2018 IEEE 25th International Conference on High Performance Computing (HiPC)

Parallel Nonnegative CP Decomposition of Dense Tensors

Grey Ballard and Koby Hayashi
Wake Forest University
Winston Salem NC 27109
Email: {ballard,hayakb13}@wfu.edu

Ramakrishnan Kannan
Oak Ridge National Laboratory
Oak Ridge, TN 37830
Email: kannanr@ornl.gov

***Abstract*—The CP tensor decomposition is a low-rank approximation of a tensor. We present a distributed-memory parallel algorithm and implementation of an alternating optimization method for computing a CP decomposition of dense tensors that can enforce nonnegativity of the computed low-rank factors. The principal task is to parallelize the Matricized-Tensor Times Khatri-Rao Product (MTTKRP) bottleneck subcomputation. The algorithm is computation efficient, using dimension trees to avoid redundant computation across MTTKRPs within the alternating method. Our approach is also communication efficient, using a data distribution and parallel algorithm across a multidimensional processor grid that can be tuned to minimize communication. We benchmark our software on synthetic as well as hyperspectral image and neuroscience dynamic functional connectivity data, demonstrating that our algorithm scales well to 100s of nodes (up to 4096 cores) and is faster and more general than the currently available parallel software.**

I. Introduction

The CP decomposition is a low-rank approximation of a multi-dimensional array, or tensor, which generalizes matrix approximations like the truncated singular value decomposition. It approximates the input tensor by a sum of rank-one tensors, which are outer products of vectors. CP is often used for finding hidden patterns, or latent factors, within tensor data, particularly when the goal is to interpret the factors, and it is popular within the signal processing, machine learning, and scientific computing communities.

To aid in interpretability, domain-specific constraints are often imposed on the computed factors. We focus in this paper on dense tensors (when nearly all of the tensor entries are nonzero) and on constraining solutions to have nonnegative entries, which is useful when the tensor data itself is nonnegative. Formally, NNCP can be defined as

$$\min_{\mathbf{H}^{(i)} \geqslant 0} \left\| \mathcal{A} - \sum_{r=1}^{R} \mathbf{H}^{(1)}(:, r) \circ \cdots \circ \mathbf{H}^{(N)}(:, r) \right\|^2 \qquad (1)$$

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

where $\mathbf{H}^{(1)}(:, i) \circ \cdots \circ \mathbf{H}^{(N)}(:, i)$ is the outer product of the i^{th} vector from all the N factors that yields a rank one tensor $\mathcal{M}$ and $\sum_{r=1}^{R} \mathbf{H}^{(1)}(:, r) \circ \cdots \circ \mathbf{H}^{(N)}(:, r)$ results in a sum of R rank one tensors that will be of the same dimension as the input tensor $\mathcal{A}$. For example, in imaging and microscopy applications, tensor values often correspond to intensities, and NNCP can be used to cluster and analyze the data in a lower-dimensional space [1]. In this work, we consider two such applications: a series of time-lapse hyperspectral images [2] and a dynamic functional correlation data set generated from functional magnetic resonance images of human brains [3].

One approach to handling multidimensional data is to "matricize" it, combining sets of modes to reshape the data into a matrix, so that standard matrix methods like principal component analysis or nonnegative matrix factorization can be applied. While this approach can be effective in certain cases, reshaping the data destroys multidimensional relationships among entries that the matrix methods cannot recover. By maintaining the tensor structure of the data, the low-rank representations preserve these relationships, often producing better and more interpretable results.

However, tensor methods are more complicated both mathematically and computationally. The kernel computations within standard algorithms for computing NNCP can be formulated as matrix computations, but the complicated layout of tensors in memory prevents the straightforward use of BLAS and LAPACK libraries. In particular, the matrix formulation of subcomputations involve different views of the tensor data, so no single layout yields a column- or row-major matrix layout for all subcomputations. Likewise, the parallelization approach for tensor methods is not a straightforward application of parallel matrix computation algorithms.

In developing an efficient parallel algorithm for computing a NNCP of a dense tensor, the key is to parallelize the bottleneck computation known as Matricized-Tensor Times Khatri-Rao Product (MTTKRP), which is performed repeatedly for each mode of the tensor. The parallelization must load balance the computation, minimize communication across processors, and distribute the results so that the rest of the computation can be performed independently. In our algorithm, not only do we load balance the computation, but we also compute and store temporary values that can be used across MTTKRPs of different modes using a technique known as dimension

978-1-5386-8386-6/18/$31.00 ©2018 IEEE
DOI 10.1109/HiPC.2018.00012

trees, significantly reducing the computational cost compared to standard approaches. Our parallelization strategy also avoids communicating tensor entries and minimizes the communication of factor matrix entries, helping the algorithm to remain computation bound and scalable to high core counts.

As we detail in the related work, the general techniques for reducing computation and communication have been used in similar contexts. The recomputation avoidance was proposed in a sequential algorithm [4], the parallelization scheme was proposed and analyzed for general tensors [**?**], and the algorithm was implemented for 3D tensors [6].

We summarize our main contributions as follows:

- we present the first distributed-memory parallel implementation of NNCP algorithms for arbitrary-dimension dense tensors,
- we optimize the use of dimension trees for dense tensors, avoiding recomputation across multiple MTTKRPs,
- our parallel algorithm is communication optimal with a carefully chosen processor grid,
- we demonstrate a performance improvement of up to $2.2\times$ over the existing state-of-the-art parallel software on 3D tensors and efficient parallel scaling of up to $1771\times$ on 4096 cores.

II. Preliminaries

A. Notation

Tensors will be denoted using Euler script (e.g., $\mathcal{T}$), matrices will be denoted with uppercase boldface (e.g., $\mathbf{M}$), vectors will be denoted with lowercase boldface (e.g., $\mathbf{v}$), and scalars will not be boldface (e.g., s). We use Matlab style notation to index into tensors, matrices, and vectors, and we use 1-indexing. For example, $\mathbf{M}(:, c)$ gives the cth column of the matrix M.

We use $\circ$ to denote the outer product of two or more vectors. The Hadamard product is the element-wise matrix product and will be denoted using $*$. The Khatri-Rao product, abbreviated KRP, will be denoted with $\odot$. Given matrices $\mathbf{A}$ and $\mathbf{B}$ that are $I_A \times R$ and $I_B \times R$, the KRP $\mathbf{K} = \mathbf{A} \odot \mathbf{B}$ is $I_A I_B \times R$. It can be thought of as a row-wise Hadamard product, where $\mathbf{K}(j + I_B(i-1), :) = \mathbf{A}(i, :) * \mathbf{B}(j, :)$, or a column-wise Kronecker product, where $\mathbf{K}(:, c) = \mathbf{A}(:, c) \otimes \mathbf{B}(:, c)$.

The CP decomposition of a tensor (also referred to as the CANDECOMP/PARAFAC or canonical polyadic decomposition) is a low-rank approximation of a tensor, where the approximation is a sum of rank-one tensors and each rank-one tensor is the outer product of vectors. We use the notation $\mathcal{A} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \sum_{r=1}^{R} \mathbf{H}^{(1)}(:, r) \circ \cdots \circ \mathbf{H}^{(N)}(:, r)$ to represent a rank-R CP model, where $\mathbf{H}^{(n)}$ is called a factor matrix and collects the mode-n vectors of the rank-one tensors as columns. The columns of the factor matrices are often normalized, with weights collected into an auxiliary vector $\boldsymbol{\lambda}$ of length R; in this case we use the notation $[\![\boldsymbol{\lambda}; \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \sum_{r=1}^{R} \lambda_r \mathbf{H}^{(1)}(:, r) \circ \cdots \circ \mathbf{H}^{(N)}(:, r)$.

