SAND2020-6977J

TuckerMPI: A Parallel C++/MPI Software Package for Large-scale Data Compression via the Tucker Tensor Decomposition

GREY BALLARD, Wake Forest University
ALICIA KLINVEX and TAMARA G. KOLDA, Sandia National Laboratories

Our goal is compression of massive-scale grid-structured data, such as the multi-terabyte output of a high-fidelity computational simulation. For such data sets, we have developed a new software package called TuckerMPI, a parallel C++/MPI software package for compressing distributed data. The approach is based on treating the data as a tensor, i.e., a multidimensional array, and computing its truncated Tucker decomposition, a higher-order analogue to the truncated singular value decomposition of a matrix. The result is a low-rank approximation of the original tensor-structured data. Compression efficiency is achieved by detecting latent global structure within the data, which we contrast to most compression methods that are focused on local structure. In this work, we describe TuckerMPI, our implementation of the truncated Tucker decomposition, including details of the data distribution and in-memory layouts, the parallel and serial implementations of the key kernels, and analysis of the storage, communication, and computational costs. We test the software on 4.5 and 6.7 terabyte data sets distributed across 100 s of nodes (1,000 s of MPI processes), achieving compression ratios between 100 and 200,000×, which equates to 99–99.999% compression (depending on the desired accuracy) in substantially less time than it would take to even read the same dataset from a parallel file system. Moreover, we show that our method also allows for reconstruction of partial or down-sampled data on a single node, without a parallel computer so long as the reconstructed portion is small enough to fit on a single machine, e.g., in the instance of reconstructing/visualizing a single down-sampled time step or computing summary statistics. The code is available at https://gitlab.com/tensors/TuckerMPI.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; **Computations on matrices**;

Additional Key Words and Phrases: Tucker decomposition, tensor decomposition, higher-order singular value decomposition (HOSVD)

ACM Reference format:
Grey Ballard, Alicia Klinvex, and Tamara G. Kolda. 2020. TuckerMPI: A Parallel C++/MPI Software Package for Large-scale Data Compression via the Tucker Tensor Decomposition. *ACM Trans. Math. Softw.* 46, 2, Article 13 (May 2020), 31 pages.
https://doi.org/10.1145/3378445

This material is based upon work supported by the National Science Foundation under Grant No. OAC-1642385. This material is based upon work supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics Program. 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 No. DE-NA-0003525.

Authors' addresses: G. Ballard, Department of Computer Science, Wake Forest University, Winston-Salem, NC, 27109; email: ballard@wfu.edu; A. Klinvex and T. G. Kolda, Sandia National Laboratories, Livermore, CA, 94551; emails: {amklinv, tgkolda}@sandia.gov.

0098-3500/2020/05-ART13 $15.00
https://doi.org/10.1145/3378445

1 INTRODUCTION

The convergence of ever-faster computational platforms and algorithmic advancements in scientific simulations have led to a data deluge—it is now possible to produce very fine-grained and detailed simulations, resulting in terabyte-sized or larger datasets. Consider the problem we discuss later on in this work: a simulation on a three-dimensional rectangular grid of size $500 \times 500 \times 500$, tracking 11 variables for 400 time steps. Even this modest-sized simulation yields 4 TB of data in double precision. With current computational platforms, this data cannot easily be stored, moved, visualized, or analyzed. Nevertheless, it is well known to simulation scientists that their massive datasets have extensive latent structure and are therefore highly compressible. The problem is how to discover the redundancies automatically.

Most compression methods focus on compressing *local* structure with very little loss in precision. Fout, Ma, and Ahrens [13] do multivariate volume block data reduction to take advantage of local multiway structure, achieving up to 70% compression. Likewise, Lindstrom's ZFP compresses data in local blocks [24] with 1.3–2.6× (23–61%) compression. More recently, Di and Cappello [12] report up 3–436× (66–99.8%) compression, but again focusing on local blocks.

Our method, in contrast, aims at detecting *global* structure in the data, yielding much higher compression ratios in exchange for potential loss in local accuracy. It does not process the data in blocks but rather considers the data in its entirety. Before we delve into the mathematical details, we note briefly that lossy compression need not replace the original data; rather, the compressed version is a thumbnail or preview of the full dataset, which may reside on long-term storage or be regenerated.

In this article, we consider the Tucker tensor decomposition [32], also known as the higher-order singular value decomposition (HOSVD) [11]. This is an effective tool for compression for many application domains [1, 5, 14, 16, 18, 34]. The idea is to consider the multiway structure of the data. For instance, the problem described above is a five-way object, and so we exploit this structure in the compression procedure. Specifically, each mode is compressed individually by determining a small number of vectors that span the *fibers* in that mode, fibers being the vectors that are the higher-order analogue of matrix rows and columns. We aim to compress an N-way tensor of size $I_0 \times I_1 \times \cdots \times I_{N-1}$ to size $R_0 \times R_1 \times \cdots \times R_{N-1}$ by computing R_n vectors for each mode n that approximately span the range of the mode-n fibers. We call R_n the *rank* of mode n. The ranks are typically selected to retain a specified relative accuracy ϵ, i.e., if $\mathcal{X}$ is the data tensor and $\hat{\mathcal{X}}$ is the reconstruction from the compressed representation, then we choose the ranks such that

$$\|\mathcal{X} - \hat{\mathcal{X}}\| \leq \epsilon \|\mathcal{X}\|. \tag{1}$$

Choosing $R_n = I_n$ yields perfect reconstruction, so viable choices for the ranks always exist. If we assume $I_n = I$ and $R_n = R$ for all n, for sake of exposition, then the storage is reduced from I^N to $R^N + NIR$, and so the compression ratio is $\approx (I/R)^N$, which is the size of the original data divided by the size of the compressed representation.

In this article, we develop a parallel implementation of the sequentially truncated HOSVD (ST-HOSVD) [33]. This article builds on past work by Austin, Ballard, and Kolda [2], which showed initial results for parallel versions of ST-HOSVD as well as the higher-order orthogonal iteration (HOOI). The code is available at https://gitlab.com/tensors/TuckerMPI. Our contributions are as follows:

(1) We describe the details of the TuckerMPI software that were not provided in Reference [2], including global data distributions (even in the case when a tensor dimension is not divisible by the number of processors in that dimension), local data layouts, and sequential and parallel algorithms.

(2) We present a new and improved kernel for computing the Gram matrix in ST-HOSVD, which is faster and less sensitive to the parallel data layout than the version in Reference [2].
(3) We present more extensive compression results for terabyte-scale data by compressing data sets that are an order of magnitude larger than those in Reference [2]. Specifically, we show that we can compress 4.5 and 6.7 TB datasets by 2–5 orders of magnitude in O(10–100) seconds, less time then reading the data from the parallel file system.
(4) An advantage of the Tucker decomposition that was mentioned but not implemented for Reference [2] is that we can reconstruct just a portion of the full tensor. Here, we present an efficient method for partial reconstruction, taking care not to create any object that is larger than the compressed or reconstructed subtensor so that it can be run on a workstation rather than a parallel system. We give experimental results to showcase its efficiency.

2 NOTATION AND MATHEMATICAL BACKGROUND

We use boldface Euler script letters to denote tensors ($\mathcal{X}$) and boldface uppercase letters to denote matrices ($\mathbf{X}$). We reserve the uppercase letters I, J, K, L, M, N, P, R to denote sizes and the corresponding lowercase letters i, j, k, l, m, n, p, r to denote the indices. We use zero-indexing throughout so that if i is the index corresponding to size I, then we have $i = 0, 1, \ldots, I-1$. For any size I, we use the notation $[I]$ to denote the set $\{0, 1, \ldots, I-1\}$.

2.1 Sizes

We define a few special quantities with respect to tuples of sizes, which are used in describing both tensor and processor grid sizes. For an object with dimensions $I_0 \times I_1 \times \cdots \times I_{N-1}$, we define the product of all its sizes (the total size) as

$$I^{\circledast} = \prod_{n \in [N]} I_n. \tag{2}$$

We further define some quantities that depend on the mode $n \in [N]$:

$$I_n^{\oslash} = \prod_{k \neq n} I_k = I^{\circledast}/I_n, \qquad \text{(product of all sizes except mode } n\text{)}, \tag{3}$$

$$I_n^{\circledcirc} = \prod_{k < n} I_k = I_n^{\oslash}/I_n^{\circledcirc}, \qquad \text{(product of sizes below mode } n\text{)}, \tag{4}$$

$$I_n^{\circledcirc} = \prod_{k > n} I_k = I_n^{\oslash}/I_n^{\circledcirc}, \qquad \text{(product of sizes above mode } n\text{)}. \tag{5}$$

For the edge cases, we say $I_0^{\circledcirc} = 1$ and $I_{N-1}^{\circledcirc} = 1$.

