Alternating Optimization for Tensor Factorization with Orthogonality Constraints: Algorithm and Parallel Implementation

Paris A. Karakasis  
School of Electrical and Computer Engineering  
Technical University of Crete, Greece  
Email: pkarakasis@isc.tuc.gr

Athanasios P. Liavas  
School of Electrical and Computer Engineering  
Technical University of Crete, Greece  
Email: aliavas@isc.tuc.gr

Abstract—We consider the problem of tensor factorization in the cases where one of the factors is constrained to have orthonormal columns. We adopt the alternating optimization framework and derive an efficient algorithm that is also suitable for parallel implementation. We describe in detail a distributed memory implementation of the algorithm on a three-dimensional processor grid. The speedup attained by a message-passing implementation of the algorithm is significant, indicating that it is a competitive candidate for the solution of very large tensor factorization problems with orthogonality constraints.

Index Terms—tensors, tensor factorization, PARAFAC, orthogonality constraints, algorithms, parallel algorithms.

I. INTRODUCTION

Tensors have recently gained great popularity due to their ability to model multiway data dependencies [1], [2], [3], [4]. Tensor factorizations into latent factors are very important for numerous tasks, such as feature selection, dimensionality reduction, compression, data visualization, interpretation and completion, and are usually computed as solutions of optimization problems [1], [2]. The Canonical Decomposition or Canonical Polyadic Decomposition (CANDECOMP or CPD), also known as Parallel Factor Analysis (PARAFAC), and the Tucker Decomposition are the two most widely used tensor factorization models.

The PARAFAC model comes with theoretical background that guarantees essentially unique tensor factorizations under mild conditions. However, the problem of finding a best rank-R approximation of tensors of order 3 (three-way), in the unconstrained case, has no solution, in general [5]. Existence of an optimal solution is guaranteed if one of the factors is constrained to have orthonormal columns [6]. Also, the orthonormally constrained PARAFAC model can be unique under more relaxed conditions than the unconstrained model [7]. In this work, we focus on the PARAFAC model with unimodal orthogonality constraints.

Alternating Optimization (AO) and All-at-Once Optimization (AOO) are among the most commonly used techniques for tensor factorization [2], [8]. Recent work for constrained tensor factorization/completion includes, among others, [9], [10], [11], and [12]. In [12], the authors consider constrained matrix/tensor factorization/completion problems. They adopt the AO framework as outer loop and use the Alternating Direction Method of Multipliers (ADMM) for solving the inner constrained optimization problem for one matrix factor conditioned on the rest. The ADMM offers significant flexibility, due to its ability to efficiently handle a wide range of constraints.

In [13], two parallel algorithms for unconstrained tensor factorization/completion have been developed and results concerning the speedup attained by their Message Passing Interface (MPI) implementations on a multi-core system have been reported. Related work on parallel algorithms for sparse tensor decomposition includes [14] and [15].

A. Contribution

In this work, we focus on large tensor factorization problems with one of the factors constrained to have orthonormal columns. Our aim is to derive an efficient algorithm that is also suitable for parallel implementation. We adopt the AO framework, and develop an algorithm for the solution of the aforementioned problem. We describe in detail a parallel implementation of the algorithm on a three-dimensional processor grid and measure the speedup attained by an MPI implementation of the algorithm. We observe that the proposed algorithm is very efficient, in practice.

B. Notation

Vectors, matrices, and tensors are denoted by small, capital, and calligraphic capital bold letters, respectively; for example, \( \mathbf{x}, \mathbf{X}, \) and \( \mathbf{X}, \mathbb{R}^{I \times J} \) denotes the set of \((I \times J)\) real tensors, while \( \mathbb{R}^{I \times J} \) denotes the set of \((I \times J)\) real matrices. \( \mathbf{I} \) denotes the identity matrix of appropriate dimensions. \( S_{i,j} = \{ \mathbf{X} \in \mathbb{R}^{I \times J} : \mathbf{X}^T \mathbf{X} = \mathbf{I} \} \) denotes the Stiefel manifold formed by all orthonormal \( J \)-frames in \( \mathbb{R}^{I} \). \( \| \cdot \|_F \) denotes the Frobenius norm of the tensor or matrix argument. The outer product of vectors \( \mathbf{a} \in \mathbb{R}^{I \times 1}, \mathbf{b} \in \mathbb{R}^{J \times 1}, \) and \( \mathbf{c} \in \mathbb{R}^{K \times 1} \) is the rank-one tensor \( \mathbf{a} \circ \mathbf{b} \circ \mathbf{c} \in \mathbb{R}^{I \times J \times K} \) with

This work was supported by computational time granted from the Greek Research & Technology Network (GRNET) in the National HPC facility - ARIS - under project ID pa171201.

