

MuLOT: Multi-level optimization of the canonical polyadic tensor decomposition at large-scale

Annabelle Gillet¹, Éric Leclercq¹, and Nadine Cullot¹

LIB EA 7534 Univ. Bourgogne Franche Comté, Dijon, France
annabelle.gillet@depinfo.u-bourgogne.fr, eric.leclercq@u-bourgogne.fr,
nadine.cullot@u-bourgogne.fr

Abstract. Tensors are used in a wide range of analytics tools and as intermediary data structures in machine learning pipelines. Implementations of tensor decompositions at large-scale often select only a specific type of optimization, and neglect the possibility of combining different types of optimizations. Therefore, they do not include all the improvements available, and are less effective than what they could be. We propose an algorithm that uses both dense and sparse data structures and that leverages coarse and fine grained optimizations in addition to incremental computations in order to achieve large scale CP (CANDECOMP/PARAFAC) tensor decomposition. We also provide an implementation in Scala using Spark, MuLOT, that outperforms the baseline of large-scale CP decomposition libraries by several orders of magnitude, and run experiments to show its large-scale capability. We also study a typical use case of CP decomposition on social network data.

Keywords: Tensor decomposition · Data mining · Multi-dimensional analytics.

1 Introduction

Tensors are powerful mathematical objects, which bring capabilities to model multi-dimensional data [8]. They are used in multiple analytics frameworks, such as Tensorflow [1], PyTorch [23], Theano [3], TensorLy [18], where their ability to represent various models is a great advantage. Furthermore, associated with powerful decompositions, they can be used to discover the hidden value of Big Data. Tensor decompositions are used for various purposes such as dimensionality reduction, noise elimination, identification of latent factors, pattern discovery, ranking, recommendation and data completion. They are applied in a wide range of applications, including genomics [14], analysis of health records [29], graph mining [28] and complex networks analysis [19,4]. Papalexakis et al. in [21] review major usages of tensor decompositions in data mining applications.

Most of tensor libraries that include decompositions work with tensors of limited size, and do not consider the large-scale challenge. However, as tensors model multi-dimensional data, their global size varies exponentially depending

on the number and size of their dimensions, making them sensitive to large-scale issues. Some intermediate structures needed in the algorithms result in data explosion, such as the Khatri-Rao product in the canonical polyadic decomposition [15]. Thus, analyzing Big Data with tensors requires optimization techniques and suitable implementations, able to scale up. These optimizations are directed toward different computational aspects, such as the memory consumption, the execution time or the scaling capabilities, and can follow different principles, such as coarse grained optimizations, fine grained optimizations or incremental computations.

In this article we focus on the canonical polyadic decomposition (also known as CANDECOMP/PARAFAC or CP decomposition) that allows to factorize a tensor into smaller and more usable sets of vectors [17], and which is largely adopted in exploratory analyzes. Our contribution is twofold: 1) we propose an optimized algorithm to achieve large scale CP decomposition, that uses dense or sparse data structures depending on what suits best each step, and that leverages incremental computation, coarse and fine grained optimizations to improve every computation in the algorithm ; and 2) we provide an implementation in Scala using Spark that outperforms the state of the art of large-scale tensor CP decomposition libraries. The implementation is open source and available on Github¹, along with experimental evaluations to validate its efficiency especially at large scale.

The rest of the article is organized as follows: section 2 presents an overview of tensors including the CP decomposition, section 3 introduces a state of the art of tensor manipulation libraries, section 4 describes our scalable and optimized algorithm, section 5 details the experiments we ran to compare our algorithm to other large-scale CP decomposition libraries, section 6 presents a study on real data performed with our algorithm and finally section 7 concludes.

2 Overview of tensors and CP decomposition

Tensors are general abstract mathematical objects which can be considered according to various points of view such as a multi-linear application, or as the generalization of matrices to multiple dimensions. We will use the definition of a tensor as an element of the set of the functions from the product of N sets $I_j, j = 1, \dots, N$ to $\mathbb{R} : \mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, where N is the number of dimensions of the tensor or its order or its mode. Table 1 summarizes the notations used in this article.

Tensor operations, by analogy with operations on matrices and vectors, are multiplications, transpositions, unfolding or matricizations and factorizations (also named decompositions) [17,8]. The reader can consult these references for an overview of the major operators. We only highlight the most significant operators on tensors which are used in our algorithm. The mode- n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ noted $\mathcal{X}_{(n)}$ produces a matrix $\mathbf{M} \in \mathbb{R}^{I_n \times \prod_{j \neq n} I_j}$. The

¹ <https://github.com/AnnabelleGillet/MuLOT>

Hadamard product of two matrices having the same size (i.e., $I \times J$) noted $\mathbf{A} \odot \mathbf{B}$ is the elementwise matrix product. The Kronecker product between a matrix $\mathbf{A} \in \mathbb{R}^{I \times J}$ and a matrix $\mathbf{B} \in \mathbb{R}^{K \times L}$ noted $\mathbf{A} \otimes \mathbf{B}$ gives a matrix $\mathbf{C} \in \mathbb{R}^{(IK) \times (JL)}$, where each element of \mathbf{A} is multiplied by \mathbf{B} . The Khatri-Rao product of two matrices having the same number of columns noted $\mathbf{A} \odot \mathbf{B}$ is a columnwise Kronecker product.