A Nonnegative CP decomposition (NNCP) constrains the factor matrices to have nonnegative values. In this work, we are interested in NNCP models that are good approximations to $\mathcal{A}$ in the least squares sense. That is, we seek $\min_{\mathbf{H}^{(i)} \geqslant 0} \|\mathcal{A} - [\![\boldsymbol{\lambda}; \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]\|$, where the tensor norm is a generalization of the matrix Frobenius or vector 2-norm, the square root of the sum of squares of the entries.

The nth mode matricized tensor denoted by $\mathbf{A}_{(n)}$ is a $I_n \times I/I_n$ matrix formed by organizing the nth mode fibers of a tensor $\mathcal{X}$ with dimensions $I_1 \times \ldots \times I_N$ (and $I = \prod I_n$) into the columns of a matrix. The Matricized-Tensor Times Khatri-Rao Product or MTTKRP will be central to this work and takes the form $\mathbf{M}^{(n)} = \mathbf{A}_{(n)}\mathbf{K}^{(n)}$, where $\mathbf{K}^{(n)}$ is the Khatri-Rao product of the all the factor matrices except $\mathbf{H}^{(n)}$ defined as $\mathbf{K}^{(n)} = \mathbf{H}^{(N)} \odot \cdots \odot \mathbf{H}^{(n+1)} \odot \mathbf{H}^{(n-1)} \odot \cdots \odot \mathbf{H}^{(1)}$.

B. Block Coordinate Descent for NNCP

While there are multiple optimization methods to compute NNCP, we will focus on a class of methods that use Block Coordinate Descent (BCD), which is also known as the nonlinear Gauss-Seidel method. In BCD, the variables are partitioned into blocks, and each variable block is cyclically updated to optimality with all other blocks fixed. For details on the convergence properties and comparisons of BCD methods for nonnegative matrix and tensor decomposition problems, see [8]. We consider BCD methods for NNCP that choose the entire factor matrices as the blocks, which is also often referred to as Alternating Least Squares. In this case, every subproblem is a linear nonnegative least squares problem. Formally, the following problem is solved iteratively for $n = 1 \cdots N$:

$$\mathbf{H}^{(n)} \leftarrow \underset{\mathbf{H} \geqslant 0}{\arg\min} \left\| \mathbf{S}^{(n)} \mathbf{H}^T - {\mathbf{M}^{(n)}}^T \right\|_F^2,$$

where $\mathbf{S}^{(n)} = {\mathbf{K}^{(n)}}^T \mathbf{K}^{(n)}$, the Gram matrix of the Khatri-Rao product of the fixed factor matrices. The number of outer iterations to convergence is problem dependent but typically ranges from 10s to 1000s.

Algorithm 1 shows the pseudocode for BCD applied to NNCP. Lines 11, 12 and 14 compute matrices involved in the gradients of the subproblem objective functions, and line 13 uses those matrices to update the current factor matrix.

The NLS-Update in line 13 can be implemented in different ways. In a faithful BCD algorithm, the subproblems are solved exactly; in this case, the subproblem is a nonnegative linear least squares problem, which is convex. We use the Block Principal Pivoting (BPP) method [8], [9], which is an active-set-like method, to solve the subproblem exactly.

However, as discussed in [10] for the matrix case, there are other reasonable alternatives to updating the factor matrix without solving the subproblem exactly. For example, we can more efficiently update individual columns of the factor matrix as is done in the Hierarchical Alternating Least Squares (HALS) method [11]. In this case, the update rule is

$$\mathbf{H}^{(n)}(:, r) \leftarrow \left[\mathbf{H}^{(n)}(:, r) + \mathbf{M}^{(n)}(:, r) - (\mathbf{H}^{(n)}\mathbf{S}^{(n)})(:, r) \right]_+$$

which involves the same matrices $\mathbf{M}^{(n)}$ and $\mathbf{S}^{(n)}$ as BPP. Other possible BCD methods include Alternating Optimization and Alternating Direction Method of Multipliers (AO-ADMM) [12], [13] and Nestrov-based algorithms [14]. The parallel

algorithm presented in this paper is generally agnostic to the approach used to solve the nonnegative least squares subproblems, as all these methods are bottlenecked by the subroutine they have in common, the MTTKRP.

Algorithm 1 $[\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] = \text{NNCP}(\mathcal{A}, R)$

Require: $\mathcal{A}$ is $I_1 \times \cdots \times I_N$ tensor, R is approximation rank
1: % *Initialize data*
2: **for** $n = 2$ to N **do**
3: Initialize $\mathbf{H}^{(n)}$
4: $\mathbf{G}^{(n)} = \mathbf{H}^{(n)\mathsf{T}}\mathbf{H}^{(n)}$
5: **end for**
6: % *Compute NNCP approximation*
7: **while** not converged **do**
8: % *Perform outer iteration of BCD*
9: **for** $n = 1$ to N **do**
10: % *Compute new factor matrix in nth mode*
11: $\mathbf{M}^{(n)} = \text{MTTKRP}(\mathcal{A}, \{\mathbf{H}^{(i)}\}, n)$
12: $\mathbf{S}^{(n)} = \mathbf{G}^{(1)} * \cdots * \mathbf{G}^{(n-1)} * \mathbf{G}^{(n+1)} * \cdots * \mathbf{G}^{(N)}$
13: $\mathbf{H}^{(n)} = \text{NLS-Update}(\mathbf{S}^{(n)}, \mathbf{M}^{(n)})$
14: $\mathbf{G}^{(n)} = \mathbf{H}^{(n)\mathsf{T}}\mathbf{H}^{(n)}$
15: **end for**
16: **end while**
Ensure: $\mathcal{A} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$

C. Parallel Computation Model

To analyze our algorithms we use the MPI model of distributed-memory parallel computation, where we assume a fully connected network. Sending a message of W words from one processor to another costs $\alpha + \beta W$, where α is the latency and β to be the per word or bandwidth cost. In particular, we will use collective communication over groups of P processors, and we will assume the use of efficient algorithms [15], [16]. An All-Reduce sums data initially distributed across processors and stores the result of size W redundantly on every processor. An All-Gather collects data initially distributed across processors and stores the union of size W redundantly on all processors. A Reduce-Scatter sums data initially distributed across processors and partitions the result across processors. The communication cost of each of these collectives is $\alpha \cdot O(\log P) + \beta \cdot O(W(P-1)/P)$. Reduction operations also include a flop cost but we will omit it because it is usually dominated by communication.

III. Related Work

The formulation of NNCP with least squares error and algorithms for computing it go back to [17], [18], developed in part as a generalization of nonnegative matrix factorization algorithms [19] to tensors. Sidiropoulos et al. [20] provide a more detailed and complete survey that includes basic tensor factorization models with and without constraints, broad coverage of algorithms, and recent driving applications. The tensor operations discussed and the notation used in this paper follow Kolda and Bader's survey [21].

Recently, there has been growing interest in scaling tensor operations to bigger data and more processors in both the data mining/machine learning and the high performance computing communities. For sparse tensors, there have been parallelization efforts to compute CP decompositions both on shared-memory platforms [22], [23] as well as distributed-memory platforms [24]–[26], and these approaches can be generalized to constrained problems [13]. The focus of this work is on dense tensors, but many of the ideas for sparse tensors are applicable to the dense case, including parallel data distributions, communication pattern, and techniques to avoid recomputation across modes.