†The MPI C++ implementation of the algorithm, that we used in our experiments, is available upon request to the authors.
elements \((a \odot b \odot c)(i, j, k) = a(i)b(j)c(k)\). The Khatri-Rao (columnwise Kronecker) product of compatible matrices \(A\) and \(B\) is denoted as \(A \odot B\) and the Hadamard (elementwise) product is denoted as \(A \odot B\). Finally, inequality \(A \succeq B\) means that matrix \(A - B\) is positive semidefinite.

C. Structure

In Section II, we briefly describe the tensor factorization problem with uni-modal orthogonality constraints. In Section III, we briefly describe the Orthogonal Procrustes problem. In Section IV, we derive the AO algorithm and in Section V we describe in detail a parallel implementation of the algorithm. In Section VI, we test the efficiency of the proposed algorithm with numerical experiments in a parallel computing environment. Finally, in Section VII, we conclude the paper.

II. Tensor Factorization with Uni-modal Orthogonality Constraints

Let tensor \(\mathcal{X}^o \in \mathbb{R}^{I \times J \times K}\) admit a factorization of the form

\[
\mathcal{X}^o = [A^o, B^o, C^o] = \sum_{r=1}^R \alpha_r^o \odot b_r^o \odot c_r^o,
\]

where \(A^o = [a_1^o \ldots a_R^o] \in \mathbb{R}^{I \times R}\), \(B^o = [b_1^o \ldots b_R^o] \in \mathbb{R}^{J \times R}\), and \(C^o = [c_1^o \ldots c_R^o] \in \mathbb{R}^{K \times R}\). We observe the noisy tensor \(\mathcal{X} = \mathcal{X}^o + \mathcal{E}\), where \(\mathcal{E} \in \mathbb{R}^{I \times J \times K}\) is the additive noise. Estimates of \(A^o\), \(B^o\), and \(C^o\) can be obtained by computing matrices \(A \in \mathbb{R}^{I \times R}\), \(B \in \mathbb{S}(J,R)\), and \(C \in \mathbb{R}^{K \times R}\) that solve the optimization problem

\[
\min_{A,B \in \mathbb{S}(J,R), C} f_X(A, B, C),
\]

where \(f_X\) is a function measuring the quality of the factorization. A common choice for \(f_X\) is

\[
f_X(A, B, C) = \frac{1}{2} \| \mathcal{X} - [A, B, C] \|^2_F.
\]

If \(\mathcal{Y} = [A, B, C]\), then its matrix unfoldings, with respect to the first, second, and third mode, are given by [3]

\[
\begin{align*}
Y_A &= A (C \odot B)^T, \\
Y_B &= B (C \odot A)^T, \\
Y_C &= C (B \odot A)^T.
\end{align*}
\]

Thus, \(f_X\) can be expressed as

\[
\begin{align*}
f_X(A, B, C) &= \frac{1}{2} \| Y_A - A (C \odot B)^T \|^2_F \\
&= \frac{1}{2} \| Y_B - B (C \odot A)^T \|^2_F \\
&= \frac{1}{2} \| Y_C - C (B \odot A)^T \|^2_F.
\end{align*}
\]

These expressions form the basis for the AO algorithm in the sense that, if we fix two matrix factors, we can update the third by solving a (potentially constrained) least squares problem. The update of matrix factor \(B\) requires the solution of a least squares problem with orthogonality constraints.