Symbol	Definition	Symbol	Definition
\mathcal{X}	A tensor	\circ	Outer product
$\mathcal{X}_{(n)}$	Matricization of a tensor \mathcal{X} on mode- n	\otimes	Kronecker product
a	A scalar	\oplus	Hadamard product
\mathbf{v}	A column vector	\oslash	Hadamard division
\mathbf{M}	A matrix	\odot	Khatri-Rao product
		\dagger	Pseudo inverse

Table 1: Symbols and operators used

The canonical polyadic decomposition allows to factorize a tensor into smaller and more exploitable sets of vectors [13,25]. Given a N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ and a rank $R \in \mathbb{N}$, the CP decomposition factorizes the tensor \mathcal{X} into N column-normalized factor matrices $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$ for $i = 1, \dots, N$ with their scaling factors $\lambda \in \mathbb{R}^R$ as follows:

$$\mathcal{X} \simeq \llbracket \lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)} \rrbracket = \sum_{r=1}^R \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ \dots \circ a_r^{(N)}$$

where $a_r^{(i)}$ are columns of $\mathbf{A}^{(i)}$.

Algorithm 1 CP-ALS

Require: Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ and target rank R

- 1: Initialize $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
 - 2: **repeat**
 - 3: **for** $n = 1, \dots, N$ **do**
 - 4: $\mathbf{V} \leftarrow \mathbf{A}^{(1)T} \mathbf{A}^{(1)} \oplus \dots \oplus \mathbf{A}^{(n-1)T} \mathbf{A}^{(n-1)} \oplus \mathbf{A}^{(n+1)T} \mathbf{A}^{(n+1)} \oplus \dots \oplus \mathbf{A}^{(N)T} \mathbf{A}^{(N)}$
 - 5: $\mathbf{A}^{(n)} \leftarrow \mathcal{X}_{(n)} (\mathbf{A}^{(N)} \odot \dots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \dots \odot \mathbf{A}^{(1)}) \mathbf{V}^\dagger$
 - 6: normalize columns of $\mathbf{A}^{(n)}$
 - 7: $\lambda \leftarrow$ norms of $\mathbf{A}^{(n)}$
 - 8: **end for**
 - 9: **until** $<$ convergence $>$
-

Several algorithms have been proposed to compute the CP decomposition [26], we focus on the alternating least squares (ALS) one, described above in algorithm 1. The Matricized Tensor Times Khatri-Rao Product (MTTKRP, line 5 of the algorithm 1) is often the target of optimizations, because it involves the tensor matricized of size $\mathbb{R}^{I_n \times J}$, with $J = \prod_{j \neq n} I_j$, as well as the result of the Khatri-Rao product of size $\mathbb{R}^{J \times R}$. It is thus computationally demanding and uses a lot of memory to store the dense temporary matrix resulting of the Khatri-Rao product [24].

3 State of the art

Several tensor libraries have been proposed. They can be classified according to their capability of handling large tensors or not.

rTensor (<http://jamesyili.github.io/rTensor/>) provides users with standard operators to manipulate tensors in R language including tensor decompositions, but does not support sparse tensors. Tensor Algebra Compiler (TACO) provides optimized tensor operators in C++ [16]. High-Performance Tensor Transpose [27] is a C++ library only for tensor transpositions, thus it lacks lots of useful operators. Tensor libraries for MATLAB, such as TensorToolbox (<https://www.tensortoolbox.org/>) or MATLAB Tensor Tools (MTT, <https://github.com/andrewssobral/mtt>), usually focus on operators including tensor decompositions with optimization on CPU or GPU. TensorLy [18], written in Python, allows to switch between tensor libraries back-ends such as TensorFlow or PyTorch. All of these libraries do not take into account large tensors, which cannot fit in memory.

On the other hand, some implementations focus on performing decompositions on large-scale tensors in a distributed setting. HaTen2 [15] is a Hadoop implementation of the CP and Tucker decompositions using the map-reduce paradigm. It was later improved with BigTensor [22]. SamBaTen [12] proposes an incremental CP decomposition for evolving tensors. The authors developed a Matlab and a Spark implementations. Gudibanda et al. in [11] developed a CP decomposition for coupled tensors using Spark (i.e., different tensors having a dimension in common). ParCube [20] is a parallel Matlab implementation of the CP decomposition. CSTF [5] is based on Spark and proposes a distributed CP decomposition.