In particular, Liavas et al. [6] extend a parallel algorithm designed for sparse tensors [25] to the 3D dense case. They use the "medium-grained" dense tensor distribution and row-wise factor matrix distribution, which is exactly the same as our distribution strategy (see section IV-C2), and they use a Nesterov-based algorithm to enforce the nonnegativity constraints. Their code is publicly available, and we compare our performance with theirs in section V. A similar data distribution and parallel algorithm for computing a single dense MTTKRP computation is proposed by Ballard, Knight, and Rouse [**?**]. They prove that the algorithm is communication optimal, but they do not provide an implementation. Another approach to parallelizing NNCP decomposition of dense tensors is presented by Phan and Cichocki [27], but they use a dynamic tensor factorization, which performs different, more independent computations across processors.

The idea of using dimension trees (discussed in section IV-A) to avoid recomputation within MTTKRPs across modes is introduced in [4] for computing the CP decomposition of dense tensors. It has also been used for sparse CP [23], [26] and other tensor computations [24].

IV. Algorithm

A. Dimension Trees

An important optimization of the CP-ALS algorithm is to re-use temporary values across inner iterations [4], [23], [28], [29]. To illustrate the idea, consider a 3-way tensor $\mathcal{X}$ approximated by $[\![\mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$ and the two MTTKRP computations $\mathbf{M}^{(1)} = \mathbf{X}_{(1)}(\mathbf{W} \odot \mathbf{V})$ and $\mathbf{M}^{(2)} = \mathbf{X}_{(2)}(\mathbf{W} \odot \mathbf{U})$ used to update factor matrices $\mathbf{U}$ and $\mathbf{V}$, respectively. The underlined parts of the expressions correspond to the shared dependence of the outputs on the tensor $\mathcal{X}$ and the third factor matrix $\mathbf{W}$. Indeed, a temporary quantity, which we refer to as a *partial MTTKRP*, can be computed and re-used across the two MTTKRP expressions. We refer to the computation that combines the temporary quantity with the other factor matrix to complete the MTTKRP computation as a multi-tensor-times-vector or *multi-TTV*, as it consists of multiple operations that multiply a tensor times a set of vectors, each corresponding to a different mode.

To understand the steps of the partial MTTKRP and multi-TTV operations in more detail, we consider $\mathcal{X}$ to be $I \times J \times K$ and $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ to have R columns. Then $m_{ir}^{(1)} = \sum_{j,k} x_{ijk} v_{jr} w_{kr} = \sum_j v_{jr} \sum_k x_{ijk} w_{kr} = \sum_j v_{jr} t_{ijr}$, where $\mathcal{T}$ is an $I \times J \times R$ tensor that is the result of a partial MTTKRP between tensor $\mathcal{X}$ and the single factor matrix W.

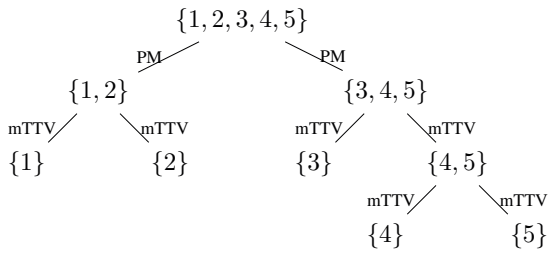
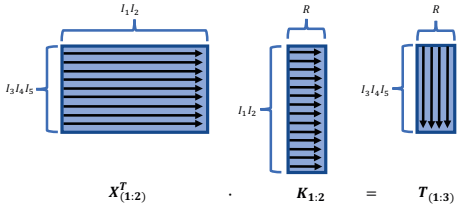
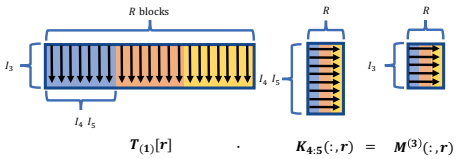


Fig. 1. Dimension tree example for $N = 5$. The data associated with the root node is the original tensor, the data associated with the leaf nodes are MTTKRP results, and the data associated with internal nodes are temporary tensors. Edges labeled with PM correspond to partial MTTKRP computations, and edges labeled with mTTV correspond to multi-TTV computations.



(a) Partial MTTKRP to compute node $\{3, 4, 5\}$ from root node $\{1, 2, 3, 4, 5\}$, executed via one GEMM call.



(b) Multi-TTV to compute node $\{3\}$ from node $\{3, 4, 5\}$, executed via R GEMV calls. Here $\mathbf{T}_{(1)}[r]$ refers to the rth contiguous block of $\mathbf{T}_{(1)}$.

Fig. 2. Data layout and dimensions for two example computations in dimension tree shown in Figure 1. In this notation, $\mathbf{X}_{(1:2)}$ is the matricization of input tensor $\mathcal{X}$ with respect to modes 3 through 5, $\mathbf{K}_{1:2} = \mathbf{H}^{(2)} \odot \mathbf{H}^{(1)}$, $\mathcal{T}$ is the temporary $I_3 \times I_4 \times I_5 \times R$ tensor corresponding to node $\{3, 4, 5\}$ in the dimension tree, $\mathbf{K}_{4:5} = \mathbf{H}^{(5)} \odot \mathbf{H}^{(4)}$, and $\mathbf{M}^{(3)}$ is the MTTKRP result for mode 3.

Likewise, $m^{(2)}_{jr} = \sum_{i,k} x_{ijk} u_{ir} w_{kr} = \sum_i u_{ir} \sum_k x_{ijk} w_{kr} = \sum_i u_{ir} t_{ijr}$, and we see that the temporary tensor $\mathcal{T}$ can be re-used. From these expressions, we can also see that computing $\mathcal{T}$ (a partial MTTKRP) corresponds to a matrix-matrix multiplication, and computing each of $\mathbf{M}^{(1)}$ and $\mathbf{M}^{(2)}$ from $\mathcal{T}$ (a multi-TTV) corresponds to R independent matrix-vector multiplications. We compute $\mathbf{M}^{(3)}$ using a full MTTKRP.

For a larger number of modes, a more general approach can organize the temporary quantities to be used over a maximal number of MTTKRPs. The general approach can yield significant benefit, decreasing the computation by a factor of approximately $N/2$ for dense N-way tensors. The idea is introduced in [4], but we adopt the terminology and notation of *dimension trees* used for sparse tensors in [28], [29]. In this notation, the root node is labeled $\{1, \ldots, N\}$ and corresponds to the original tensor, a leaf is labeled $\{n\}$ and corresponds to the nth MTTKRP result, and an internal node is labeled by a set of modes $\{i, \ldots, j\}$ and corresponds to a temporary tensor whose values contribute to the MTTKRP results of modes $i, \ldots, j$.

Figure 1 illustrates a dimension tree for the case $N = 5$. Various shapes of binary trees are possible [4], [29]. For dense tensors, the computational cost is dominated by the root's branches, which correspond to partial MTTKRP computations. We perform the splitting of modes at the root so that modes are chosen contiguously with the respect to the layout of the tensor entries in memory. In this way, each partial MTTKRP can be performed via BLAS's GEMM interface without reordering tensor entries in memory. All other edges in a tree correspond to multi-TTVs and are typically much cheaper. By organizing the memory layout of temporary quantities, the multi-TTV operations can be performed via a sequence of calls using BLAS's GEMV interface. With BLAS, we are able to obtain high performance and on-node parallelism.