III. Orthogonal Procrustes

Given two matrices \(Y \in \mathbb{R}^{N \times M}\) and \(X \in \mathbb{R}^{M \times D}\), the optimization problem

\[
\min_{G \in \mathbb{R}^{(N, D)}} \| Y - GX^T \|^2_F,
\]

is known as Orthogonal Procrustes (OP) and has a closed form solution given by [16, 17]

\[
G_{\text{opt}} = UV^T = M (M^T M)^{-\frac{1}{2}},
\]

where matrices \(U \in \mathbb{R}^{N \times D}\) and \(V \in \mathbb{R}^{D \times D}\) are given by the singular value decomposition of matrix \(M = YX = U \Sigma V^T\).

A. Computational complexity of the OP problem

For later use, we notice that an efficient way of solving the OP problem, after calculating matrix \(M\) with computational complexity \(O(NMDN^2D^2)\) arithmetic operations and when \(\min(N,M) > D\), is the following algorithm:

1) Calculate \(M^T M\), with complexity \(O(ND^2)\);
2) Calculate the eigen-decomposition of \(M^T M = \Sigma V^T\) with complexity \(O(D^3)\);
3) Set \(G = MV\Sigma^{-\frac{1}{2}}V^T\) with complexity \(O(ND^2)\).

Thus, the overall complexity is \(O(ND^2)\) in contrast to computing the singular value decomposition of matrix \(M\) in \(O(N^2D^2D)\). The most demanding computation of this approach is the computation of matrix \(M\).

IV. AO Uni-modal Orthogonal Tensor Factorization

In Algorithm 1, we present the AO Uni-modal Orthogonal Tensor Factorization (AO UOTF) algorithm. We start from point \((A_0, B_0, C_0)\) and solve, in a circular manner, a least squares problem (via function LS Update) for updating factors \(A_k\) and \(C_k\), while we update factor \(B_k\) by solving the OP problem (via function OP Update).

Updating factors \(A_k\) and \(C_k\) can be done as follows:

\[
A_{k+1} = X_A (C_k \odot B_k)[(C_k^T C_k) \odot (B_k^T B_k)]^{-1} \\
= X_A (C_k \odot B_k)[(C_k^T C_k) \odot I]^{-1} \\
= X_A (C_k \odot B_k) D_{C_k}^{-1} C_k
\]

and

\[
C_{k+1} = X_C (B_{k+1} \odot A_{k+1}) [I \odot (A_{k+1}^T A_{k+1})]^{-1} \\
= X_C (B_{k+1} \odot A_{k+1}) D_{A_{k+1}}^{-1} A_{k+1}
\]

that solve the corresponding least squares problems exploiting the orthonormality of factor \(B\). For a matrix \(G\), matrix \(D_{G}^{-1}\) denotes the diagonal matrix with elements the inverses of the corresponding diagonal elements of matrix \(G^T G\).

After the factor updates, we use two functions which have been proven very useful in our experiments, in the sense that they significantly reduce the number of outer iterations necessary to reach convergence. Function Normalize normalizes each column of \(A_{k+1}\) to unit Euclidean norm, putting all the power on the respective columns of \(C_{k+1}\). We denote its
Algorithm 1: AO UOTF

Input: $\mathbf{X}$, $A_0$, $B_0$, $C_0$.

1. Set $k = 0$
2. while (1) do
3.   $A_{k+1} = \text{LS\_Update}(A_k, C_k \odot B_k)$
4.   $B_{k+1} = \text{OP\_Update}(X_B, C_k \odot A_{k+1})$
5.   $C_{k+1} = \text{LS\_Update}(X_C, A_{k+1} \odot B_{k+1})$
6.   $(A_N^{k+1}, C_N^{k+1}) = \text{Normalize}(A_{k+1}, C_{k+1})$
7.   if (termination condition is TRUE) then break; endif
8.   $(A_{k+1}, B_{k+1}, C_{k+1}) = \text{Accelerate}(A_N^{k+1}, A_k, B_k, C_{k+1}, C_N^{k+1})$
9.   $k = k + 1$
10. return $A_k, B_k, C_k$.

output as $A_N^{k+1}$ and $C_N^{k+1}$. Function Accelerate implements the acceleration technique used in the function parafac of n-way toolbox [18], briefly described in [19].

We can use various termination criteria for the AO UOTF algorithm based, for example, on the (relative) change of the cost function and/or the latent factors.

