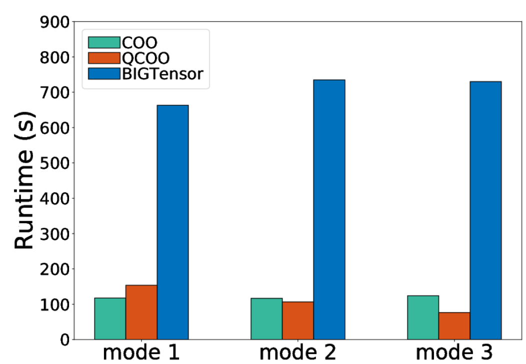


(b) Shuffle data read from local processors

Figure 4: Remote and local data read in CSTF (COO and QCOO) for delicious3d and flickr on an 8 nodes cluster.



(a) MTTKRP of different modes on the nell-1 tensor



(b) MTTKRP of different modes on the delicious3d tensor

Figure 5: MTTKRP runtimes of CSTF-COO, CSTF-QCOO, and BIGtensor across each tensor mode for 3-order CP-ALS on 4 nodes.

all cluster sizes CSTF-QCOO achieves speedups of 0.98× to 1.7× over CSTF-COO.

8 ACKNOWLEDGEMENT

This work is supported by the U.S. National Science Foundation (NSF) Award Number 1657175. This work uses the Extreme Science and Engineering Discovery Environment (XSEDE) [26], which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

[1] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. 2014. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research* 15, 1 (2014), 2773–2832.

[2] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L Willke. 2017. Bridging the gap between hpc and big data frameworks. *Proceedings of the VLDB Endowment* 10, 8 (2017), 901–912.

[3] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka Jr, and Tom M Mitchell. 2010. Toward an Architecture for Never-Ending Language Learning.. In *AAAI*, Vol. 5. 3.

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

[5] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[6] Hancheng Ge, Kai Zhang, Majid Alfifi, Xia Hu, and James Caverlee. [n. d.]. DisTenC: A Distributed Algorithm for Scalable Tensor Completion on Spark. ([n. d.]).

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

[8] Apache Hadoop. 2009. Hadoop. (2009).

[9] Furong Huang, UN Niranjan, Mohammad Umar Hakeem, and Animashree Anandkumar. 2015. Online tensor methods for learning latent variable models. *Journal of Machine Learning Research* 16 (2015), 2797–2835.

[10] Inah Jeon, Evangelos E Papalexakis, U Kang, and Christos Faloutsos. 2015. Haten2: Billion-scale tensor decompositions. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 1047–1058.

[11] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. 2012. 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*. ACM, 316–324.

[12] Oguz Kaya et al. 2016. High Performance Parallel Algorithms for the Tucker Decomposition of Sparse Tensors. In *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 103–112.

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

[14] Oguz Kaya and Bora UcĪğar. 2018. Parallel Candecomp/Parafac Decomposition of Sparse Tensors Using Dimension Trees. *SIAM Journal on Scientific Computing* 40, 1 (2018), C99–C130.

[15] Venera Khoromskaia and Boris N Khoromskij. 2015. Tensor numerical methods in quantum chemistry: from Hartree–Fock to excitation energies. *Physical Chemistry Chemical Physics* 17, 47 (2015), 31491–31509.

[16] Tamara G Kolda and Jimeng Sun. 2008. Scalable tensor decompositions for multi-aspect data mining. In *2008 Eighth IEEE international conference on data mining*. IEEE, 363–372.

[17] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2017. Model-driven sparse CP decomposition for higher-order tensors. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 1048–1057.

[18] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. 2017. A Unified Optimization Approach for Sparse Tensor Operations on GPUs. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 47–57.

[19] Namyong Park, Byungsoo Jeon, Jungwoo Lee, and U Kang. 2016. BIGtensor: Mining Billion-Scale Tensor Made Easy. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2457–2460.

[20] Namyong Park, Sejoon Oh, and U Kang. 2017. Fast and scalable distributed boolean tensor factorization. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 1071–1082.

[21] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor contractions with extended BLAS kernels on CPU and GPU. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*. IEEE, 193–202.

[22] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. (2017). http://frostt.io/

[23] Shaden Smith and George Karypis. 2015. *Dms: Distributed sparse tensor factorization with alternating least squares*. Technical Report. Technical Report 15-007, Department of Computer Science and Engineering, University of Minnesota.

[24] Shaden Smith and George Karypis. 2016. A Medium-Grained Algorithm for Distributed Sparse Tensor Factorization. *30th IEEE International Parallel & Distributed Processing Symposium* (2016).

[25] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 61–70.

[26] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* 16, 5 (Sept.-Oct. 2014), 62–74. https://doi.org/10.1109/MCSE.2014.80

[27] M Alex O Vasilescu. 2011. Multilinear projection for face recognition via canonical decomposition. In *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*. IEEE, 476–483.

[28] M Alex O Vasilescu and Demetri Terzopoulos. 2002. Multilinear analysis of image ensembles: Tensorfaces. In *European Conference on Computer Vision*. Springer, 447–460.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[31] Kaihui Zhang, Yusuke Tanimura, Hidemoto Nakada, and Hirotaka Ogawa. 2017. Understanding and improving disk-based intermediate data caching in Spark. In *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 2508–2517.