As a conclusion, the study of the state of the art shows some limitations of the proposed solutions. A majority of frameworks are limited to 3 or 4 dimensions which is a drawback for analyzing large-scale, real and complex data. They focus on a specific type of optimization, and use only sparse structures to satisfy the sparsity of large tensors. This is a bottleneck to performance, as they do not consider all the characteristics of the algorithm (i.e., the factor matrices are dense). Furthermore, they are not really data centric, as they need an input only with integer indexes, for dimensions and for values of dimensions. Thus it reduces greatly the user-friendliness as the mapping between real values (e.g., user name or timestamp) and indexes has to be supported by the user. The Hadoop implementations need a particular input format, thus necessitate data transformations to execute the decomposition and to interpret the results, leading to laborious prerequisites and increasing the risk of mistakes when working with the results. Moreover, not all of the implementations are open-source, some only give the binary code.

4 Distributed, scalable and optimized ALS for Apache Spark

Optimizing the CP ALS tensor decomposition induces several technical challenges, that gain importance proportionally to the size of the data. Thus, to compute the decomposition at large scale, several issues have to be resolved.

First, the **data explosion of the MTTKRP** is a serious computational bottleneck (line 5 of algorithm 1), that can overflow memory, and prevent to work with large tensors, even if they are extremely sparse. Indeed, the matrix produced by the Khatri-Rao has $J \times R$ non-zero elements, with $J = \prod_{j \neq n} I_j$, for an input tensor of size $\mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$. We propose to reorder carefully this operation, in order to avoid the data explosion and to improve significantly the execution time (see algorithm 3).

The main operations in the ALS algorithm, i.e., the update of the factor matrices, **are not themselves parallelizable** (lines 4 and 5 of algorithm 1). In such a situation, it is profitable to think of other methods to take advantage of parallelism, that could be applied on fine grained operations. For example, leveraging parallelism for matrices multiplication is an optimization that can be applied in many situations. This also eases the reuse of such optimizations, without expecting specific characteristics from the algorithm (see section 4.2).

The **nature of data structures used in the CP decomposition are mixed**: tensors are often sparse, while factor matrices are dense. Their needs to be efficiently implemented diverge, so rather than sticking globally to sparse data structures to match the sparsity of tensors, each structure should take advantage of their particularities to improve the whole execution (see section 4.1). To the best of our knowledge, this strategy has not been explored by others.

The **stopping criterion** can also be a bottleneck. In distributed implementations of the CP ALS, the main solutions used to stop the algorithm are to wait for a fixed number of iterations, or to compute the Frobenius norm on the difference between the input tensor and the tensor reconstructed from the factor matrices. The first solution severely lacks in precision, and the second is computationally demanding as it involves outer products between all the factor matrices. However, an other option is available to check the convergence, and consists in measuring the similarity of the factor matrices between two iterations, evoked in [8,17]. It is a very efficient solution at large-scale, as it merges precision and light computations (see section 4.3).

Finally, the implementation should facilitate the **data loading**, and avoid data transformations only needed to fit the expected input of the algorithm. It should also produce easily interpretable results, and minimize the risk of errors induced by laborious data manipulations (see section 4.4). The study of the state of the art of tensor libraries shows that tensors are often used as multi dimensional arrays, that are manipulated through their indexes, even if they represent real world data. The mapping between indexes and values is delegated to the user, although being an error-prone step. As it is a common task, it should be handled by the library.

To tackle these challenges, we leverage three optimization principles to develop an efficient decomposition: coarse grained optimization, fine grained optimization, and incremental computation. The coarse grained one relies on specific data structures and capabilities of Spark to efficiently distribute operations. The incremental computation is used to avoid to compute the whole Hadamard product at each iteration. The fine grained optimization is applied

on the MTTKRP to reduce the storage of large amount of data and costly computations. For this, we have extended Spark’s matrices with the operations needed for the CP decomposition. In addition, we choose to use an adapted converging criteria, efficient at large-scale. For the implementation of the algorithm, we take a data centric point of view to facilitate the loading of data and the interpretation of the results. Our CP decomposition implementation is thus able to process tensors with billions of elements (i.e., non zero entries) on a mid-range workstation, and small and medium size tensors can be processed in a short time on a low-end personal computer.

4.1 Distributed and scalable matrix data structures

A simple but **efficient sparse matrix storage structure** is COO (COOrdinate storage) [2,10]. The `CoordinateMatrix`, available in the `mllib` package of Spark [6], is one of those structures, that stores only the coordinates and the value of each existing element in a RDD (Resilient Distributed Datasets). **It is well suited to process sparse matrices.**

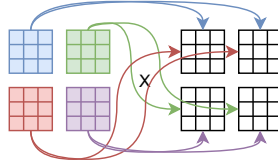


Fig. 1: Blocks mapping for a multiplication between two `BlockMatrix`

Another useful structure is the `BlockMatrix`. It is composed of multiple blocks containing each a fragment of the global matrix. Operations can be parallelized by executing it on each sub-matrix. For binary operations such as multiplication, only blocks from each `BlockMatrix` that will be associated are sent to each other, and the result is then aggregated if needed (see figure 1). **It is thus an efficient structure for dense matrices**, and allows distributed computations to process all blocks.