V. PARALLEL IMPLEMENTATION

In this section, we consider the implementation of the AO UOTF algorithm in a computing environment with $p = p_A \times p_B \times p_C$ processing elements. In general, the $p$ processors form a three-dimensional Cartesian grid, with each processor denoted as $p_{i_A,i_B,i_C}$, for $i_A = 1, \ldots, p_A$, $i_B = 1, \ldots, p_B$, and $i_C = 1, \ldots, p_C$.

A. Variable partitionings and data allocation

In order to describe the parallel implementation, we introduce certain partitionings of the factor matrices and the tensor matricizations. We partition the factor matrix $A_k$ into $p_A$ block rows as

$$A_k = \begin{bmatrix} (A_1^k)^T & \cdots & (A_{p_A}^k)^T \end{bmatrix}^T,$$  \hspace{1cm} (10)

with $A_i^k \in \mathbb{R}^{J_A \times R_i}$, for $i_A = 1, \ldots, p_A$. We partition accordingly the matricization $X_A$ and get

$$X_A = \begin{bmatrix} (X_A^1)^T & \cdots & (X_A^{p_A})^T \end{bmatrix}^T,$$  \hspace{1cm} (11)

with $X_i^A \in \mathbb{R}^{N_A \times J_A}$, for $i_A = 1, \ldots, p_A$. In a similar manner, we partition $B_k$ and $X_B$ into $p_B$ block rows, each of size $\frac{1}{p_B} \times R$ and $\frac{1}{p_B} \times I$, respectively, and $C_k$ and $X_C$ into $p_C$ block rows, each of size $\frac{1}{p_C} \times R$ and $\frac{1}{p_C} \times I$, respectively.

We partition tensor $\mathbf{X}$ into $p$ subtensors, according to the partitioning of the factor matrices (see Fig. 1), and allocate its parts to the various processors. Thus, processor $p_{i_A,i_B,i_C}$ receives subtensor $X_{i_A,i_B,i_C}^A$, as defined in (9), at the top of this page.

We assume that, at the end of the $k$-th outer AO iteration,

a) processor $p_{i_A,i_B,i_C}$ knows $A_i^B, B_i^B, C_i^C$;

b) all processors know $A_i^C A_k$ and $C_i^C C_k$ (note that, due the orthogonality constraints $B_i^B B_k = I$).

B. Communication Groups

We start by defining certain communication groups, also known as communicators [20], over subsets of the $p$ processors. The communicators are used for the efficient collaborative implementation of specific computational tasks, as explained in detail later.

We define $p_A$ two-dimensional processor groups, each involving the $p_B \times p_C$ processors $p_{i_A,i_B,i_C}$, for $i_A = 1, \ldots, p_A$ (horizontal layers), with the $i_A$-th processor group used for the collaborative update of $A_i^B$. Similarly, we define groups $p_{i_B,i_C}$ for $i_B = 1, \ldots, p_B$, and $p_{i_C,i}$ for $i_C = 1, \ldots, p_C$, which are used for the collaborative update of $B_i^B$ and $C_i^C$, respectively.

We define $p_B \times p_C$ one-dimensional processor groups, each involving the $p_C$ processors $p_{i_B,i_C}$. Each of these groups is used for the collaborative computation of $A_{k+1}^B A_k$. Similarly, we define groups $p_{i_B,i_C}$ and $p_{i_A,i_B,i_C}$, which are used for the collaborative update of $B_k$ and computation of $C_{k+1}^C C_k$, respectively.
C. Factor Update Implementation

We describe in detail the updates of $A_k$ and $B_k$. The update of $C_k$ is similar to the update of $A_k$.