Figure 2 shows the data layout and dimensions of a partial MTTKRP and a multi-TTV taken from the example dimension tree in Figure 1. Figure 2a shows a partial MTTKRP between the input tensor $\mathcal{X}$ and the Khatri-Rao product of the factor matrices in modes 1 and 2, which produces a temporary tensor $\mathcal{T}$ corresponding to the $\{3, 4, 5\}$ node in the dimension tree. The key to efficiency in this computation is that the matricization of $\mathcal{X}$ that assigns modes 1 through 2 to rows and modes 3 through 5 to columns is already column-major in memory. Thus, we can use the GEMM interface and compute the temporary tensor $\mathcal{T}$ without reordering any tensor entries. Figure 2b depicts a multi-TTV that computes the results $\mathbf{M}^{(3)}$ from $\mathcal{T}$ and the factor matrices in modes 4 and 5. Here, the tensor $\mathcal{T}$ is matricized with respect to only its first mode (of dimension I_3), but this matricization is also column-major in memory. We choose the ordering of the modes of $\mathcal{T}$ such that each of R contiguous blocks is used to compute one column of the output matrix via a matrix-vector operation with a corresponding column of the Khatri-Rao product of the other factor matrices.

No matter how the dimension tree is designed, the computational cost of each partial MTTKRP is $O(IR)$, where I is the number of tensor entries and R is the rank of the CP decomposition. This is the same operation count as a full MTTKRP. The computational cost of a multi-TTV is the number of entries in the temporary tensor, which is the product of a *subset* of the original tensor dimensions multiplied by R. Thus, it is computationally cheaper than the partial MTTKRPs, but it is also memory bandwidth bound.

The other subroutine necessary for implementing the dimension tree approach is the Khatri-Rao product of sets of factor matrices. We implement the operation as a row-wise Hadamard product of a set of factor matrix rows, and we use OpenMP parallelization to obtain on-node parallelism. The computational cost of this operation is also typically lower order, but the running time in practice suffers from also being

memory bandwidth bound.

B. Relative Error Computation

Given a model $\mathcal{M} = [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$, we compute the relative error $\|\mathcal{A} - \mathcal{M}\|/\|\mathcal{A}\|$ efficiently by using the identity $\|\mathcal{A} - \mathcal{M}\|^2 = \|\mathcal{A}\|^2 - 2\langle \mathcal{A}, \mathcal{M}\rangle + \|\mathcal{M}\|^2$. The quantity $\|\mathcal{A}\|$ is fixed, and the other two terms can be computed cheaply given the temporary matrices computed during the course of the BCD algorithm. The second term can be computed using the identity $\langle \mathcal{A}, \mathcal{M}\rangle = \langle \mathbf{M}^{(N)}, \mathbf{H}^{(N)}\rangle$, where $\mathbf{M}^{(N)} = \mathbf{A}_{(N)}(\mathbf{H}^{(N-1)} \odot \cdots \odot \mathbf{H}^{(1)})$ is the MTTKRP result in the Nth mode. The third term can be computed using the identity $\|\mathcal{M}\|^2 = \mathbf{1}^\mathsf{T}(\mathbf{S}^{(N)} * \mathbf{H}^{(N)\mathsf{T}}\mathbf{H}^{(N)})\mathbf{1}$ where $\mathbf{S}^{(N)} = \mathbf{H}^{(1)\mathsf{T}}\mathbf{H}^{(1)} * \cdots * \mathbf{H}^{(N-1)\mathsf{T}}\mathbf{H}^{(N-1)}$. Both matrices $\mathbf{M}^{(N)}$ and $\mathbf{S}^{(N)}$ are computed during the course of the BCD algorithm for updating the factor matrix $\mathbf{H}^{(N)}$. The extra computation involved in computing the relative error is negligible. These identities have been used previously [14], [21], [25], [30].

C. Parallel Algorithm

Algorithm 2 $[\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!] =$ Par-NNCP$(\mathcal{A}, R)$

Require: $\mathcal{A}$ is an $I_1 \times \cdots \times I_N$ tensor distributed across a $P_1 \times \cdots \times P_N$ grid of P processors, so that $\mathcal{A}_\mathbf{p}$ is $(I_1/P_1) \times \cdots \times (I_N/P_N)$ and is owned by processor $\mathbf{p} = (p_1, \ldots, p_N)$, R is rank of approximation

1: **for** $n = 2$ to N **do**
2: Initialize $\mathbf{H}^{(n)}_\mathbf{p}$ of dimensions $(I_n/P) \times R$
3: $\overline{\mathbf{G}} =$ Local-SYRK$(\mathbf{H}^{(n)}_\mathbf{p})$
4: $\mathbf{G}^{(n)} =$ All-Reduce$(\overline{\mathbf{G}}, $ ALL-PROCS$)$
5: $\mathbf{H}^{(n)}_{p_n} =$ All-Gather$(\mathbf{H}^{(n)}_\mathbf{p}, $ PROC-SLICE$(n, p_n))$
6: **end for**
7: % *Compute NNCP approximation*
8: **while** not converged **do**
9: % *Perform outer iteration of BCD*
10: **for** $n = 1$ to N **do**
11: % *Compute new factor matrix in nth mode*
12: $\overline{\mathbf{M}} =$ Local-MTTKRP$(\mathcal{A}_{p_1 \cdots p_N}, \{\mathbf{H}^{(i)}_{p_i}\}, n)$
13: $\mathbf{M}^{(n)}_\mathbf{p} =$ Reduce-Scatter$(\overline{\mathbf{M}}, $ PROC-SLICE$(n, p_n))$
14: $\mathbf{S}^{(n)} = \mathbf{G}^{(1)} * \cdots * \mathbf{G}^{(n-1)} * \mathbf{G}^{(n+1)} * \cdots * \mathbf{G}^{(N)}$
15: $\mathbf{H}^{(n)}_\mathbf{p} =$ Local-NLS-Update$(\mathbf{S}^{(n)}, \mathbf{M}^{(n)}_\mathbf{p})$
16: % *Organize data for later modes*
17: $\overline{\mathbf{G}} = \mathbf{H}^{(n)\mathsf{T}}_\mathbf{p}\mathbf{H}^{(n)}_\mathbf{p}$
18: $\mathbf{G}^{(n)} =$ All-Reduce$(\overline{\mathbf{G}}, $ ALL-PROCS$)$
19: $\mathbf{H}^{(n)}_{p_n} =$ All-Gather$(\mathbf{H}^{(n)}_\mathbf{p}, $ PROC-SLICE$(n, p_n))$
20: **end for**
21: **end while**

Ensure: $\mathcal{A} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$

Ensure: Local matrices: $\mathbf{H}^{(n)}_\mathbf{p}$ is $(I_n/P) \times R$ and owned by processor $\mathbf{p} = (p_1, \ldots, p_N)$, for $1 \leqslant n \leqslant N$, $\boldsymbol{\lambda}$ stored redundantly on every processor

1) Algorithm Overview: The basic sequential algorithm is given in Algorithm 1, and the parallel version is given in Algorithm 2. We will refer to both the inner iteration, in which one factor matrix is updated (line 10 to line 20), and the outer iteration, in which all factor matrices are updated (line 8 to line 21). In the parallel algorithm, the processors are organized into a logical multidimensional grid with as many modes as the data tensor. The communication patterns used in the algorithm are the MPI collectives All-Reduce, Reduce-Scatter, and All-Gather. The processor communicators (across which collectives are performed) include all processors and the sets of processors within the same processor slice. Processors within a mode-n slice all have the same nth coordinate.

The method of enforcing the nonnegativity constraints of the linear least squares solve (or update) generally affects only local computation because each row of a factor matrix can be updated independently. In our algorithm, each processor solves the linear problem or computes the update for its subset of rows (see line 15). The most expensive (and most complicated) part of the parallel algorithm is the computation of the MTTKRP, which corresponds to lines 12, 13 and 19.