2.2 Tensor Operations

We discuss key tensor operations for computing the Tucker decomposition. Here, we assume an N-way tensor $\mathcal{X}$ of size $I_0 \times I_1 \times \cdots \times I_{N-1}$.

2.2.1 Tensor Unfolding. The mode-n unfolding of $\mathcal{X}$ rearranges the elements of the N-way tensor into a matrix, denoted $\mathbf{X}_{(n)}$, of size $I_n \times I_n^{\oslash}$. We map tensor element $(i_0, i_1, \ldots, i_{N-1})$ to matrix element (i_n, i'_n), where

$$i'_n = \sum_{k<n} i_k \cdot I_k^{\circledcirc} + \sum_{k>n} i_k \cdot I_k^{\circledcirc}/I_n.$$

See also Section 4.2 for examples of unfolded tensors and details of the organization in computer memory.

2.2.2 Tensor Norm. The norm of a tensor is the square root of the sum of squares of all the elements. This means it is equivalent to the Frobenious norm of any unfolding, i.e., $\|\mathcal{X}\| = \|\mathbf{X}_{(n)}\|_F$ for any $n \in [N]$.

2.2.3 Tensor Times Matrix. The *mode-n product* of $\mathcal{X}$ with a matrix $\mathbf{U}$ of size $J \times I_n$ is denoted $\mathcal{X} \times_n \mathbf{U}$, and the result is of size $I_0 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_{N-1}$. This can be expressed in terms of unfolded tensors, i.e.,

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{U} \quad \Leftrightarrow \quad \mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}.$$

This is also known as the tensor-times-matrix (TTM) product. For different modes, the order is irrelevant so that

$$\mathcal{X} \times_m \mathbf{U} \times_n \mathbf{V} = \mathcal{X} \times_n \mathbf{V} \times_m \mathbf{U} \text{ for } m \neq n.$$

In the same mode, order matters so that

$$\mathcal{X} \times_n \mathbf{U} \times_n \mathbf{V} = \mathcal{X} \times_n \mathbf{V}\mathbf{U}, \tag{6}$$

where $\mathbf{V}$ is $K \times J$.

3 REVIEW OF THE SEQUENTIALLY TRUNCATED HIGHER-ORDER SVD

In this section, we describe the Tucker decomposition and review the ST-HOSVD method that we parallelize to compute it. Before we do so, we introduce some notation and basic theory.

Let the N-way tensor $\mathcal{X}$ of size $I_0 \times I_1 \times \cdots \times I_{N-1}$ denote the data tensor to be compressed. The goal is to approximate $\mathcal{X}$ as

$$\mathcal{X} \approx \hat{\mathcal{X}} \equiv \mathcal{G} \times_0 \mathbf{U}_0 \times_1 \mathbf{U}_1 \cdots \times_{N-1} \mathbf{U}_{N-1}. \tag{7}$$

The tensor $\mathcal{G}$ is called the *core tensor*, and its size is denoted by $R_0 \times R_1 \times \cdots \times R_{N-1}$. Each *factor matrix* $\mathbf{U}_n$ is necessarily of size $I_n \times R_n$ for $n \in [N]$, and we assume throughout that the factor matrices have orthonormal columns. This means $\mathbf{U}_n^\mathsf{T}\mathbf{U}_n = \mathbf{I}$ (the $R_n \times R_n$ identity matrix). The storage of $\mathcal{X}$ is $I^{\circledast}$ as compared to the storage for $\hat{\mathcal{X}}$, which is $R^{\circledast} + \sum_{n\in[N]} R_n I_n$. If, for example, $I_n/R_n = 2$ for all n, then the compression ratio is $I^{\circledast}/R^{\circledast} \approx 2^N$.

We review a few relevant facts about Tucker per Reference [20, Section 4.2]. If the factor matrices are given, then it can be shown that the optimal $\mathcal{G}$ is

$$\mathcal{G} = \mathcal{X} \times_0 \mathbf{U}_0^\mathsf{T} \times_1 \mathbf{U}_1^\mathsf{T} \cdots \times_{N-1} \mathbf{U}_{N-1}^\mathsf{T}. \tag{8}$$

Substituting Equation (8) back into Equation (7) and using identity Equation (6), we have

$$\hat{\mathcal{X}} = \mathcal{X} \times_0 \mathbf{U}_0\mathbf{U}_0^\mathsf{T} \times_1 \mathbf{U}_1\mathbf{U}_1^\mathsf{T} \cdots \times_{N-1} \mathbf{U}_{N-1}\mathbf{U}_{N-1}^\mathsf{T}. \tag{9}$$

Thus, the factor matrices determine the projections (one per mode) of the original tensor down to the reduced space. It is easy to show that $\|\mathcal{X} - \hat{\mathcal{X}}\|^2 = \|\mathcal{X}\|^2 - \|\mathcal{G}\|^2$, which means that $\mathcal{G}$ retains most of the *mass* of $\mathcal{X}$ but is just represented with respect to a different basis.

It is convenient to make a few special definitions. Assume that the factor matrices are specified. Then define the *partial core* that is the result of applying $n+1$ factor matrices to $\mathcal{X}$:

$$\tilde{\mathcal{G}}_n = \mathcal{X} \times_0 \mathbf{U}_0^\mathsf{T} \cdots \times_n \mathbf{U}_n^\mathsf{T} \quad \text{for} \quad n \in [N]. \tag{10}$$

This means that $\tilde{\mathcal{G}}_n$ is only reduced in the first $(n+1)$ modes and so is of size $R_0 \times \cdots \times R_n \times I_{n+1} \cdots \times I_{N-1}$. By definition, $\mathcal{G} = \tilde{\mathcal{G}}_{N-1}$. Likewise, we can define the *incremental approximation* using the partial core to be

$$\tilde{\mathcal{X}}_n = \tilde{\mathcal{G}}_n \times_0 \mathbf{U}_0 \cdots \times_n \mathbf{U}_n = \mathcal{X} \times_0 \mathbf{U}_0\mathbf{U}_0^\mathsf{T} \cdots \times_n \mathbf{U}_n\mathbf{U}_n^\mathsf{T} \quad \text{for} \quad n \in [N]. \tag{11}$$

By definition, $\hat{\mathcal{X}} = \tilde{\mathcal{X}}_{N-1}$.

3.1 Key Theorem

The following theorem categorizes the error in terms of the incremental approximations and orthogonal projections based on the factor matrices. We refer the reader to Reference [33] for the proof. This theorem is key to understanding how to pick the factor matrices in the ST-HOSVD.

Theorem 3.1 [Vannieuwenhoven, Vandebril, and Meerbergen [33, Theorem 5.1]]. *Let $\mathcal{X}$ be a tensor of size $I_0 \times I_1 \times \cdots \times I_{N-1}$, which is approximated by $\hat{\mathcal{X}}$ as defined in Equation (9), in which the factor matrices $\mathbf{U}_n$ of size $I_n \times R_n$ have orthonormal columns. The approximation error is then given by*

$$\left\| \mathcal{X} - \hat{\mathcal{X}} \right\|^2 = \left\| \mathcal{X} \times_0 \left(\mathbf{I} - \mathbf{U}_0 \mathbf{U}_0^\mathsf{T} \right) \right\|^2 + \left\| \tilde{\mathcal{X}}_0 \times_1 \left(\mathbf{I} - \mathbf{U}_1 \mathbf{U}_1^\mathsf{T} \right) \right\|^2 + \cdots + \left\| \tilde{\mathcal{X}}_{N-2} \times_{N-1} \left(\mathbf{I} - \mathbf{U}_{N-1} \mathbf{U}_{N-1}^\mathsf{T} \right) \right\|^2, \quad (12)$$

where $\tilde{\mathcal{X}}_n$ is as defined in Equation (11). Additionally, the error is bounded by

$$\left\| \mathcal{X} - \hat{\mathcal{X}} \right\|^2 \leq \sum_{n \in [N]} \left\| \mathcal{X} \times_n \left(\mathbf{I} - \mathbf{U}_n \mathbf{U}_n^\mathsf{T} \right) \right\|^2. \quad (13)$$