Collaborative Update of $A_k$: The update of $A_k$ is achieved via the parallel updates of $A_k^{iA}$, for $i_A = 1,\ldots,p_A$, and consists of the following stages:

1) Processors $p_{i_A,\ldots,p_A}$, for $i_A = 1,\ldots,p_A$, collaboratively compute the $\frac{l}{p_A} \times R$ matrix

$$\overrightarrow{W}_{i_A} := X_{i_A}^{iA} (C_k \odot B_k),$$

and the result is scattered among the processors in the group; thus, each processor in the group receives $\frac{l}{p_A p_B p_C}$ successive rows of $\overrightarrow{W}_{i_A}$. Term $\overrightarrow{W}_{i_A}$ can be computed collaboratively because

$$X_{i_A}^{iA} (C_k \odot B_k) = \sum_{i_B=1}^{p_B} \sum_{i_C=1}^{p_C} X_{i_A,i_B,i_C}^{iA} (C_k^{iC} \odot B_k^{iC}),$$

where $X_{i_A,i_B,i_C}^{iA}$ is the matricization of $X_{i_A,i_B,i_C}$, with respect to the first mode; processor $p_{i_A,i_B,i_C}$ knows $X_{i_A,i_B,i_C}^{iA}$, $B_k^{iC}$, and $C_k^{iC}$, and computes the corresponding term of (13). The sum is computed and scattered among processors $p_{i_A,\ldots,p_A}$ via a reduce-scatter operation.

2) Each processor in the group $p_{i_A,\ldots,p_A}$ uses the scattered part of $\overrightarrow{W}_{i_A}$ and $\overrightarrow{Z}_{i_A} = I \otimes C_k^{iC} B_k$, and computes the updated part of $A_{k+1}^{iA}$, via a left-squares update. Then, the updated processors are all-gathered at the processors of the group $p_{i_A,\ldots,p_A}$ so that all processors in the group learn the whole updated $A_{k+1}^{iA}$.

3) By applying an all-reduce operation to $(A_{k+1}^{iA})^T A_{k+1}^{iA}$, for $i_A = 1,\ldots,p_A$, on each of the single dimensional processor groups $p_{i_A,i_B,i_C}$, for $i_B = 1,\ldots,p_B$ and $i_C = 1,\ldots,p_C$, all $p$ processors learn $A_{k+1}^{iA}$.

Collaborative Update of $B_k$: The update of $B_k$ is achieved via the parallel updates of $B_k^{iB}$, for $i_B = 1,\ldots,p_B$, and consists of the following stages:

1) Processors $p_{i_B,\ldots,p_B}$, for $i_B = 1,\ldots,p_B$, collaboratively compute the $\frac{l}{p_B} \times R$ matrix

$$\overrightarrow{W}_{i_B} := X_{i_B}^{iB} (C_k \odot A_{k+1}),$$

by applying an all-reduce operation since

$$X_{i_B}^{iB} (C_k \odot A_{k+1}) = \sum_{i_A=1}^{p_A} \sum_{i_C=1}^{p_C} X_{i_A,i_B,i_C}^{iA} (C_k^{iC} \odot A_{k+1}^{iA}),$$

where $X_{i_A,i_B,i_C}^{iA}$ is the matricization of $X_{i_A,i_B,i_C}$, with respect to the second mode.

2) Processors $p_{i_A,\ldots,p_A}$, for $i_A = 1,\ldots,p_A$ and $i_C = 1,\ldots,p_C$, collaboratively compute the $R \times R$ matrix

$$\overrightarrow{W}_B \overleftarrow{W}_B = \sum_{i_B=1}^{p_B} (\overrightarrow{W}_{i_B})^T \overrightarrow{W}_{i_B}$$

by applying an all-reduce operation. We notice that at the end of this step, all $p$ processors know matrix $\overrightarrow{W}_B \overleftarrow{W}_B$.

3) Each processor $p_{i_A,i_B,i_C}$, for $i_A = 1,\ldots,p_A$, $i_B = 1,\ldots,p_B$, and $i_C = 1,\ldots,p_C$, computes the updated partial factor $B_{k+1}^{iB}$ as

$$B_{k+1}^{iB} = \overrightarrow{W}_{i_B} (\overrightarrow{W}_B \overleftarrow{W}_B)^{-\frac{1}{2}}.$$

We note that the Euclidean norms of the columns of the new matrices $A_{k+1}$ and $C_{k+1}$ appear on the diagonals of $A_{k+1}^{iA}$ and $C_{k+1}^{iC}$, which are known to all processors. Thus, no additional communication is necessary for the normalization of the updated partial factors.