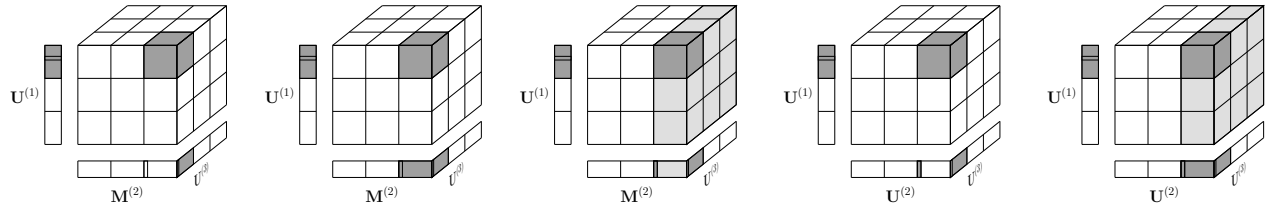
The details that are omitted from this presentation of the algorithm include the normalization of each computed factor matrix and the computation of the residual error at the end of an outer iteration. The computations involve both local computation and communication, but their costs are negligible.

2) Data Distribution: Given a logical processor grid of processors $P_1 \times \cdots \times P_N$, we distribute the tensor $\mathcal{A}$ in a block or Cartesian partition. Each processor owns a local tensor of dimensions $(I_1/P_1) \times \cdots \times (I_N/P_N)$, and only one copy of the tensor is stored. Locally, the tensor is stored linearly, with entries ordered in a natural mode-descending way that generalizes column-major layout of matrices. Given a processor $\mathbf{p} = (p_1, \ldots, p_N)$, we denote its local tensor $\mathcal{A}_\mathbf{p}$.

Each factor matrix is distributed across processors in a block row partition, so that each processor owns a subset of the rows. We use the notation $\mathbf{H}^{(n)}_\mathbf{p}$, which has dimensions $I_n/P \times R$ to denote the local part of the nth factor matrix stored on processor $\mathbf{p}$. However, we also make use a redundant distribution of the factor matrices across processors, because all processors in a mode-n processor slice need access to the same entries of $\mathbf{H}^{(n)}$ to perform their computations. The notation $\mathbf{H}^{(n)}_{p_n}$ denotes the $I_n/P_n \times R$ submatrix of $\mathbf{H}^{(n)}$ that is redundantly stored on all processors whose nth coordinate is p_n (there are P/P_n such processors).

Other matrices involved in the algorithm include $\mathbf{M}^{(n)}_\mathbf{p}$, which is the result of the MTTKRP computation and has the same distribution scheme as $\mathbf{H}^{(n)}_\mathbf{p}$, and $\mathbf{G}^{(n)}$, which is the $R \times R$ Gram matrix of the factor matrix $\mathbf{H}^{(n)}$ and is stored redundantly on all processors.

3) Inner Iteration: The inner iteration is displayed graphically in Figure 3 for a 3-way example and an update of the 2nd factor matrix. The main idea is that at the start of the nth inner iteration (Figure 3a), all of the data is in place for each processor to perform a local MTTKRP computation. This means that all processors in a slice redundantly own the same rows of the corresponding factor matrix (for all modes except n). After the local MTTKRP is computed (Figure 3b), each processor has computed a contribution to a subset of the rows of the global MTTKRP $\mathbf{M}^{(n)}$, but its contribution must be summed up with the contributions of all other processors in its mode-n slice. This summation is performed with a Reduce-Scatter collective across the mode-n



(a) Start nth iteration with redundant subset of rows of each input matrix.

(b) Compute local MTTKRP for contribution to output matrix $\mathbf{M}^{(2)}$.

(c) Reduce-Scatter to compute and distribute rows of $\mathbf{M}^{(2)}$.

(d) Compute local NLS update to obtain $\mathbf{H}^{(2)}_{\mathbf{p}}$ from $\mathbf{M}^{(2)}_{\mathbf{p}}$ (and $\mathbf{S}^{(2)}$).

(e) All-Gather to collect rows of $\mathbf{H}^{(2)}$ needed for later inner iterations.

Fig. 3. Illustration of 2nd inner iteration of Par-NNCP algorithm for 3-way tensor on a $3 \times 3 \times 3$ processor grid, showing data distribution, communication, and computation across steps. Highlighted areas correspond to processor $(1, 3, 1)$ and its processor slice with which it communicates. The column normalization and computation of $\mathbf{G}^{(2)}$, which involve communication across all processors, is not shown here.

processor slice that achieves a row-wise partition of the result (in Figure 3c, the light gray shading corresponds to the rows of $\mathbf{M}^{(2)}$ to which processor $(1, 3, 1)$ contributes and the dark gray shading corresponds to the rows it receives as output). The output distribution of the Reduce-Scatter is designed so that afterwards, the update of the factor matrix in that mode can be performed row-wise in parallel. Along with $\mathbf{S}^{(n)}$, which can be computed locally, each processor updates its own rows of the factor matrix given its rows of the MTTKRP result (Figure 3d). The remainder of the inner iteration is preparing and distributing the new factor matrix data for future inner iterations, which includes an All-Gather of the newly computed factor matrix $\mathbf{H}^{(n)}$ across mode-n processor slices (Figure 3e) and recomputing $\mathbf{G}^{(n)} = \mathbf{H}^{(n)^\mathsf{T}}\mathbf{H}^{(n)}$.

4) Analysis: We will analyze the cost of a single outer iteration. While the number of outer iterations is sensitive to the NLS method used, the outer iteration time is generally comparable across NLS methods.

a) Computation*:* The local computation occurs at lines 12, 14, 15 and 17. The cost of line 14 is $O(NR^2)$, the cost of line 15 is $O(R^3 I_n/P)$, which is a loose upper bound for BPP and other methods [31], and the cost of line 17 is $O(R(I_n/P)^2)$. The sum of these three costs across all inner iterations is $O(R^2N^2 + (R^3/P)\sum I_n + (R/P^2)\sum I_n^2)$, which is dominated by the cost of the MTTKRP. When using dimension trees to perform the MTTKRP (line 12), we compute the cost amortized over all inner iterations. In this case, the cost is dominated by the two partial MTTKRP computations (from the root of the tree), which together are $O((R/P)\prod I_n) = O(IR/P)$ and dominate the costs of the multi-TTVs. We note that this cost involves the product of all the tensor dimensions, which is why it dominates, and we note that it scales linearly with P.

b) Communication*:* The communication within the inner iteration occurs at lines 13, 18 and 19. Line 18 involves $O(R^2)$ data and a collective across all processors. Lines 13 and 19 involve $O(I_n R/P_n)$ data across a subset of P/P_n processors. Thus, the All-Reduce dominates the latency cost and the Reduce-Scatter/All-Gather dominate the bandwidth cost, for a total outer iteration communication cost of $O(R\sum I_n/P_n)$ words and $O(N \log P)$ messages. If the optimal processor grid can be chosen to minimize communication (assuming P is sufficiently factorable), then the bandwidth cost can achieve a value of $O(NRI^{1/N}/P^{1/N})$ by making the local tensors as cubical as possible. We note that this cost scales with $P^{1/N}$, which is far from linear scaling. However, it is proportional to the geometric mean of the tensor dimensions (on the order of one tensor dimension), which is much less than the computation dependence on the product of all dimensions.

c) Memory*:* The algorithm requires extra local memory to run. Aside from the memory required to store the local tensor of $O(I/P)$ words and factor matrices of cumulative size $O((R/P)\sum I_n)$, each processor must be able to store a redundant subset of the rows of the factor matrices it needs to perform MTTKRP computations. This corresponds to storing P/P_n redundant copies of every factor matrix, which results in a local memory requirement of $O(R\sum I_n/P_n)$ for a general processor grid. The processor grid that minimizes communication also minimizes local memory, and the extra memory requirement can be as low as $O(NRI^{1/N}/P^{1/N})$, which is typically dominated by $O(I/P)$.