3.2 HOSVD Method

The bound in Equation (13) from Theorem 3.1 is key to understanding the HOSVD method, originally known as Tucker1 [11, 32]. Suppose the goal is to find a Tucker decomposition of the form in Equation (7) with relative error no greater then ϵ, i.e.,

$$\frac{\|\mathcal{X} - \hat{\mathcal{X}}\|}{\|\mathcal{X}\|} \leq \epsilon. \quad (14)$$

Then the idea is as follows. Let the eigenvalue decomposition of the Gram matrix $\mathbf{S}$ of the mode-n unfolding be given by

$$\mathbf{S} \equiv \mathbf{X}_{(n)} \mathbf{X}_{(n)}^\mathsf{T} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^\mathsf{T}. \quad (15)$$

Here, $\mathbf{\Lambda} = \operatorname{diag}(\{ \lambda_1, \ldots, \lambda_{I_n} \})$ and $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_{I_n} \geq 0$ are the eigenvalues in descending order. The matrix $\mathbf{V}$ contains the corresponding eigenvectors. We choose $\mathbf{U}_n$ and R_n so that

$$\mathbf{U}_n = \mathbf{V}(:, 1{:}R_n) \quad \text{where} \quad R_n = \min_{R \in [I_n]} R \quad \text{subject to} \quad \sum_{i=R+1}^{I_n} \lambda_i \leq \epsilon^2 \|\mathcal{X}\|^2 / N. \quad (16)$$

This choice of $\mathbf{U}_n$ ensures that

$$\left\| \mathcal{X} \times_n \left(\mathbf{I} - \mathbf{U}_n \mathbf{U}_n^\mathsf{T} \right) \right\|^2 \leq \epsilon^2 \|\mathcal{X}\|^2 / N.$$

Repeating this procedure for each $n \in [N]$ ensures that the desired error bound Equation (14) holds. This discussion can be framed equivalently in terms of the leading left singular vectors of $\mathbf{X}_{(n)}$, which is how the HOSVD is usually described. However, we explicitly form the Gram matrices in our parallelized method, so this presentation is more convenient in our exposition.

3.3 ST-HOSVD

The HOSVD works with the full tensor at each step. The idea behind the sequentially truncated version introduced in Reference [33] is that we work with the partial cores. In other words, at step n, we compute the next factor matrix based on $\tilde{\mathcal{G}}_{n-1}$. This is possible because we can swap $\tilde{\mathcal{G}}_{n-1}$ for $\tilde{\mathcal{X}}_{n-1}$ in the summands in the error expression Equation (12) due to the following

equivalence:

$$\left\| \tilde{\mathcal{X}}_{n-1} \times_n \left(\mathbf{I} - \mathbf{U}_n \mathbf{U}_n^\mathsf{T} \right) \right\|^2 = \left\| \tilde{\mathcal{G}}_{n-1} \times_n \left(\mathbf{I} - \mathbf{U}_n \mathbf{U}_n^\mathsf{T} \right) \right\|^2. \tag{17}$$

Hence, substituting Equation (17) into Equation (12) then yields the following key corollary.

Corollary 3.2. *Let the conditions of Theorem 3.1 hold. Then*

$$\left\| \mathcal{X} - \hat{\mathcal{X}} \right\|^2 = \left\| \mathcal{X} \times_0 \left(\mathbf{I} - \mathbf{U}_0 \mathbf{U}_0^\mathsf{T} \right) \right\|^2 + \left\| \tilde{\mathcal{G}}_0 \times_1 \left(\mathbf{I} - \mathbf{U}_1 \mathbf{U}_1^\mathsf{T} \right) \right\|^2 + \cdots + \left\| \tilde{\mathcal{G}}_{N-2} \times_{N-1} \left(\mathbf{I} - \mathbf{U}_{N-1} \mathbf{U}_{N-1}^\mathsf{T} \right) \right\|^2, \tag{18}$$

where $\tilde{\mathcal{G}}_n$ is as defined in Equation (10).

Corollary 3.2 means that we can pick the $(n+1)$st factor matrix, $\mathbf{U}_n$, using the partial core $\tilde{\mathcal{G}}_{n-1}$. Choosing $\mathbf{U}_0$ is unchanged. For $n > 1$, however, we replace the eigenvalue problem in Equation (15) with

$$\mathbf{S} \equiv \mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^\mathsf{T} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^\mathsf{T} \quad \text{where} \quad \mathcal{Y} = \tilde{\mathcal{G}}_{n-1}. \tag{19}$$

We can still pick $\mathbf{U}_n$ according to Equation (16), but we are just working with the eigen decomposition of a different matrix. The ST-HOSVD algorithm is given in Algorithm 1.

ALGORITHM 1: Sequentially Truncated Higher-Order SVD (ST-HOSVD) [33]

function $(\mathcal{G}, \mathcal{U}) = \text{ST-HOSVD}(\mathcal{X}, \epsilon)$ ▷ $\mathcal{X}$ is data tensor of size $I_0 \times \cdots \times I_{N-1}$ and ϵ is desired accuracy
  $\mathcal{Y} \leftarrow \mathcal{X}$
  for $n = 0, 1, \ldots, N-1$ **do**
    $\mathbf{S} \leftarrow \mathbf{Y}_{(n)} \mathbf{Y}_{(n)}^\mathsf{T}$ ▷ S is referred to as the *Gram matrix*
    $(\boldsymbol{\lambda}, \mathbf{V}) \leftarrow \text{eig}(\mathbf{S})$ ▷ Eigenvalues in descending order
    $R_n \leftarrow \min \{ R \in [I_n] \mid \sum_{i=R+1}^{I_n} \lambda_i \leq \epsilon^2 \|\mathcal{X}\|^2 / N \}$ ▷ Choose R_n to satisfy the error bound
    $\mathbf{U}_n \leftarrow \mathbf{V}(:, 1\!:\!R_n)$ ▷ $\mathbf{U}_n$ is the leading R_n eigenvectors of S
    $\mathcal{Y} \leftarrow \mathcal{Y} \times_n \mathbf{U}_n^\mathsf{T}$ ▷ Sets $\mathcal{Y} = \tilde{\mathcal{G}}_n$, the partial core
  end for
  $\mathcal{G} \leftarrow \mathcal{Y}$ ▷ Core of size $R_0 \times R_1 \times \cdots \times R_{N-1}$
  $\mathcal{U} \leftarrow \{ \mathbf{U}_0, \ldots, \mathbf{U}_{N-1} \}$ ▷ Factor matrices with $\mathbf{U}_n$ of size $I_n \times R_n$ for all $n \in [N]$
  return $(\mathcal{G}, \mathcal{U})$ ▷ Result satisfies $\|\mathcal{X} - \hat{\mathcal{X}}\| / \|\mathcal{X}\| \leq \epsilon$ where $\hat{\mathcal{X}} = \mathcal{G} \times_0 \mathbf{U}_0 \cdots \times_{N-1} \mathbf{U}_{N-1}$
end function

3.4 Quasi-Optimality

Neither the HOSVD nor the ST-HOSVD yields an optimal rank-$(R_0, R_1, \ldots, R_{N-1})$ decomposition, and neither is necessarily more accurate than the other. The major advantage of the ST-HOSVD is in terms of computational cost: the cost of the Gram computation is decreased by a factor of I_n / R_n at each step. Both methods yield quasi-optimal results with error within a factor of the square root of the number of modes of the best approximation, as specified by the following theorem from [15].

Theorem 3.3 [Hackbusch [15, Theorems 6.9–10]]. *Let $\mathcal{X}$ be a tensor of size $I_0 \times I_1 \times \cdots \times I_{N-1}$, $\hat{\mathcal{X}}_{\text{HOSVD}}$ be the rank $R_0 \times R_1 \times \cdots \times R_{N-1}$ approximation computed by HOSVD, and $\hat{\mathcal{X}}_{\text{ST-HOSVD}}$ be the rank $R_0 \times R_1 \times \cdots \times R_{N-1}$ approximation computed by ST-HOSVD. Then*

$$\left\| \mathcal{X} - \hat{\mathcal{X}}_{\text{HOSVD}} \right\| \leq \sqrt{N} \left\| \mathcal{X} - \hat{\mathcal{X}}_{\text{opt}} \right\|$$

and

$$\left\| \mathcal{X} - \hat{\mathcal{X}}_{\text{ST-HOSVD}} \right\| \leq \sqrt{N} \left\| \mathcal{X} - \hat{\mathcal{X}}_{\text{opt}} \right\|,$$

where $\hat{\mathcal{X}}_{\text{opt}}$ is the optimal rank-$R_0 \times R_1 \times \cdots \times R_{N-1}$ approximation of $\mathcal{X}$.