After the normalization step of the $(k+1)$-st AO iteration, processor $p_{i_A,i_B,i_C}$ knows the parts of the normalized factors, that is, $A_{k+1}^{iA}$, $B_{k+1}^{iB}$, $C_{k+1}^{iC}$, as well as $A_{k+1}^{iA'}$, $B_{k+1}^{iB'}$, and $C_{k+1}^{iC'}$, and can collaboratively implement the acceleration mechanism as explained in detail in [19].

D. Communication Cost

We focus on the parallel updates of $A_k^{iA}$, for $i_A = 1,\ldots,p_A$, and $B_k^{iB}$, for $i_B = 1,\ldots,p_B$, and present results concerning the associated communication cost. The communication cost of the update of $C_k^{iC}$, for $i_C = 1,\ldots,p_C$, can be computed by following analogous steps to those used for the computation of the communication cost of the update of $A_k^{iA}$.

We assume that an $n$-word message is transferred from one process to another with communication cost $t_s + t_w m$, where $t_s$ is the latency, or startup time for the data transfer, and $t_w$ is the word transfer time [20].

Updating $A_k$ in parallel: Communication occurs at three algorithm execution points.

1) The $\frac{l}{p_A} \times R$ matrix $\overrightarrow{W}_A$ is computed and scattered among the $p_B \times p_C$ processors of group $p_{i_A,\ldots,p_A}$, using a reduce-scatter operation, with communication cost [20, §4.2]

$$C_1^A = t_s (p_B + p_C - 2) + t_w \frac{IR}{p_B p_C} (p_B p_C - 1).$$

2) Processors $p_{i_A,\ldots,p_A}$ learn the updated $A_{k+1}^{iA}$ through an all-gather operation on its updated parts, each of dimension $\frac{l}{p_A p_B p_C} \times R$, with communication cost [20, §4.2]

$$C_2^A = t_s (p_B + p_C - 2) + t_w \frac{IR}{p_B p_C} (p_B p_C - 1).$$

3) Matrix $A_{k+1}^T A_{k+1}$ is computed by using an all-reduce operation on quantities $(A_{k+1}^{iA})^T A_{k+1}^{iA}$, for $i_A = 1,\ldots,p_A$, on each single-dimensional processor group $p_{i_A,i_B,i_C}$, with communication cost [20, §4.3]

$$C_3^A = (t_s + t_w R^2) \log_2 p_A.$$

Updating $B_k$ in parallel: Communication occurs at two algorithm execution points.

1) The $\frac{l}{p_B} \times R$ matrix $\overrightarrow{W}_B$ is computed among the $p_A \times p_C$ processors of group $p_{i_A,\ldots,p_A}$, using an all-reduce operation, with communication cost [20, §4.2]

$$C_1^B = t_s + t_w \frac{J}{p_B} R \log_2 (p_A p_C).$$
2) Matrix $\overline{W}_B^T \overline{W}_B$ is computed by using an all-reduce operation on quantities $(\overline{W}_B^{in})^T \overline{W}_B^{in}$ within each single-dimensional processor group $p_{A_1 \ldots A_r}$, with communication cost [20, §4.3]

$$C_2^B = (t_w + t_w'R^2) \log_2 p_B. \quad (18)$$

The communication that takes place during the acceleration step involves scalar quantities and, thus, is ignored.

When we are dealing with large messages, the $t_w$ terms dominate the communication cost. Thus, if we ignore the startup time, the total communication time, for updating $A_k$, is

$$C_A = t_w \left( \frac{2IR}{p_A p_B p_C} (p_B p_C - 1) + R^2 \log_2 p_A \right) \approx t_w \left( \frac{2IR}{p_A} + R^2 \log_2 p_A \right) \approx 2IRt_w \frac{1}{p_A} \quad (19)$$

with the second approximation being accurate for $R \ll \frac{1}{p_A}$.

The presence of $p_A$ in the denominator of the last expression of (19) implies that our implementation is scalable in the sense that, if we double $I$, then we can have (approximately) the same communication cost per processor by doubling $p_A$. Analogous results hold for the update of factor $C_k$.

As for the update of factor $B_k$, if we ignore the startup time, the total communication time is

$$C_B = t_w \left( \frac{JR}{p_B} \log_2 (p_A p_C) + R^2 \log_2 p_B \right) \approx t_w \left( \frac{JR}{p_B} \log_2 (p_A p_C) \right) \quad (20)$$

with the approximation being accurate for $R \ll \frac{1}{p_B}$. We again observe that our implementation is scalable in the above mentioned sense.