The dimension tree algorithm also requires extra temporary memory space, but the space required tends to be much smaller than what is required to store the local tensor. If the tensor dimensions can be partitioned into two parts with approximately equal geometric means, the extra memory requirement for running a dimension tree is as small as $O(R\sqrt{I/P})$, which is also typically dominated by $O(I/P)$.

V. Performance Results

A. Datasets

1) Hyperspectral Images (HSI): For comparison with previous work [6], we consider the same 3D hyperspectral imaging dataset called "Souto_wood_pile" [2]. NNCP is often used on HSI data sets for classification and blind source separation of materials with differing spectral signatures. The hyperspectral datacube has dimensions $1024 \times 1344 \times 33$ and represents a set of 33 grayscale images of size 1344 $\times$ 1024 pixels sampled at wavelengths 400, 410, ..., 720 nm, with each pixel value representing spectral radiance in $Wm^{-2}sr^{-1}nm^{-1}$. We also consider the Nogueiró scene dataset, which is a sequence of 9 time-lapse HSI images of the same scene acquired at about

1-hour intervals. In each scene, hyperspectral images were acquired at about 1-hour intervals. Each Nogueiró scene HSI image has the same properties as the Souto_wood_pile data set, yielding a $1024 \times 1344 \times 33 \times 9$ tensor.

2) Dynamic Functional Connectivity (dFC): We also consider dynamic functional connectivity datasets that are generated from fMRI images of human brains. Given a 4D fMRI data set of voxel measurements across multiple timesteps, voxels containing brain data are partitioned into a set of regions of interest (specified using domain-specific knowledge), and a single time-series signal is aggregated for each region of interest. Then, an instantaneous correlation is computed for each time point and pair of regions, and this process is repeated for a number of subjects. Computing a CP decomposition of this data helps to discover patterns of brain connectivity among different regions and also differentiate among individuals. For our representative dFC data set, we consider 246 brain regions, which yields 30,012 unique pairs of regions, 1200 times steps, and 500 subjects, a $30{,}012 \times 1200 \times 500$ tensor [3], [32].

3) Synthetic: Our synthetic data sets are constructed from a CP model with an exact low rank with no added noise. In this case we can confirm that the residual error of our algorithm with a random start converges to zero. For the purposes of benchmarking, we run a fixed number of iterations of the BCD algorithm rather than using a convergence check.

B. Machine Details

The entire experimentation was performed on Eos, a supercomputer at the Oak Ridge Leadership Computing Facility. Eos is a 736-node Cray XC30 cluster of Intel Xeon E5-2670 processors with a total of 47.104TB of memory. Its compute nodes are organized in blades where each blade contains 4 nodes, and every node has 2 sockets with 8 physical cores and 64GB memory. In total, Eos contains 11,776 traditional processor cores and our experiments used up to 4,096 cores (35% of the machine).

Our objective of the implementation is using open source software as much as possible to promote reproducibility and reuse of our code. We use Armadillo [33] for matrix representation and operations. In Armadillo, the elements of the dense matrix are stored in column major order. For dense BLAS and LAPACK operations, we linked Armadillo with the default LAPACK/BLAS wrappers from Cray. We use multithreaded LAPACK/BLAS except as noted in Section V-C. We use GNU C++ Compiler (g++ (GCC) 6.3.0) and Cray's MPI library.

C. Comparison Implementations

The implementation proposed by Liavas et al. [6] is the only publicly available distributed-memory software (of which we are aware) for computing the CP decomposition of dense tensors, with or without constraints. We use the acronym NbAO-NTF for Nesterov-based Alternating Optimization Nonnegative Tensor Factorization to refer to their code.

It is based on the same parallel algorithm as our implementation, though it is limited to 3D tensors. The code uses MPI collectives for communication and Eigen [34] as an interface to BLAS and LAPACK. We compiled the code linked to BLAS/LAPACK wrappers from Cray (the same BLAS implementation used by our code) but we were unable to run multithreaded BLAS with their code. For fair comparison, we use a flat MPI configuration (one MPI process per core) on all comparisons between the two implementations (Figures 4, 5 and 10). For all other experiments, we use hybrid MPI (one MPI process per node) and OpenMP and multithreaded BLAS for on-node parallelism.

We also point out a difference between the Nesterov-based algorithm and the BPP algorithm for solving the NLS subproblems. The Nesterov-based algorithm attempts an acceleration step using a linear combination of the current and proposed future step; however, it re-computes the residual error before deciding whether or not to accept or reject the acceleration step. This residual error cannot always be computed cheaply, using the technique described in section IV-B, and it contributes significantly (approximately 25%) to the overall run time. Because the BPP algorithm does not require this extra computation, and studying convergence behavior of the different NLS algorithms is beyond the scope of this work, we remove the time spent in the acceleration step of NbAO-NTF in all our comparisons.

Our proposed algorithm uses dimension trees, but we also benchmark our implementation without that optimization to highlight its importance. We use an existing implementation to perform the individual MTTKRPs [35] with this approach.

D. Strong Scaling

We perform two strong scaling experiments on 3D tensors to compare performance with NbAO-NTF. The experiments use a cubical synthetic tensor and the HSI image used in [6].

The performance on the cubical synthetic tensor is shown in Figure 4. We can observe from the figure that all the three algorithms scale nearly linearly as the problem remains compute bound. Our algorithm (with the dimension tree optimization) achieves a speedup of $1771\times$ on 4096 cores over the same implementation running on 1 core. Recall from that the computation scales linearly with $1/P$ while the communication scales with $1/P^{1/N} = 1/P^{1/3}$. As is evident from the figure, the communication cost does not degrade performance even for thousands of cores. Our proposed algorithm with dimension trees is 35% faster than NbAO-NTF at 512 cores (with similar relative difference for other core counts). This performance improvement is due in large part to the 50% reduction in arithmetic provided by the dimension tree optimization. There is little difference in performance between our implementation without dimension trees and NbAO-NTF.

Figure 5 shows the strong scaling on the HSI data. In this case, our proposed algorithm with dimension trees is over $2\times$ faster than NbAO-NTF, but part of this speedup is due to differences in the NLS update algorithms. For the low core count, the dimension tree provides a 60% speedup compared to the MTTKRP time in NbAO-NTF. At the high core counts, the local MTTKRP is no longer the dominating cost.

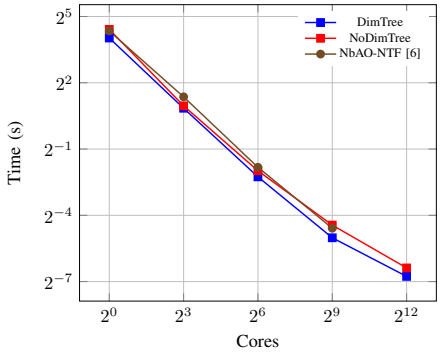


Fig. 4. Strong scaling of 3D synthetic tensor with dimension $1024 \times 1024 \times 1024$ and rank 32 on processor grids $2^k \times 2^k \times 2^k$ for $k \in \{0, 1, 2, 3, 4\}$.