Unfortunately, only some basic operations are available for `BlockMatrix`, such as multiplication or addition. The more complex ones, such as the Hadamard and Khatri-Rao products, are missing. We have extended Spark `BlockMatrix` with more advanced operations, that keep the coarse grained optimization logic of the multiplication. We also added new operations, that involve `BlockMatrix` and `CoordinateMatrix` to take advantage of the both structures for our optimized MTTKRP (see below).

4.2 Mixing three principles of optimization

Tensors have generally a high level of sparsity. In the CP decomposition, they only appear under their matricized form, thus they are naturally manipulated as `CoordinateMatrix` in our implementation. On the other hand, the factor matrices A of the CP decomposition are dense, because they hold information

for each value of each dimension. They greatly benefit from the capabilities of the extended `BlockMatrix` we developed. By using the most suitable structure for each part of the algorithm, we leverage specific optimizations that can speed up the whole algorithm.

Algorithm 2 CP-ALS adapted to Spark

Require: Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ and target rank R

- 1: Initialize $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
- 2: $\mathbf{V} \leftarrow \mathbf{A}^{(1)T} \mathbf{A}^{(1)} \oplus \dots \oplus \mathbf{A}^{(N)T} \mathbf{A}^{(N)}$
- 3: **repeat**
- 4: **for** $n = 1, \dots, N$ **do**
- 5: $\mathbf{V} \leftarrow \mathbf{V} \oslash \mathbf{A}^{(n)T} \mathbf{A}^{(n)}$
- 6: $\mathbf{A}^{(n)} \leftarrow \text{MTTKRP}(\mathcal{X}_{(n)}, (\mathbf{A}^{(N)}, \dots, \mathbf{A}^{(n+1)}, \mathbf{A}^{(n-1)}, \dots, \mathbf{A}^{(1)})) \mathbf{V}^\dagger$
- 7: $\mathbf{V} \leftarrow \mathbf{V} \otimes \mathbf{A}^{(n)T} \mathbf{A}^{(n)}$
- 8: normalize columns of $\mathbf{A}^{(n)}$
- 9: $\lambda \leftarrow$ norms of $\mathbf{A}^{(n)}$
- 10: **end for**
- 11: **until** $< \text{convergence} >$

Besides to using and improving Spark's matrices according to the specificities of data, we also have introduced fine grained optimization and incremental computing into the algorithm to avoid costly operations in terms of memory and execution time. Those improvements are synthesized in algorithm 2 and explained below.

First, to avoid computing \mathbf{V} completely at each iteration for each dimension, we propose to do it incrementally. Before iterating, we calculate the Hadamard product for all \mathbf{A} (line 2 of the algorithm 2). At the beginning of the iteration, $\mathbf{A}^{(n)T} \mathbf{A}^{(n)}$ is element-wise divided from \mathbf{V} , giving the expected result at this step (line 5 of the algorithm 2). At the end of the iteration, the Hadamard product between the new $\mathbf{A}^{(n)T} \mathbf{A}^{(n)}$ and \mathbf{V} prepares \mathbf{V} for the next iteration (line 7 of the algorithm 2).

The MTTKRP part (line 6 of the algorithm 2) is sensitive to improvement, as stated in section 2. Indeed, by focusing on the result rather than on the operation, it can be easily reordered. For example, if we multiply a 3-order tensor matricized on dimension 1 with the result of $\mathbf{A}^{(3)} \oslash \mathbf{A}^{(2)}$, we can notice that in the result, the indexes of the dimensions in the tensor \mathcal{X} correspond directly to those in the matrices $\mathbf{A}^{(3)}$ and $\mathbf{A}^{(2)}$. This behaviour is represented below — with notation shortcut $B_i = \mathbf{A}^{(2)}(i, 1)$ and $C_i = \mathbf{A}^{(3)}(i, 1)$ — in an example simplified with only one rank:

$$\begin{bmatrix} a_1 b_1 c_1 & a_1 b_2 c_1 & a_1 b_1 c_2 & a_1 b_2 c_2 \\ a_2 b_1 c_1 & a_2 b_2 c_1 & a_2 b_1 c_2 & a_2 b_2 c_2 \end{bmatrix} \times \begin{bmatrix} B_1 C_1 \\ B_2 C_1 \\ B_1 C_2 \\ B_2 C_2 \end{bmatrix} =$$

$$\begin{bmatrix} a_1 b_1 c_1 \cdot B_1 C_1 + a_1 b_2 c_1 \cdot B_2 C_1 + a_1 b_1 c_2 \cdot B_1 C_2 + a_1 b_2 c_2 \cdot B_2 C_2 \\ a_2 b_1 c_1 \cdot B_1 C_1 + a_2 b_2 c_1 \cdot B_2 C_1 + a_2 b_1 c_2 \cdot B_1 C_2 + a_2 b_2 c_2 \cdot B_2 C_2 \end{bmatrix}$$

Thus, rather than computing the full Khatri-Rao product and performing the multiplication with the matricized tensor, we apply a fine grained optimization that takes advantage of the mapping of indexes, and that anticipates the construction of the final matrix. For each entry of the `CoordinateMatrix` of the matricized tensor (i.e., all non-zero values), we find in each factor matrix \mathbf{A} which element will be used, and compute elements of the final matrix (algorithm 3).