VI. NUMERICAL EXPERIMENTS

In this section, we present results obtained from the MPI implementation described in detail in Section V. The program is executed on a DELL PowerEdge R820 system with Sandy-Bridge - Intel(R) Xeon(R) CPU E5 - 4650v2 (in total, 16 nodes with 40 cores each at 2.4 Gz) and 512 GB RAM per node. The matrix operations are implemented using routines of the C++ library Eigen [21]. We assume a noiseless tensor $\mathcal{X}$, whose true latent factors $A^o$ and $C^o$ have i.i.d elements, uniformly distributed in $[0, 1]$, while true latent factor $B^o$ was produced from the left singular vectors of a matrix with i.i.d elements, uniformly distributed in $[0, 1]$.

The AO terminates at iteration $k$ if

$$\text{RFE}(A_k, B_k, C_k) < 10^{-3},$$

where

$$\text{RFE}(A, B, C) := \frac{\|\mathcal{X} - [A, B, C]\|_F}{\|\mathcal{X}\|_F}. \quad (21)$$

We test the behavior of our implementation for various tensor sizes and rank $R = 15, 50, 100$. The performance metric we compute is the speedup attained using $p = p_A \times p_B \times p_C$ processors.

In Figures 2–4, we plot the speedup for the following cases$^2$:

1) Cubic tensor: we set $I = J = K = 2000$ and implement the algorithm on a grid with $p_A = p_B = p_C = \sqrt{p}$, for $p = 1, 8, 27, 64, 125, 216, 343$.

2) Two large dimensions: we set $I = 5000, J = 320, K = 5000$ and implement the algorithm on a grid with $p_A = p_B = p_C = \sqrt{p}$, for $p = 1, 4, 9, 36, 64, 121, 225, 361$.

3) One large dimension: we set $I = 400, J = 50000, K = 400$ and implement the algorithm on a grid with $p_A = p_B = p_C = 1$, for $p = 1, 8, 27, 64, 125, 216, 343$.

In order to highlight the need of parallel processing for the decomposition of very large tensors, we quote the serial execution times ($p = 1$) in Table I. We observe that, in all cases, we attain significant speedup, which is rather insensitive to the tensor shape and rank.

![Fig. 2. Speedup achieved for a $2000 \times 2000 \times 2000$ tensor with $p$ cores, for $p = 1, 8, 27, 64, 125, 216, 343$.](image)

VII. CONCLUSION

We considered the UOTF problem. We adopted the AO framework and described in detail a parallel implementation of the AO UOTF algorithm on a three-dimensional processor grid. The speedup attained by the MPI implementation of the algorithm was significant in all the cases we considered, rendering our algorithm a strong candidate for the solution of very large-scale dense UOTF problems.

Future work includes the development and implementation of efficient algorithms for UOTF with further constraints, like nonnegativity and sparsity.

$^2$To the best of our knowledge, there is no other parallel algorithm solving the UOTF problem, thus, we cannot compare with any competing state-of-the-art algorithm.
TABLE I
EXECUTION TIMES FOR p = 1 OVER DIFFERENT TENSOR SIZES AND RANKS

<table>
<thead>
<tr>
<th>Size</th>
<th>R</th>
<th>Time of Execution (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 × 2000 × 2000</td>
<td>15</td>
<td>6,272.35</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>14,714.09</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>34,381.45</td>
</tr>
<tr>
<td>5000 × 320 × 5000</td>
<td>15</td>
<td>7,345.13</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>17,908.56</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>42,628.99</td>
</tr>
<tr>
<td>400 × 5000 × 400</td>
<td>15</td>
<td>6,434.69</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>17,790.75</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>36,599.60</td>
</tr>
</tbody>
</table>

Fig. 3. Speedup achieved for a 5000 × 320 × 5000 tensor with p cores, for p = 1, 4, 9, 36, 64, 121, 225, 361.

Fig. 4. Speedup achieved for a 400 × 50000 × 400 tensor with p cores, for p = 1, 8, 27, 64, 125, 216, 343.

REFERENCES