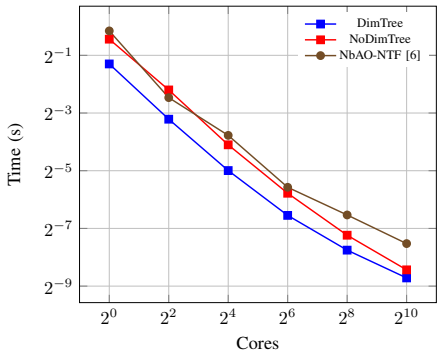


Fig. 5. Strong scaling of 3D HSI real world data with dimension $1024 \times 1344 \times 33$ and rank 32 on processor grids of $k \times k \times 1$ for $k \in 1, 2, 4, 8, 16, 32$.

In Figure 6, we benchmark performance for a 5D cubical tensor with each dimension set to 64. Because the tensor is 5D, we can no longer compare against NbAO-NTF. We see a $13-16\times$ speed up using a dimension tree over not using one. As predicted, the dimension tree optimization saves relatively more arithmetic for higher-order tensors. However, the reduction in leading order flop cost is only $2.5\times$ for $N = 5$; the rest of the speedup comes from more efficient DGEMM performance and avoiding memory-bound KRP computation. That is, although the flop count of KRP computation is lower order, it still contributes to the run time because it is inefficient. Also, for tensors with balanced dimensions, the dimension tree approach yields more favorable shapes for DGEMM.

E. Weak Scaling Time Breakdown

We also perform a weak scaling experiment to understand the time it takes to solve bigger problems with more processors. In this experiment, we use a synthetic 4D tensor and keep the amount of tensor data assigned to each processor constant, with tensor and processor grid of dimensions $128k \times 128k \times 128k \times 128k$ and $k \times k \times k \times k$ for $k \in \{1, 2, 3, 4\}$ and the rank fixed at 32. The results of the breakdown plot is shown in Figure 7. In this case, the algorithm is compute bound with and without the use of the dimension tree, so the total time of the weak scaling remains fixed for both cases.

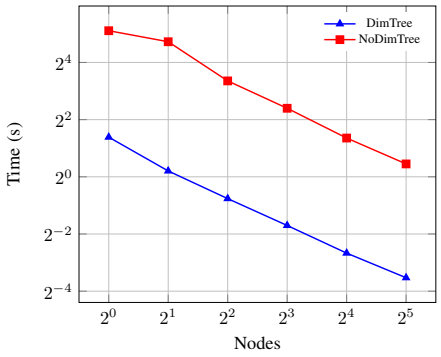


Fig. 6. Strong scaling of 5D synthetic tensor with dimension $64 \times 64 \times 64 \times 64 \times 64$ and rank 32 on processor grids $1 \times 1 \times 1 \times 1 \times 1$, $2 \times 1 \times 1 \times 1 \times 1$, $\ldots$, $2 \times 2 \times 2 \times 2 \times 2$.

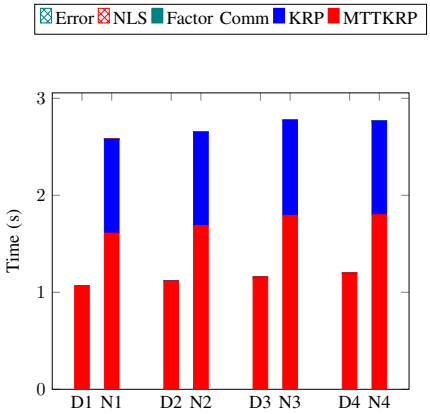


Fig. 7. Weak scaling of 4D synthetic tensors with (D) and without (N) the use of dimension trees. The tensor and processor grid dimensions are $128k \times 128k \times 128k \times 128k$ and $k \times k \times k \times k$ for $k \in \{1, 2, 3, 4\}$, and the rank is fixed at 32. The reported times are per iteration.

Using a dimension tree, the time is completely dominated by the MTTKRP computation. Without using a dimension tree, we observe that the KRP is expensive and yields a $2.5\times$ slower total run time even in the 4D case.

F. Varying Processor Grid

In order to illustrate the effect of processor grid choice on running time, we show in Figure 8 a time breakdown over various processor grid choices for a 4D problem on 81 processors. Because the tensor is cubical and 81 has a restricted factorization into 4 numbers, there are 5 distinct processor grids. The overall takeaway is that the processor grid has very little effect on running time; in this experiment there is less than 10% variation in overall time. While the optimal processor grid reduced the communication time by approximately $3\times$ compared to the other processor grids, the running time is dominated by local computation, so it had little effect on overall time. Furthermore, adjusting the processor grid affects the local tensor dimensions and the performance of the local computations, and the optimal processor grid led to slower local performance. For $R = 10$, all of the local

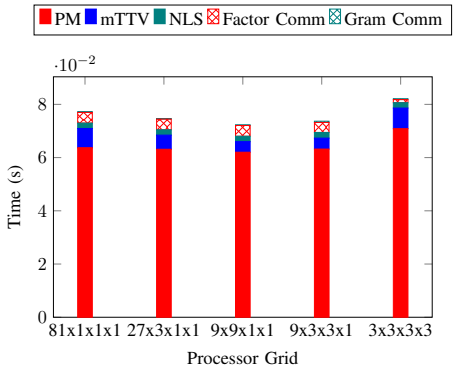


Fig. 8. Time breakdown for $243 \times 243 \times 243 \times 243$ tensor and rank $R = 10$ on 81 processors for varying processor grids.

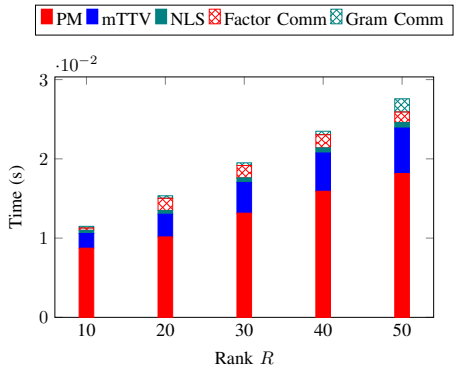


Fig. 9. Per-iteration time breakdown of our implementation (using dimension trees) over various ranks for a time-lapse HSI dataset with dimensions $1344 \times 1024 \times 33 \times 9$ on 64 processors arranged in a $8 \times 8 \times 1 \times 1$ grid.

computation is memory bandwidth bound, and we believe the variations in running times to be effects of some temporary quantities fitting into smaller levels of cache.

G. Varying Approximation Rank

One of the challenges of the CP (and NNCP) decomposition in practice is the choice of decomposition rank. The most common technique is to compute multiple CP decompositions for various ranks. As the rank R increases, the approximation error $\|\mathcal{A} - \mathcal{M}\|$ decreases with the better approximation power of more parameters. However, the benefit of increasing R eventually diminishes if the data is truly low rank. Towards this end, we experiment with various values of R to observe the relative increase in time for two real-world data sets.

Figure 9 shows the time breakdown of our implementation using a dimension tree on the 4D HSI dataset for $R = \{10, \ldots, 50\}$. We observe an overall time increase with increased R, but each part of the computation scales slightly differently. The multi-TTV computation (*mTTV*) scales linearly with the increasing R, whereas the partial MTTKRP (*PM*) is scaling super-linearly. This is because mTTV is cast as matrix-vector products (DGEMV) and PM is cast as matrix-matrix products (DGEMM). As R increases from 10 to 50, DGEMM performance improves but DGEMV performance is constant. The local NLS time is increasing with $O(R^3)$ as expected and the All-Reduce required for the Gram matrices scales with $O(R^2)$, becoming a significant cost for larger R.

In Figure 10 we compare performance for various ranks R across all 3 algorithms, again used flat MPI. Starting at $R = 10$ we see the largest speed up of $2\times$ for our implementation with a dimension tree over NbAO-NTF. This is due to a combination of the dimension tree performing fewer flops in the MTTKRPs and KRPs. However, as the rank increases this speed up diminishes to $1.6\times$. The loss of speed up is a result of the fact that, as we observed in Figure 9, the multi-TTV operations do not scale as well as the partial-MTTKRPs for increasing R. Again, the performance of our implementation without using dimension trees is comparable to NbAO-NTF.