Algorithm 3 Detail of the MTTKRP

Require: The index of the factor matrix n , the matricized tensor $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times J}$ with $J = \prod_{j \neq n} I_j$ and $\mathbf{A}^{(1)}, \dots, \mathbf{A}^{(n-1)}, \mathbf{A}^{(n+1)}, \dots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$

- 1: Initialize $\mathbf{A}^{(n)}$ at 0, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
- 2: **for each** (x, y, v) in $\mathcal{X}_{(n)}$ with x, y the coordinates and v the value **do**
- 3: **for** $r = 1, \dots, R$ **do**
- 4: $value \leftarrow v$
- 5: **for each** $\mathbf{A}^{(i)}$ with $i \neq n$ **do**
- 6: $c \leftarrow \text{extract } \mathbf{A}^{(i)} \text{ coordinate from } y$
- 7: $value \leftarrow value \times \mathbf{A}^{(i)}(c, r)$
- 8: **end for**
- 9: $\mathbf{A}^{(n)}(x, r) \leftarrow \mathbf{A}^{(n)}(x, r) + value$
- 10: **end for**
- 11: **end for**

4.3 Stopping criterion

To evaluate the convergence of the algorithm and when it can be stopped, a majority of CP decomposition implementations uses the Frobenius norm on the difference between the original tensor and the reconstructed tensor from the factor matrices. However, at large-scale the reconstruction of the tensor from the factor matrices is an expensive computation, even more than the naive MTTKRP. Waiting for a predetermined number of iterations is not very effective to avoid unnecessary iterations. Thus, other stopping criteria such as the evaluation of the difference between the factor matrices with those of the previous iteration [8,17] are much more interesting, as they work on smaller chunks of data. To this end, we use the Factor Match Score (FMS) [7] to measure the difference between factor matrices of the current iteration ($\llbracket \lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(N)} \rrbracket$) and those of the previous iteration ($\llbracket \hat{\lambda}, \hat{\mathbf{A}}^{(1)}, \hat{\mathbf{A}}^{(2)}, \dots, \hat{\mathbf{A}}^{(N)} \rrbracket$). The FMS is defined as follows:

$$FMS = \frac{1}{R} \sum_{r=1}^R \left(1 - \frac{\xi - \hat{\xi}}{\max(\xi, \hat{\xi})} \right) \prod_{n=1}^N \frac{a_r^{(n)T} \hat{a}_r^{(n)}}{\|a_r^{(n)}\| \cdot \|\hat{a}_r^{(n)}\|}$$

where $\xi = \lambda_r \prod_{n=1}^N \|a_r^{(n)}\|$ and $\hat{\xi} = \hat{\lambda}_r \prod_{n=1}^N \|\hat{a}_r^{(n)}\|$

4.4 Data centric implementation

Our implementation of the CP decomposition, in addition to being able to run with any number of dimensions, is data centric: it takes a Spark `DataFrame`

as input to execute the CP directly on real data. Thus, it benefits from Spark capabilities to retrieve data directly from various datasources.

A specific column of the DataFrame contains the values of the tensor and all the other columns contain the values for each dimension. The CP operators returns a map associating the original names of the dimensions to a new DataFrame with three columns for each dimension: the dimension's values, the rank, and the value computed by the CP decomposition. By using DataFrame as input, we allow the use of any type as dimensions' values. For example, users could create a DataFrame with four columns: username, hashtag, time and value, with username and hashtag being of type String in order to easily interpret the decomposition result. This avoids having to handle an intermediate data structure containing the mapping between indexes and real values, and thus reduces the risk of mistakes when transforming data.

5 Experiments

To validate our algorithm, we have run experiments on tensors produced by varying the size of dimensions and the sparsity, on a Dell PowerEdge R740 server (Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 20 cores, 256GB RAM). We compare our execution time to those of the baseline of distributed CP tensor decomposition libraries available: HaTen2 [15], BigTensor [22], SamBaTen [12] and CSTF [5]. Hadoop 2.6.0 was used to execute HaTen and BigTensor. We also study the scalability of MuLOT by varying the number of cores used by Spark.