3.5 Pre- and Post-Processing

Algorithms for computing Tucker models like ST-HOSVD (Algorithm 1) minimize the relative approximation error in a norm-wise sense. As a result, the relative component-wise errors are generally smaller for large entries in the data tensor than for small entries. If the data is not preprocessed, then the differences in magnitude are artifacts of the type of variable or unit of measurement rather than reflecting the importance of the data value.

To address these issues, we pre-process the original data using hyperslice-wise computations, where a single hyperslice corresponds to the values of a particular variable like pressure or temperature, all with the same physical units. These computations include gathering statistics, such as mean or maximum value, for each hyperslice of a particular mode and applying a single linear function to every element in a hyperslice to uniformly shift and/or scale the values. The vector of shifting and scaling values are stored so that the inverse operations can be applied in post-processing after the (partial) reconstruction process.

The hyperslice-wise statistics that can be collected include the mean, standard deviation, maximum, minimum, vector 1-norm, and vector 2-norm. Common preprocessing techniques are (a) shifting by the mean value (to center each hyperslice at 0) and scaling by the standard deviation (to impose a variance of 1) and (b) scaling by the maximum absolute value (to impose a range of -1 to 1). We refer to (a) as "standardization" and (b) as "max rescaling."

4 DATA LAYOUTS

In this section, we describe how tensors are stored in local memory and how they are distributed for parallel computation. Factor matrices are stored redundantly on every processor. We assume throughout that we have a generic N-way tensor $\mathcal{Y}$ of size $J_0 \times J_1 \times \cdots \times J_{N-1}$. WLOG, this $\mathcal{Y}$ tensor is the one that is updated in Algorithm 1, so each J_n is either R_n or I_n. We assume that the processors are logically arranged into an N-way Cartesian processor grid, with dimensions $P_0 \times P_1 \times \cdots \times P_{N-1}$, as in [2, 3, 29]. In the analysis, we assume $P_n \in \{1, \ldots, J_n\}$ for each $n \in [N]$. We use the size shorthands discussed in Section 2.1, e.g., $P^{\circledast}$ denotes the total number of processors.

4.1 Tensor Data Layout

Consider the *natural descending* format that generalizes the column-major matrix layout. That is, tensor element $(j_0, j_1, \ldots, j_{N-1})$ can be mapped to

$$j' = \text{idx2lin}\Big\{(j_0, j_1, \ldots, j_{N-1}),\ (J_0, J_1, \ldots, J_{N-1})\Big\} \equiv \sum_{n \in [N]} j_n \cdot J_n^{\circledcirc}, \tag{20}$$

which we refer to as the *linear index*. We can also define the inverse operation,

$$(j_0, j_1, \ldots, j_{N-1}) = \text{lin2idx}\Big\{j',\ (J_0, J_1, \ldots, J_{N-1})\Big\}.$$

Figure 1 shows a $3 \times 4 \times 3 \times 2$ tensor with each entry labeled by its linear index.

4.2 Unfolded Tensor Data Layout

If the tensor is stored in the natural descending format, then the data layout of each unfolding can be thought of as a set of contiguous submatrices, where each submatrix is stored in row-major ordering. In other words, there is no reason to do any data movement in memory to work with the unfolded matrix, because BLAS calls can operate on submatrices stored in row- or column-major order, as observed in References [2, 17, 22, 28]. The number and dimensions of these submatrices depend on the mode of the unfolding. For the nth-mode unfolding, the number of submatrices is $J_n^{\circledcirc}$, and each submatrix is of size $J_n \times J_n^{\circledcirc}$. Let us look at these for the $3 \times 4 \times 3 \times 2$ tensor in Figure 1. Recall that the values indicate the relative position in the natural descending format.

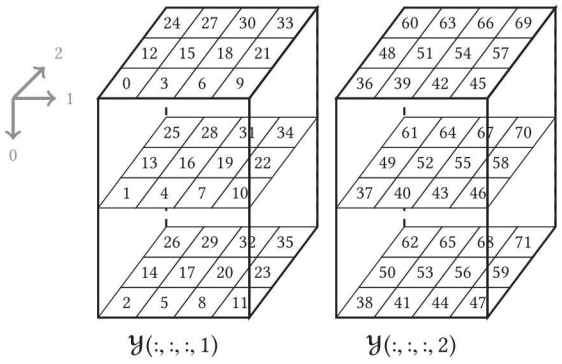


Fig. 1. A $3 \times 4 \times 3 \times 2$ tensor, where labels correspond to linear indices. The arrows show the direction for each mode, except the fourth mode, which is denoted explicitly in the labels at the bottom.

In the case $n = 0$, each submatrix has one column, which corresponds to a global column-major matrix, so it has 24 submatrices of size 3×1:

$$\mathbf{Y}_{(0)} = \left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} 0 & 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 & 27 & 30 & 33 & 36 & 39 & 42 & 45 & 48 & 51 & 54 & 57 & 60 & 63 & 66 & 69 \\ 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 & 25 & 28 & 31 & 34 & 37 & 40 & 43 & 46 & 49 & 52 & 55 & 58 & 61 & 64 & 67 & 70 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 & 26 & 29 & 32 & 35 & 38 & 41 & 44 & 47 & 50 & 53 & 56 & 59 & 62 & 65 & 68 & 71 \end{array}\right].$$

Note that this yields a special mode-0 case in the implementation of operations, because it is more efficient to treat the data as one column-major block than many blocks of vectors. The mode-1 unfolding has 6 row-major submatrices of size 4×3:

$$\mathbf{Y}_{(1)} = \left[\begin{array}{ccc|ccc|ccc|ccc|ccc|ccc} 0 & 1 & 2 & 12 & 13 & 14 & 24 & 25 & 26 & 36 & 37 & 38 & 48 & 49 & 50 & 60 & 61 & 62 \\ 3 & 4 & 5 & 15 & 16 & 17 & 27 & 28 & 29 & 39 & 40 & 41 & 51 & 52 & 53 & 63 & 64 & 65 \\ 6 & 7 & 8 & 18 & 19 & 20 & 30 & 31 & 32 & 42 & 43 & 44 & 54 & 55 & 56 & 66 & 67 & 68 \\ 9 & 10 & 11 & 21 & 22 & 23 & 33 & 34 & 35 & 45 & 46 & 47 & 57 & 58 & 59 & 69 & 70 & 71 \end{array}\right].$$

The mode-2 unfolding has two row-major submatrices of size 3×12:

$$\mathbf{Y}_{(2)} = \left[\begin{array}{cccccccccccc|cccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 & 60 & 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 \end{array}\right].$$

In the case $n = N - 1$, there is only one submatrix, which corresponds to a global row-major matrix, so we have one submatrix of size 2×36:

$$\mathbf{Y}_{(3)} = \left[\begin{array}{cccccccccccc|c|cccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \cdots & 26 & 27 & 28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & \cdots & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 \end{array}\right].$$

These layouts become important when we discuss the local operations in Section 5.

4.3 Global Tensor Data Distribution

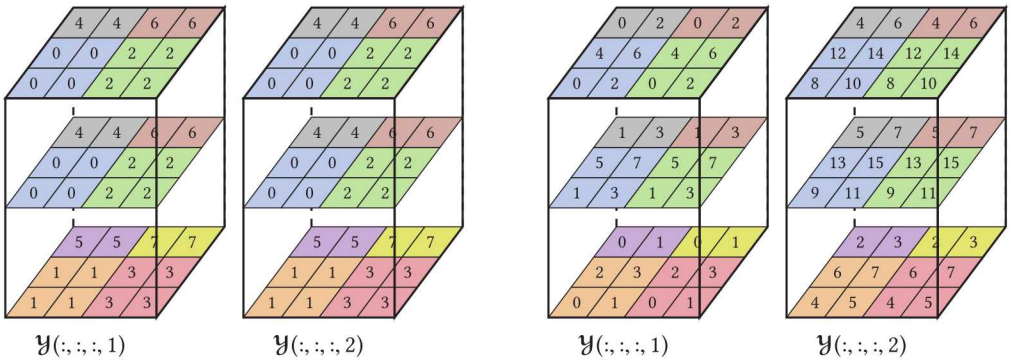
Now we consider how the tensor is distributed on the global processor grid. We let an overbar indicate local quantities with respect to a specific processor. If J_n is the global size of mode n, then $\bar{J}_n$ is the corresponding *local size* for the subtensor owned by a processor $(\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$.