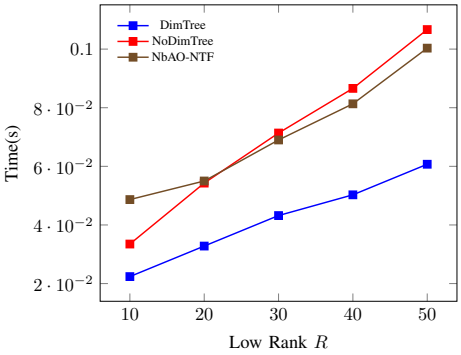


Fig. 10. Overall running time for dFC dataset with dimensions $30{,}012 \times 1200 \times 500$ on 1440 cores arranged in a $2 \times 6 \times 120$ processor grid and varying choices of rank R.

VI. Conclusion

In this work, we present a new implementation for distributed-memory NNCP. The algorithm is general enough to handle any number of modes in the data tensor and can be adapted to use any NLS algorithm within the context of BCD (ALS). We use a dimension tree optimization to avoid unnecessary recomputation within the bottleneck local MTTKRP computation, and we use an efficient parallelization that minimizes communication cost. Our performance results show the ability to scale well, and we show favorable performance in comparison to state-of-the-art software for 3D tensors.

In particular, the performance results demonstrate that computing NNCP for dense tensors involves heavy computation relative to the sizes of the computed factor matrices. By avoiding the communication of tensor entries and communicating only the factor matrices, the parallel algorithm is nearly always compute bound. This observation is supported by the theoretical analysis: although the communication does not scale as well with P, the total amount of data depends on a sum of dimensions rather the product of the dimensions, which determines the total amount of computation.

We can also conclude from the performance results that the dimension tree optimization is the key to performance

improvement over the state-of-the-art approaches. For 3D tensors, we observe a benefit larger than the theoretical 50% reduction in computation, and for larger numbers of modes, the improvement is only magnified. Besides the reduction in flops, the dimension tree approach enjoys better DGEMM performance and avoids memory-bound KRP computations. Furthermore, we see that tuning the processor grid had much less effect on overall performance. Not only do reductions in communication not matter as much as computation, but different local tensor sizes can also cause variations in local performance that outweigh the savings in communication.

ACKNOWLEDGMENTS

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy. This project was partially funded by the Laboratory Director's Research and Development fund at ORNL.

This work has also been funded in part by the Laboratory-Directed Research & Development program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This material is also based upon work supported by the National Science Foundation Grant No. ACI-1642385.

REFERENCES

[1] S. Jesse, M. Chi *et al.*, "Using multivariate analysis of scanning-Ronchigram data to reveal material functionality," *Microscopy and Microanalysis*, vol. 22, pp. 292–293, 07 2016.

[2] D. H. Foster, K. Amano, and S. M. Nascimento, "Time-lapse ratios of cone excitations in natural scenes," *Vision Research*, vol. 120, pp. 45 – 60, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0042698915001121

[3] D. V. Essen, K. Ugurbil *et al.*, "The Human Connectome Project: a data acquisition perspective," *Neuroimage*, vol. 62, no. 4, pp. 2222–2231, 2012.

[4] A.-H. Phan, P. Tichavsky, and A. Cichocki, "Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations," *IEEE Transactions on Signal Processing*, vol. 61, no. 19, pp. 4834–4846, Oct 2013.

[5] G. Ballard, N. Knight, and K. Rouse, "Communication lower bounds for matricized tensor times Khatri-Rao product," arXiv, Tech. Rep. 1708.07401, 2017.

[6] A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, "Nesterov-based alternating optimization for nonnegative tensor factorization: Algorithm and parallel implementation," *IEEE Transactions on Signal Processing*, Nov 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8119874/

[7] D. P. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1999.

[8] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework," *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, Feb 2014. [Online]. Available: https://doi.org/10.1007/s10898-013-0035-4

[9] J. Kim and H. Park, "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011.

[10] R. Kannan, G. Ballard, and H. Park, "MPI-FAUN: An MPI-based framework for alternating-updating nonnegative matrix factorization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 544–558, 2018.

[11] A. Cichocki and A.-H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E92-A, pp. 708–721, 2009.

[12] K. Huang, N. D. Sidiropoulos, and A. P. Liavas, "Efficient algorithms for universally constrained matrix and tensor factorization," in *23rd European Signal Processing Conference*. IEEE, 2015, pp. 2521–2525.

[13] S. Smith, A. Beri, and G. Karypis, "Constrained tensor factorization with accelerated AO-ADMM," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 111–120.

[14] A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, "Nesterov-based parallel algorithm for large-scale nonnegative tensor factorization," in *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2017, pp. 5895–5899.

[15] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005. [Online]. Available: http://hpc.sagepub.com/content/19/1/49.abstract

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.1206

[17] P. Paatero, "A weighted non-negative least squares algorithm for three-way PARAFAC factor analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 38, no. 2, pp. 223 – 242, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169743997000312

[18] M. Welling and M. Weber, "Positive tensor factorization," *Pattern Recognition Letters*, vol. 22, no. 12, pp. 1255 – 1261, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865501000708

[19] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, p. 788, 1999.

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

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

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

[23] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *IEEE International Parallel and Distributed Processing Symposium*, May 2017, pp. 1048–1057.

[24] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*, 2016, pp. 103–112. [Online]. Available: https://doi.org/10.1109/ICPP.2016.19

[25] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *IEEE 30th International Parallel and Distributed Processing Symposium*, May 2016, pp. 902–911.

[26] O. Kaya and B. Uçar, "Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees," *SIAM J. Scientific Computing*, vol. 40, no. 1, 2018. [Online]. Available: https://doi.org/10.1137/16M1102744

[27] A. H. Phan and A. Cichocki, "PARAFAC algorithms for large-scale problems," *Neurocomputing*, vol. 74, no. 11, pp. 1970–1984, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231211000415

[28] O. Kaya and B. Uçar, "Parallel CP decomposition of sparse tensors using dimension trees," Inria, Tech. Rep. RR-8976, Nov. 2016. [Online]. Available: https://hal.inria.fr/hal-01397464

[29] O. Kaya, "High performance parallel algorithms for tensor decompositions," Ph.D. dissertation, University of Lyon, Sep. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01623523

[30] A.-H. Phan, P. Tichavsky, and A. Cichocki, "TENSORBOX: a MATLAB package for tensor decomposition," 2013. [Online]. Available: http://www.bsp.brain.riken.jp/~phan/tensorbox.php

[31] R. Kannan, G. Ballard, and H. Park, "A high-performance parallel algorithm for nonnegative matrix factorization," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, February 2016, pp. 9:1–9:11. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851152

[32] M. J. Tobia, K. Hayashi, G. Ballard, I. H. Gotlib, and C. E. Waugh, "Dynamic functional connectivity and individual differences in emotions during social stress," *Human Brain Mapping*, vol. 38, no. 12, pp. 6185–6205, 2017. [Online]. Available: http://dx.doi.org/10.1002/hbm.23821

[33] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," NICTA, Tech. Rep., 2010. [Online]. Available: http://arma.sourceforge.net/armadillo_nicta_2010.pdf

[34] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[35] K. Hayashi, G. Ballard, Y. Jiang, and M. J. Tobia, "Shared-memory parallelization of MTTKRP for dense tensors," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2018, pp. 393–394. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178522