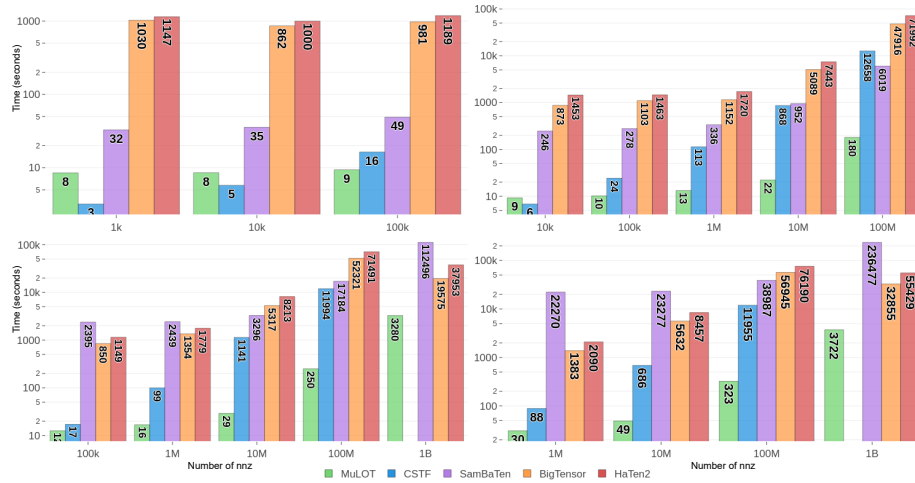


Fig. 2: Execution time for tensors with 3 dimensions of size 100 (top-left), 1 000 (top-right), 10 000 (bottom-left) and 100 000 (bottom-right). CSTF produces an Out Of Memory exception for tensors with 1B elements

Tensors were created randomly with 3 dimensions of the same size, from 100 to 100k. The sparsity ranges from 10^{-1} to 10^{-9} , and tensors were created only if the number of non-zero elements is superior to $3 \times \text{size}$ and inferior or equal to 1B (with dimensions of size 100 and 1 000, tensors can only have respectively 10^6 and 10^9 non-zero elements at most, with a sparsity up to 10^{-1} they cannot reach 1B elements, but respectively 10^5 and 10^8 non-zero elements). We have run the CP decomposition for 5 iterations, and have measured the execution time. Results are shown in figure 2. The source code of the experiments and the tool used to create tensors are available at <https://github.com/AnnabelleGillet/MuLOT/tree/main/experiments>.

Our implementation clearly outperforms the state of the art, with speed-up reaching several order of magnitude. CSTF keeps up concerning the execution time of small tensors, but is no match for large tensors, and cannot compute the decomposition for tensors with 1B elements. Execution times of MuLOT are nearly linear following the number of non-zero elements. The optimization techniques applied show efficient results even for very large tensors of billion elements, with a maximum execution time for a 3-order tensor with dimensions of size 100k of 62 minutes, while the closest, BigTensor, takes 547 minutes. It also does not induce a high overhead for small tensors, as the decomposition on those with dimensions of size 100 takes less than 10 seconds.

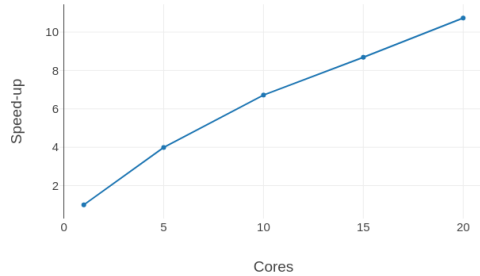


Fig. 3: Near-linear scalability of our algorithm

We also studied the scalability of our algorithm (figure 3). We measured the speed-up depending on the number of cores used by Spark. Our algorithm shows a sub-linear scalability, but without a big gap. The scalability is an important property for large-scale computations.

6 Real data study

We have experimented our decomposition in the context of Cocktail², an interdisciplinary research project aiming to study trends and weak signals in discourses about food and health on Twitter. In order to test our decomposition on real data, we focus on french tweets revolving around COVID-19 vaccines, harvested with Hydre, a high performance plateforme to collect, store and analyze tweets [9]. The corpus contains 9 millions of tweets from the period of November 18th 2020 to January 26th 2021.

We would like to study the communication patterns in our corpus. To this end, we have built a 3-order tensor, with a dimension representing the users, another the hashtags and the last one the time (with a week granularity). For each user, we kept only the hashtags that he had used at least five times on the whole period. The size of the tensor is $10340 \times 5469 \times 80$, with 135k non-zero elements. We have run the CP decomposition with 20 ranks.

This decomposition allowed us to discover meaningful insights on our data, some of the most interesting ranks have been represented in figure 4 (the accounts have been anonymised). We have a background discourse talking about lockdown and curfew, with some hashtags related to media and the French Prime Minister. It corresponds to the major actuality subjects being discussed around the vaccines.

It also reveals more subject-oriented patterns, with one being anti-Blanquer (the French Minister of Education), where accounts that seem to belong to highschool teachers use strong hashtags against the Minister (the translation of some of the hashtags are: Blanquer lies, Blanquer quits, ghost protocol, the protocol refers to the health protocol in french schools). We can identify in this pattern a strong movement of disagreement, with teachers and parents worrying about the efficiency and the applicability of the measures took to allow schools to stay open during the pandemic.

Another pattern appears to be anti-government, with some signs of conspiracy. They use hashtags such as health dictatorship, great reset, deep state corruption, wake up, we are the people, disobey, etc. Indeed, the pandemic inspired a rise in doubt and opposition to some decisions of the government to handle the situation, that sometimes lead to conspiracy theories.