Processor $(\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$ owns a block of size $\bar{J}_0 \times \bar{J}_1 \times \cdots \times \bar{J}_{N-1}$ where $\bar{J}_n$ is given by

$$\bar{J}_n = \mathrm{lsz}\{\bar{p}_n, J_n, P_n\} \equiv \begin{cases} (J_n \operatorname{div} P_n) + 1 & \text{if } \bar{p}_n < (J_n \bmod P_n) \\ (J_n \operatorname{div} P_n) & \text{if } \bar{p}_n \geq (J_n \bmod P_n) \end{cases}. \tag{21}$$

Table 1. Local Tensor Sizes for a $3 \times 4 \times 3 \times 2$ Tensor Distributed on a $2 \times 2 \times 2 \times 1$ Processor Grid, Demonstrating How Uneven Sizes Are Handled

Proc. ID (p_0, p_1, p_2, p_3)	Proc. Linear ID p'	Local Size $\bar{J}_0 \times \bar{J}_1 \times \bar{J}_2 \times \bar{J}_3$
$(0, 0, 0, 0)$	0	$2 \times 2 \times 2 \times 2$
$(1, 0, 0, 0)$	1	$1 \times 2 \times 2 \times 2$
$(0, 1, 0, 0)$	2	$2 \times 2 \times 2 \times 2$
$(1, 1, 0, 0)$	3	$1 \times 2 \times 2 \times 2$
$(0, 0, 1, 0)$	4	$2 \times 2 \times 1 \times 2$
$(1, 0, 1, 0)$	5	$1 \times 2 \times 1 \times 2$
$(0, 1, 1, 0)$	6	$2 \times 2 \times 1 \times 2$
$(1, 1, 1, 0)$	7	$1 \times 2 \times 1 \times 2$



(a) Processor Linear ID (b) Local linear indices

Fig. 2. The $3 \times 4 \times 3 \times 2$ tensor from Figure 1 distributed on a $2 \times 2 \times 2 \times 1$ processor grid. Each of the eight processors is represented by a different color.

Table 1 shows the size of each local subtensor for a tensor of size $3 \times 4 \times 3 \times 2$ distributed on a processor grid of size $2 \times 2 \times 2 \times 1$. Table 1 also indicates the *processor linear index*, which is defined analogously to Equation (20):

$$\bar{p}' = \text{idx2lin}\Big\{(\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1}), (P_0, P_1, \ldots, P_{N-1})\Big\} = \sum_{n \in [N]} \bar{p}_n \cdot P_n^{\circledcirc} \quad \in [P^{\circledast}].$$

The tensor element with global index $(j_0, j_1, \ldots, j_{N-1})$ is mapped to processor $(\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$ where $\bar{p}_n$ is given by

$$\bar{p}_n = \text{glb2prc}\{j_n, J_n, P_n\} \equiv \begin{cases} \left\lfloor \frac{j_n}{D_n + 1} \right\rfloor & \text{if } j_n < M_n(D_n + 1), \\ \left\lfloor \frac{j_n - M_n(D_n+1)}{D_n} \right\rfloor + M_n & \text{otherwise,} \end{cases}$$
$$\text{where} \quad D_n = (J_n \text{ div } P_n) \quad \text{and} \quad M_n = (J_n \bmod P_n). \quad (22)$$

Figure 2(a) shows the mapping of elements to processors for the tensor in Figure 1 using a processor grid of size $2 \times 2 \times 2 \times 1$.

We define the set of mode-n global indices mapped to processor $\bar{p}_n$ to be

$$\bar{\mathcal{J}}_n = \text{prcmap}\{\bar{p}_n, J_n, P_n\} \equiv \{\bar{p}_n D_n + \min\{\bar{p}_n, M_n\}, \ldots, (\bar{p}_n+1)D_n + \min\{\bar{p}_n+1, M_n\} - 1\}, \quad \text{where} \quad D_n = (J_n \text{ div } P_n) \quad \text{and} \quad M_n = (J_n \bmod P_n). \tag{23}$$

The per-mode assignments can be combined to get all the indices assigned to a specific processor. One interesting note is that elements that are contiguous in the global natural descending linear index are not necessarily mapped to the same processor. For instance, in the example in Figure 2, consider that processor $\bar{p}' = 2$ owns the following global linear indices:

$$\bar{\mathcal{J}} = \{6, 7, 9, 10, 18, 19, 21, 22\},$$

which is not contiguous in the global linear ordering.

The local index is denoted $(\bar{j}_0, \bar{j}_1, \ldots, \bar{j}_{N-1})$ where j_n is given by

$$\bar{j}_n = \text{gbl2lcl}\{j_n, \bar{p}_n, J_n, P_n\} \equiv j_n - (\bar{p}_n(J_n \text{ div } P_n) + \min\{\bar{p}_n, (J_n \bmod P_n)\}). \tag{24}$$

Locally, the subtensors are stored contiguously according to the natural descending index and in the same order as their global linear index. Figure 2(b) shows the local linear indices for the example in Figure 2(a). This equation can be reversed to find the global index $(j_0, j_1, \ldots, j_{N-1})$ where j_n is given by

$$j_n = \text{lcl2gbl}\{\bar{j}_n, \bar{p}_n, J_n, P_n\} \equiv \bar{j}_n + (\bar{p}_n(J_n \text{ div } P_n) + \min\{\bar{p}_n, (J_n \bmod P_n)\}). \tag{25}$$

4.4 Global Unfolding Data Distribution

In the parallel case, the unfolded tensor is distributed among the processors in a block fashion. For instance, the mode-1 unfolding is here color-coded to the processor (using the same colors as in Figure 2):

$$\mathbf{Y}_{(1)} = \left[\begin{array}{cc|cc|cc|cc|cc|cc|cc|cc|cc} 0 & 1 & 2 & 12 & 13 & 14 & 24 & 25 & 26 & 36 & 37 & 38 & 48 & 49 & 50 & 60 & 61 & 62 \\ 3 & 4 & 5 & 15 & 16 & 17 & 27 & 28 & 29 & 39 & 40 & 41 & 51 & 52 & 53 & 63 & 64 & 65 \\ 6 & 7 & 8 & 18 & 19 & 20 & 30 & 31 & 32 & 42 & 43 & 44 & 54 & 55 & 56 & 66 & 67 & 68 \\ 9 & 10 & 11 & 21 & 22 & 23 & 33 & 34 & 35 & 45 & 46 & 47 & 57 & 58 & 59 & 69 & 70 & 71 \end{array}\right].$$

Note that a single MPI process may own multiple contiguous pieces of the unfolded tensor. In particular, the data distribution of the unfoldings are not a standard distribution such as blocked or block-cyclic. However, the distribution is still a blocked matrix distribution, equivalent to unfolding the local tensor. Here, we show the *local* linear index of each entry:

$$\mathbf{Y}_{(1)} = \left[\begin{array}{cc|cc|cc|cc|cc|cc|cc|cc|cc} 0 & 1 & 0 & 4 & 5 & 2 & 0 & 1 & 0 & 8 & 9 & 4 & 12 & 13 & 6 & 4 & 5 & 2 \\ 2 & 3 & 1 & 6 & 7 & 3 & 2 & 3 & 1 & 10 & 11 & 5 & 14 & 15 & 7 & 6 & 7 & 3 \\ 0 & 1 & 0 & 4 & 5 & 2 & 0 & 1 & 0 & 8 & 9 & 4 & 12 & 13 & 6 & 4 & 5 & 2 \\ 2 & 3 & 1 & 6 & 7 & 3 & 2 & 3 & 1 & 10 & 11 & 5 & 14 & 15 & 7 & 6 & 7 & 3 \end{array}\right].$$

Notice that they are contiguous for each processor. This means that we can work with the locally unfolded tensor for the local operations in distributed computations.

4.5 Processor Fibers

Just as for the data tensor, there is a corresponding processor grid "unfolding" that corresponds to the tensor unfoldings. Instead of an N-way processor grid, we can think of the processors as

rearranged into a 2-way grid. So, if we revisit the example in the previous subsection, then the mode-1 processor unfolding for the $2 \times 2 \times 2 \times 1$ processor grid is

(0, 0, 0, 0)	(1, 0, 0, 0)	(0, 0, 1, 0)	(1, 0, 1, 0)
(0, 1, 0, 0)	(1, 1, 0, 0)	(0, 1, 1, 0)	(1, 1, 1, 0)

.

We see the processors in the same column match in every index except the mode-1 index, since this is the mode-1 unfolding. We see later that certain operations require collective communications within each processor column in this unfolding. We refer to these column groups of processors as *mode-n processor fibers*. We refer to row groups of processors as *mode-n processor slices*.

5 LOCAL KERNELS

In Algorithm 1, there are three key kernels: Gram of the mode-n unfolded tensors in line 4, the $I_n \times I_n$ eigen problem in line 7, and tensor-times-matrix to shrink in mode n in line 8. Here, we explain the serial implementation, which is also used for the local computations in the parallelized version. We explain all the functions with respect to a generic tensor $\mathcal{Y}$ of size $J_0 \times J_1 \times \cdots \times J_{N-1}$.

5.1 Local Gram

We want to compute $\mathbf{S} = \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^\mathsf{T}$ so $\mathbf{S}$ will be of size $J_n \times J_n$. The arithmetic cost is $O(J^{\circledast} J_n)$. Although the computation is mathematically one matrix operation, the data layout of the unfolding of the input tensor prevents a single call to the `syrk` BLAS subroutine, which requires strided row- or column-major ordering. Thus, the algorithm works on the native row-major submatrices as described in Section 4.2, except in the case of $n = 0$ where the algorithm works on the native column-major matrices. For $n > 0$, $\mathbf{Y}_{(n)}$ has $J_n^{\circledcirc}$ submatrices of size $J_n \times J_n^{\circledcirc}$. We denote the jth block-column submatrix as $\mathbf{Y}_{(n)}[j] \equiv \mathbf{Y}_{(n)}(:, j \cdot J_n^{\circledcirc} : j \cdot J_n^{\circledcirc} + J_n^{\circledcirc} - 1)$, which is stored in row-major form. The algorithm is shown in Algorithm 2.

ALGORITHM 2: Local Gram (compute gram matrix of unfolding in mode n)

```
function S = GRAM(Y, n)
    if n = 0 then
        S = Y_(n) Y_(n)^T                          ▷ Call to syrk, Y_(n) in column-major format
    else
        S ← 0
        for j ∈ [ J_n^⊚ ] do
            S ← S + Y_(n)[j] Y_(n)[j]^T            ▷ Call to syrk, Y_(n)[j] denotes jth block column in row-major format
        end for
    end if
end function
```

5.2 Local Eigenvalue Decomposition

We need to compute the eigenvalue decomposition of a $J_n \times J_n$ matrix $\mathbf{S}$. The computational cost is $O(J_n^3)$. We use the the LAPACK direct eigenvalue computation routine `syev` to compute all the eigenvalues and all the eigenvectors.

A couple of notes are in order. The cost of the full eigenvector decomposition is approximately $\frac{10}{3}J_n^3$. One alternative approach would be to compute the full set of eigenvalues at a cost of $\frac{4}{3}J_n^3$, and then compute only the leading eigenvectors with $O(J_n^2 R_n)$ flops. Iterative methods, such as subspace iteration, would also work well in this case. However, because this phase of computation has never been a bottleneck for our applications, we have not implemented these cheaper approaches. If J_n is relatively large compared to the product of the other dimensions, $J_n^{\oslash}$, then the eigenvalue decomposition may become a bottleneck, so we leave this as a topic for future work.

5.3 Local TTM

We want to compute the product $\mathcal{Z} = \mathcal{Y} \times_n \mathbf{V}$ where $\mathbf{V}$ is a matrix of size $K \times J_n$. Recall that the TTM operation is defined in Section 2.2.3. The arithmetic cost of TTM is $O(J^{\circledast} K)$.

As in the case of Gram, although TTM is mathematically a matrix multiplication ($\mathbf{Z}_{(n)} = \mathbf{V}\mathbf{Y}_{(n)}$), the data layouts of $\mathbf{Y}_{(n)}$ and $\mathbf{Z}_{(n)}$ prevent a single call to the gemm subroutine in BLAS, because BLAS requires strided row- or column-major access to the matrices. We use the same notation as for Gram, so that $\mathbf{Y}_{(n)}[j]$ is the jth block column of $\mathbf{Y}_{(n)}$ stored in row-major form. Likewise, $\mathbf{Z}_{(n)}$ is organized into row-major block column submatrices of size $K \times J_n^{\oslash}$, and the jth submatrix is denoted as $\mathbf{Z}_{(n)}[j]$. In the case of $n = 0$, both $\mathbf{Y}_{(n)}$ and $\mathbf{Z}_{(n)}$ are natively in column-major mode. The algorithm is shown in Algorithm 3.

ALGORITHM 3: Local TTM (tensor-times-matrix in mode n)

```
function Z = TTM(Y, n, V)          ▷ Y is tensor of size J_0 × ··· × J_{N−1} and V is matrix of size K × J_n
    if n = 0 then
        Z_(n) = V Y_(n)            ▷ Call to gemm, Z_(n) and Y_(n) in column-major format
    else
        for j ∈ [ J_n^⊘ ] do
            Z_(n)[j] ← V Y_(n)[j]  ▷ Call to gemm, Z_(n)[j] and Y_(n)[j] denote jth block column in row-major format
        end for
    end if
end function
```

6 DISTRIBUTED KERNELS

To parallelize STHOSVD (Algorithm 1), we use parallel algorithms for the two key kernels: Gram and TTM. All other computations are performed redundantly on each processor, including the eigen decomposition.

Throughout this section, we consider a generic tensor $\mathcal{Y}$ of size $J_0 \times J_1 \times \cdots \times J_{N-1}$, distributed on a processor grid of size $P_0 \times P_1 \times \cdots \times P_{N-1}$, as described in Section 4.3. We use an overbar to denote the *local* portions/versions/sizes of distributed variables. For instance, $\bar{\mathcal{Y}}$ denotes the local portion of $\mathcal{Y}$, and $\bar{J}_0 \times \bar{J}_1 \times \cdots \times \bar{J}_{N-1}$ is its size. Note that local quantities, including their sizes, may vary from processor to processor.

6.1 Assumptions on Collective Communication

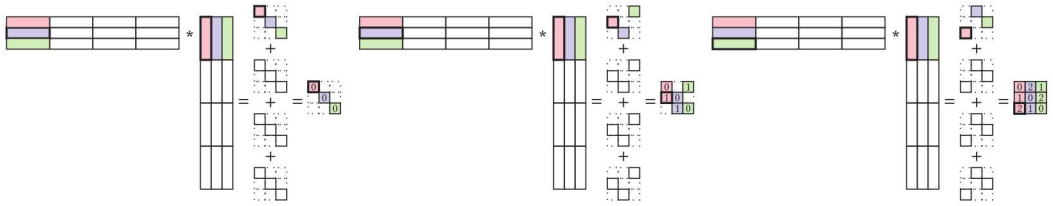
To analyze our algorithms, we use the MPI model of distributed-memory parallel computation. We assume a fully connected network of P processors and therefore do not model network contention. For simplicity of discussion in our analysis, we assume that P is a power of two and that optimal collective communication algorithms are used. The cost to *send/receive* a message of size W words between any two processors is $\alpha + W\beta$, where α is the latency cost and β is the per-word transfer cost. The cost of the collective communications used in this work are given in Table 2, with γ corresponding to the time per floating point operation (flop). For simplicity of presentation, we will ignore the flop cost of the reductions in later analysis, as they are typically dominated by the bandwidth costs. For more discussion of the model and descriptions of efficient collectives, see References [8, 31].

6.2 Parallel Gram

The goal here is to compute $\mathbf{S} = \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^{\mathsf{T}}$ where $\mathcal{Y}$ is distributed as described in Section 4.3. Each processor owns $\bar{\mathcal{Y}}$ (local portion of $\mathcal{Y}$) of size $\bar{J}_0 \times \bar{J}_1 \times \cdots \times \bar{J}_{N-1}$ where $\bar{J}_n = \mathrm{lsz}\{\bar{p}_n, J_n, P_n\}$ and

Table 2. Communication Costs in MPI Model, Where W Is the Local Input Data Size and P Is the Number of Processors

Send/Receive	$\alpha + \beta W$
Reduce/All-Reduce	$2\alpha \log P + (2\beta + \gamma)\frac{P-1}{P} W$
Reduce-Scatter	$\alpha \log P + (\beta + \gamma)\frac{P-1}{P} W$
All-to-All	$\alpha(P-1) + \beta\frac{P-1}{P} W$



(a) First matrix multiplication (b) Second matrix multiplication (c) Final matrix multiplication

Fig. 3. Round-robin variant of parallel Gram matrix computation with $P_n = 3$ [2].

the mode-n indices correspond to the global indices in $\bar{\mathcal{J}}_n = \text{prcmap}\{\bar{p}_n, J_n, P_n\}$. In the end, each processor will own the entirety of $\mathbf{S}$, which is only of size $J_n \times J_n$.