It is interesting to see that the CP decomposition is able to highlight some isolated patterns. With this capability, we identify a bot in our corpus, that quotes tweets that it judges as conspiracy-oriented, and gives them a score to measure the degree of conspiracy.

The CP decomposition is well-suited to real case studies. It is a great tool for our project, as it shows promising capabilities to detect patterns in data along tensor dimensions, with a good execution time. The results given by the decomposition can be easily interpreted and visualized: they can be shared with researchers in social science to specify the meaning of each rank, and thus giving valuable insights on the corpus.

² <https://projet-cocktail.fr/>

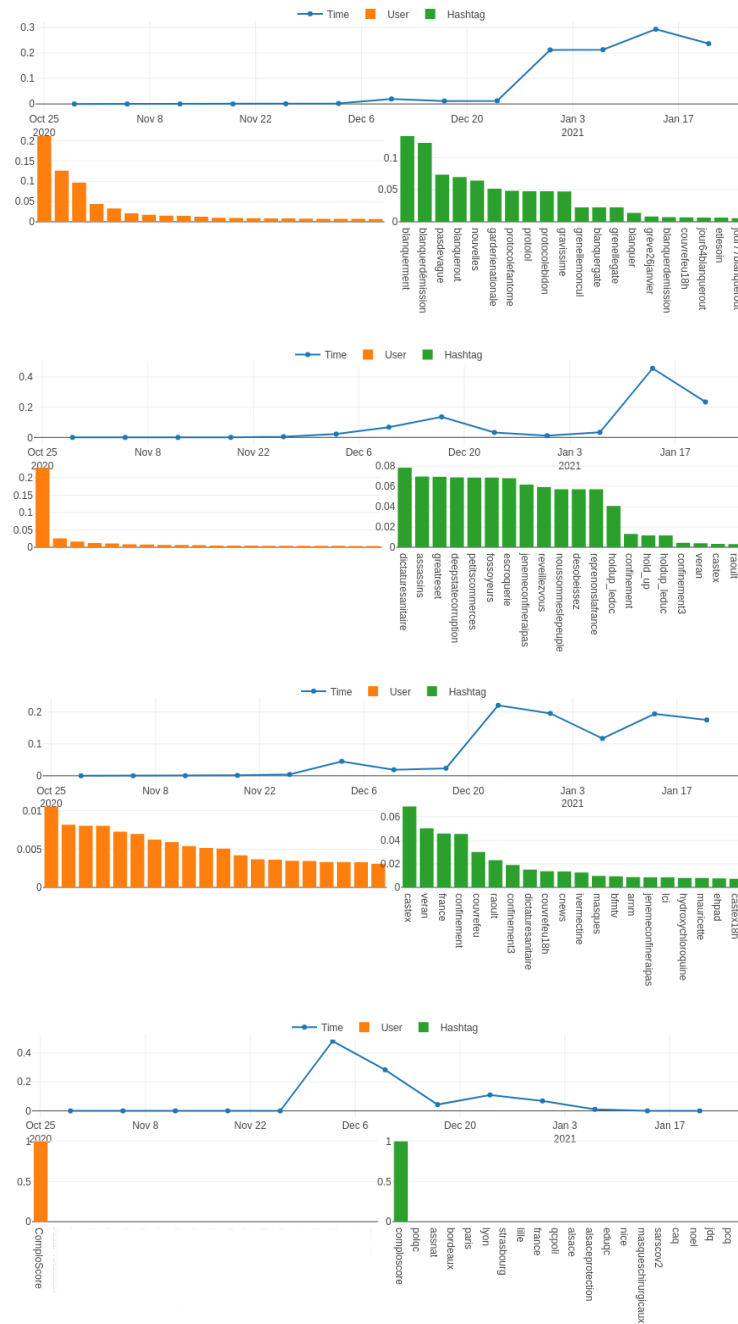


Fig. 4: Communication patterns in the vaccine corpus (from top to bottom): the anti-Blanquer, the conspirators/anti-government, the background speech, and a bot to measure conspiracy score of tweets

7 Conclusion

We have proposed an optimized algorithm for the CP decomposition at large-scale. We have validated this algorithm with a Spark implementation, and shows that it outperforms the state of the art by several orders of magnitude. We also put data at the core of tensors, by taking care of the mapping between indexes and values without involving the user, thus allowing to focus on data and analyses. Through experiments, we proved that our library is well-suited for small to large-scale tensors, and that it can be used to run the CP decomposition on low-end computers for small and medium tensors, hence making possible a wide range of use cases.

We plan to continue our work on tensor decompositions by 1) exploring their use in social networks analyzes ; 2) developing other tensor decompositions such as Tucker, HOSVD or DEDICOM ; and 3) studying the impact of the choice of the norm for the scaling of the factor matrices.