Austin et al. [2] previously proposed a parallel Gram as illustrated in Figure 3. In this algorithm, each processor fiber, which owns a column block of $\mathbf{Y}_{(n)}$, computes a contribution $\mathbf{V}$ to the result $\mathbf{S}$; $\mathbf{V}$ is distributed across the processor fiber. To compute $\mathbf{V}$ within the fiber, the processors rotate their tensor data around in a round-robin fashion, and at each step, each processor computes a $\bar{J}_n \times \bar{J}_n$ block of $\mathbf{V}$. To compute $\mathbf{S}$, the processors perform an All-Reduce across processor slices, so that $\mathbf{S}$ is redundantly stored on every processor fiber but distributed across the processors within the fiber. We refer to the older version as the *round-robin* variant.

We propose a new version of parallel Gram that is nearly always faster, which we refer to as the *redistribution* variant. Algorithm 4 presents the new parallel algorithm for computing the Gram matrix corresponding to a particular mode. The algorithm assumes that the input tensor $\mathcal{Y}$ is block distributed; at the end of the algorithm, the output matrix $\mathbf{S} = \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^{\mathsf{T}}$ is redundantly stored on every processor. The parallel Gram algorithm presented here differs from the one described in Reference [2], depicted in Figure 3. When the number of processors in the specified mode P_n is 1, the algorithms are identical. However, when $P_n > 1$, the previous algorithm uses $P_n - 1$ communication steps within the processor fiber and then communicates across the processor slice. As we describe in more detail below, Algorithm 4 works by first redistributing the tensor data with an All-to-All collective within the processor fiber and then performing a reduction across all processors. We compare the communication costs of the two algorithms at the end of the section.

As in the case of Algorithm 5, line 2 and 3 of Algorithm 4 define the processor's index and the set of processors within the processor's nth-mode processor fiber. The goal of line 5 is to obtain a 1D parallel distribution of the tensor. With this distribution, all processors can perform Gram computations with their local data, and the only remaining communication is to sum up the results over all processors. Figure 4 illustrates the redistribution. Because each processor fiber stores a set of columns of the matricized tensor (distributed row-wise), the redistribution occurs within each fiber independently and converts the row-wise distribution to a column-wise distribution.

Again, a key issue in the implementation is the need to pack and unpack buffers for the All-to-All collective, depending on n. The input buffer must be arranged so that the data to be received

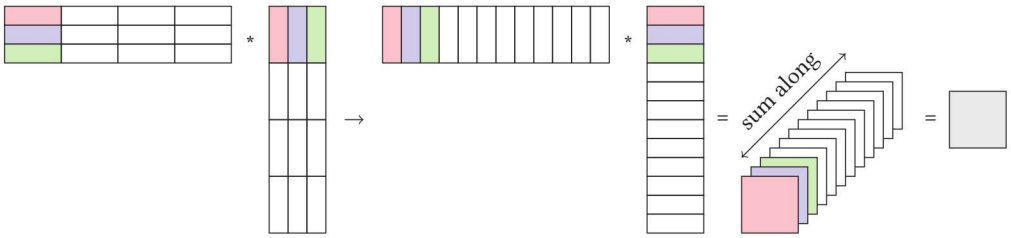


Fig. 4. New redistribution variant of parallel Gram matrix computation.

by each processor is stored contiguously, and the result buffer is ordered so that the data each processor sent is contiguous. In the case $n = 0$, the local matricized tensor is column-major, so no packing is necessary. After the All-to-All, the result buffer consists of P_n contiguous column-major blocks, where each column has length $\bar{J}_n$. The unpacking consists of collecting the P_n chunks of each mode-n fiber (of length J_n) into contiguous columns. In the case $0 < n < N-1$, the local matricized tensor is stored as an array of row-major submatrices. The packing consists of converting every row-major submatrix to column-major ordering, which makes every local mode-n fiber contiguous. The unpacking is the same as for $n = 0$: for each mode-n fiber, the P_n chunks are made contiguous. We note that the local ordering of the columns is not consistent with a matricized tensor format, but the Gram computation is invariant under column permutations, so this does not affect the result. In the case $n = N-1$, the local matricized tensor is row-major. Instead of converting the entire matrix to column-major, each contiguous row is broken up into P_n chunks, and the row ordering is maintained. After the All-to-All, no unpacking is necessary, because the result buffer is a (vertical) concatenation of row-major matrices and is thus also row-major.

ALGORITHM 4: Parallel Gram (redistribution variant)

1: **function** $\mathbf{S} = \text{PAR\_GRAM}(\bar{\mathcal{Y}}, n, (P_0, P_1, \ldots, P_{N-1}))$
2: `myProcID` $\leftarrow (\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$
3: `myProcFiber` $\leftarrow (\bar{p}_0, \ldots, \bar{p}_{n-1}, :, \bar{p}_{n+1}, \ldots, \bar{p}_{N-1})$ ▷ Processor group of size P_n
4: `allProcs` $\leftarrow$ all $P^{\circledast}$ processor ids
5: $\bar{\mathbf{Z}} = \text{AllToAll}(\overline{\mathbf{Y}}_{(n)}, \texttt{myProcFiber})$ ▷ Change from within-fiber block row to block column distribution
6: $\bar{\mathbf{W}} = \bar{\mathbf{Z}}\bar{\mathbf{Z}}^{\mathsf{T}}$ ▷ Local computation
7: $\mathbf{S} = \text{AllReduce}(\bar{\mathbf{W}}, \texttt{allProcs})$ ▷ All processors own a copy of the same matrix
8: **end function**

After obtaining a 1D distribution of the matricized tensor, each processor performs its local computation in line 6, computing the Gram matrix associated with its subset of the columns of the matricized tensor. To compute the Gram matrix of the entire matricized tensor, all processors participate in an All-Reduce (line 7), which results in (symmetric) $\mathbf{S}$ being redundantly stored on all processors.

The cost of Algorithm 4 is

$$C_{\text{GRAM}} = \underbrace{(P_n - 1)\left(\alpha + \beta\frac{\bar{J}^{\circledast}}{P_n}\right)}_{\text{line 5}} + \underbrace{\gamma J_n \bar{J}^{\circledast}}_{\text{line 6}} + \underbrace{2\alpha \log P^{\circledast} + \beta J_n^2}_{\text{line 7}}$$
$$= \alpha \cdot O(P_n) + \beta \cdot O\left(\bar{J}^{\circledast} + J_n^2\right) + \gamma \cdot O\left(J_n \bar{J}^{\circledast}\right).$$

For comparison, the cost of the previous parallel Gram algorithm [2] is

$$\alpha \cdot O\left(P_n\right) + \beta \cdot O\left(P_n \bar{J}^{\circledast} + \frac{J_n^2}{P_n}\right) + \gamma \cdot O\left(J_n \bar{J}^{\circledast}\right),$$

where the major difference is in the bandwidth cost. In particular, Algorithm 4 is more efficient when $P_n \bar{J}^{\circledast} \geq J_n^2$. In this case, both bandwidth cost terms of the new algorithm are smaller than the first term of the old algorithm. (If the converse is true, then both bandwidth cost terms of the old algorithm are smaller than the second term of the new algorithm.) We expect the inequality to hold in nearly all cases (except for extreme strong-scaling cases), because $\bar{J}^{\circledast}$ itself is the size of the local input tensor, while J_n^2 is the size of the nth-mode Gram matrix.

The temporary memory requirement of Algorithm 4 (besides the input and output data) is $2\bar{J}^{\circledast} + J_n^2$ words. Two temporary arrays of the size of the local tensor are required for the send and receive buffers in the All-to-All (local data has to be reordered to match the requirements of the input buffer), and J_n^2 space is required for $\mathbf{W}$ (All-Reduce requires separate input and output buffers). For comparison, the temporary memory requirement of the previous Gram algorithm is $\bar{J}^{\circledast} + \bar{J}_n^2$.