Acknowledgments This work is supported by ISITE-BFC (ANR-15-IDEX-0003) co-ordinated by G. Brachotte, CIMEOS Laboratory (EA 4177), University of Burgundy.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation. pp. 265–283 (2016)
2. Ahmed, N., Mateev, N., Pingali, K., Stodghill, P.: A framework for sparse matrix code synthesis from high-level specifications. In: SC’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. pp. 58–58. IEEE (2000)
3. Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., et al.: Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688 (2016)
4. Araujo, M., Papadimitriou, S., Günnemann, S., Faloutsos, C., Basu, P., Swami, A., Papalexakis, E.E., Koutra, D.: Com2: fast automatic discovery of temporal (‘comet’) communities. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 271–283. Springer (2014)
5. Blanco, Z., Liu, B., Dehnavi, M.M.: Cstf: Large-scale sparse tensor factorizations on distributed platforms. In: Proceedings of the 47th International Conference on Parallel Processing. pp. 1–10 (2018)
6. Bosagh Zadeh, R., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A., Zaharia, M.: Matrix computations and optimization in apache spark. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 31–38 (2016)
7. Chi, E.C., Kolda, T.G.: On tensors, sparsity, and nonnegative factorizations. SIAM Journal on Matrix Analysis and Applications **33**(4), 1272–1299 (2012)
8. Cichocki, A., Zdunek, R., Phan, A.H., Amari, S.: Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation. John Wiley & Sons (2009)

9. Gillet, A., Leclercq, É., Cullot, N.: Lambda+, the renewal of the Lambda Architecture: Category Theory to the rescue (to be published). In: Conference on Advanced Information Systems Engineering (CAiSE). pp. 1–15 (2021)
10. Goharian, N., Jain, A., Sun, Q.: Comparative analysis of sparse matrix algorithms for information retrieval. *computer* **2**, 0–4 (2003)
11. Gudibanda, A., Henretty, T., Baskaran, M., Ezick, J., Lethin, R.: All-at-once decomposition of coupled billion-scale tensors in apache spark. In: High Performance extreme Computing Conference. pp. 1–8. IEEE (2018)
12. Gujral, E., Pasricha, R., Papalexakis, E.E.: Sambaten: Sampling-based batch incremental tensor decomposition. In: International Conference on Data Mining. pp. 387–395. SIAM (2018)
13. Harshman, R.A., et al.: Foundations of the PARAFAC procedure: Models and conditions for an “explanatory” multimodal factor analysis (1970)
14. Hore, V., Viñuela, A., Buil, A., Knight, J., McCarthy, M.I., Small, K., Marchini, J.: Tensor decomposition for multiple-tissue gene expression experiments. *Nature genetics* **48**(9), 1094–1100 (2016)
15. Jeon, I., Papalexakis, E.E., Kang, U., Faloutsos, C.: Haten2: Billion-scale tensor decompositions. In: International Conference on Data Engineering. pp. 1047–1058. IEEE (2015)
16. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. *OOPSLA* pp. 1–29 (2017)
17. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. *SIAM review* **51**(3), 455–500 (2009)
18. Kossaifi, J., Panagakis, Y., Anandkumar, A., Pantic, M.: Tensorly: Tensor learning in python. *The Journal of Machine Learning Research* **20**(1), 925–930 (2019)
19. Papalexakis, E.E., Akoglu, L., Ience, D.: Do more views of a graph help? community detection and clustering in multi-graphs. In: International Conference on Information Fusion. pp. 899–905. IEEE (2013)
20. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: ParCube: Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition. *ACM Transactions on Knowledge Discovery from Data (TKDD)* **10**(1), 1–25 (2015)
21. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *Transactions on Intelligent Systems and Technology (TIST)* **8**(2), 16 (2016)
22. Park, N., Jeon, B., Lee, J., Kang, U.: Bigtensor: Mining billion-scale tensor made easy. In: ACM International on Conference on Information and Knowledge Management. pp. 2457–2460 (2016)
23. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: PyTorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems. pp. 8024–8035 (2019)
24. Phan, A.H., Tichavský, P., Cichocki, A.: Fast alternating ls algorithms for high order candecomp/parafac tensor factorizations. *Transactions on Signal Processing* **61**(19), 4834–4846 (2013)
25. Rabanser, S., Shchur, O., Günnemann, S.: Introduction to tensor decompositions and their applications in machine learning. *arXiv preprint arXiv:1711.10781* (2017)
26. Sidiropoulos, N.D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E.E., Faloutsos, C.: Tensor decomposition for signal processing and machine learning. *Transactions on Signal Processing* **65**(13), 3551–3582 (2017)

27. Springer, P., Su, T., Bientinesi, P.: Hptt: a high-performance tensor transposition c++ library. In: ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 56–62 (2017)
28. Sun, J., Tao, D., Faloutsos, C.: Beyond streams and graphs: dynamic tensor analysis. In: ACM SIGKDD International Conference on Knowledge Discovery and Data mining. pp. 374–383. ACM (2006)
29. Yang, K., Li, X., Liu, H., Mei, J., Xie, G., Zhao, J., Xie, B., Wang, F.: Tagited: Predictive task guided tensor decomposition for representation learning from electronic health records. In: Proc. of the Thirty-First AAAI Conference on Artificial Intelligence (2017)