6.3 Parallel TTM

We consider the problem of computing the TTM $\mathcal{Z} = \mathcal{Y} \times_n \mathbf{V}$ where the input $\mathcal{Y}$ of size $J_0 \times J_1 \times \cdots \times J_{N-1}$ is block distributed, the matrix $\mathbf{V}$ of size $K_n \times J_n$ with $K_n < J_n$ is stored redundantly on every processor, and the output tensor $\mathcal{Z}$ of size $J_1 \times \cdots \times J_{n-1} \times K_n \times J_{n+1} \times \cdots \times J_N$ will be block distributed. Specifically, the data is distributed so that processor $(\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$ owns the following:

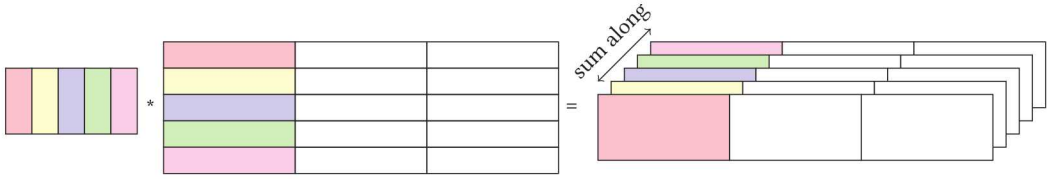
- $\bar{\mathcal{Y}}$ (local portion of $\mathcal{Y}$) is of size $\bar{J}_0 \times \bar{J}_1 \times \cdots \times \bar{J}_{N-1}$ where $\bar{J}_n = \mathrm{lsz}\{\bar{p}_n, J_n, P_n\}$ and the mode-n indices correspond to the global indices in $\bar{\mathcal{J}}_n = \mathrm{prcmap}\{\bar{p}_n, J_n, P_n\}$.
- $\bar{\mathcal{Z}}$ (local portion of $\mathcal{Z}$) is of size $\bar{J}_1 \times \cdots \times \bar{J}_{n-1} \times \bar{K}_n \times \bar{J}_{n+1} \times \cdots \times \bar{J}_N$ where $\bar{K}_n = \mathrm{lsz}\{\bar{p}_n, K_n, P_n\}$; it is distributed the same as $\bar{\mathcal{Y}}$ except that the mode-n indices correspond to the global indices in $\bar{\mathcal{K}}_n = \mathrm{prcmap}\{\bar{p}_n, K_n, P_n\}$.
- $\bar{\mathbf{V}}$ is of size $K_n \times \bar{J}_n$ and is the submatrix of $\mathbf{V}$ corresponding to the columns in the set $\bar{\mathcal{J}}_n$. (Although every processor owns all of $\mathbf{V}$, the distributed TTM only needs $\bar{\mathbf{V}}$.)

Algorithm 5 presents the parallel algorithm for distributed TTM. This is the same algorithm as previously presented by Austin et al. [2], but here we provide additional implementation details and analysis.

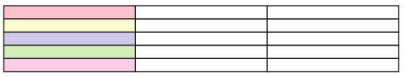
The computation reduces to a large matrix-matrix product, i.e., $\mathbf{Z}_{(n)} = \mathbf{V}\mathbf{Y}_{(n)}$. If $\mathbf{Y}_{(n)}$ is partitioned into column blocks, then the computation can be computed separately in each block. Therefore, since each mode-n processor fiber owns a separate column block of $\mathbf{Y}_{(n)}$, they are independent. For this reason, line 3 defines the local processor fiber, and all communication for that processor occurs within the P_n nodes of that fiber. There are $P_n^{\oslash}$ independent column fibers.

The algorithm chooses between two methods based on the size of K_n. The decision is based on the size of the intermediate quantities that are computed. The choice of method ensures that the temporary memory never exceeds the memory of $\bar{\mathcal{Y}}$.

In the *Reduce-Scatter variant*, each processor computes the local matrix-matrix product, $\bar{\mathbf{W}} = \bar{\mathbf{V}}\bar{\mathbf{Y}}_{(n)}$, and then sums and distributes the result using a Reduce-Scatter. The temporary object $\bar{\mathbf{W}}$ is of size $K_n \times \bar{J}_n^{\oslash}$, which may be too large to store on the processor. We assume that the processor has enough extra memory to store something the same size as $\bar{\mathbf{Y}}_{(n)}$ (i.e., $\bar{\mathcal{Y}}$), which is of size $\bar{J}_n \times \bar{J}_n^{\oslash}$, and so we use this variant if $K_n \leq \lfloor J_n / P_n \rfloor \approx \bar{J}_n$. This process is demonstrated in Figure 5 where five processors are shaded (in the first processor fiber) to understand that data ownership of each processor.



(a) Local matrix multiplication produces a local product that needs to be summed and distributed



(b) The Reduce-Scatter collective produces final result, with communication only within the processor fiber

Fig. 5. Reduce-Scatter variant of TTM, used if temporary storage does not exceed original data. The data belonging to five processors (in the same processor fiber) is color-coded for tracking.

ALGORITHM 5: Parallel TTM [2]

1: **function** $\bar{\mathcal{Z}} = \text{PAR\_TTM}(\bar{\mathcal{Y}}, n, \bar{\mathbf{V}}, (P_0, P_1, \ldots, P_{N-1}))$ ▷ $\bar{\mathcal{Y}}$ is local portion of the tensor and $\bar{\mathbf{V}}$ is column block of $\mathbf{V}$
2: myProcID $\leftarrow (\bar{p}_0, \bar{p}_1, \ldots, \bar{p}_{N-1})$
3: myProcFiber $\leftarrow (\bar{p}_0, \ldots, \bar{p}_{n-1}, :, \bar{p}_{n+1}, \ldots, \bar{p}_{N-1})$ ▷ Processor group of size P_n
4: **if** $K_n \leq \lfloor J_n / P_n \rfloor$ **then** ▷ Reduce-scatter variant
5: $\bar{\mathcal{W}} = \text{TTM}(\bar{\mathcal{Y}}, n, \bar{\mathbf{V}})$
6: $\bar{\mathcal{Z}} = \text{ReduceScatter}(\bar{\mathcal{W}}, \text{myProcFiber})$
7: **else** ▷ Multiple-reduction variant
8: Partition $\bar{\mathbf{V}}$ into row blocks $\bar{\mathbf{V}}[\ell]$ of size $\hat{K}_\ell = \text{lsz}\{\ell, K_n, P_n\} \times \hat{J}_n$ for $\ell \in [P_n]$
9: **for** $\ell \in [P_n]$ **do**
10: $\bar{\mathcal{W}} = \text{TTM}(\bar{\mathcal{Y}}, n, \bar{\mathbf{V}}[\ell])$
11: $\bar{\mathcal{Z}} = \text{Reduce}(\bar{\mathcal{W}}, \text{myProcFiber}, \ell)$ ▷ Root for Reduce is processor $(\bar{p}_0, \ldots, \bar{p}_{n-1}, \ell, \bar{p}_{n+1}, \ldots, \bar{p}_{N-1})$
12: **end for**
13: **end if**
14: **end function**

A key issue in the implementation is that the entries' ordering in $\bar{\mathcal{W}}$ is not correct for a Reduce-Scatter and so must be reorganized when the data is packed for the Reduce-Scatter to obtain the proper layout of $\bar{\mathcal{Z}}$ at the end of the call. The input buffer for Reduce-Scatter must be arranged so that contributions to each processor's result $\bar{\mathcal{Z}}$ are contiguous. The ordering of $\bar{\mathcal{W}}$ consists of $\bar{J}_n^{\oslash}$ row-major submatrices of dimension $K \times \bar{J}_n^{\oslash}$, and it must be reordered into P_n contiguous sub-blocks, each consisting of $\bar{J}_n^{\oslash}$ row-major submatrices of dimension $K/P_n \times \bar{J}_n^{\oslash}$ (assuming P_n divides K evenly). If $n = N - 1$ (the last dimension), then no reordering is necessary, and no unpacking is necessary after the Reduce-Scatter for any dimension.

In the *multiple-reduction variant*, the algorithm uses a blocked approach that involves P_n local TTMs and P_n collective communications. Here, each iteration computes the contribution to one processor's output (within the processor fiber—all processor fibers are working concurrently) and uses a Reduce collective to compute the sum across all processors in the fiber and store the result on the ℓth processor. The matrix $\bar{\mathbf{V}}$ is divided into block rows so that $\bar{\mathbf{V}}[\ell]$ owns the rows in the set prcmap$\{\bar{p}_n, \ell, K_n, P_n, P_n\}$. In this case, the dimensions of the temporary tensor $\bar{\mathcal{W}}$ in the ℓth iteration is bounded above by $\bar{J}_0 \times \cdots \times \lceil K_n / P_n \rceil \times \cdots \times \bar{J}_{N-1}$, which is essentially the same size as $\bar{\mathcal{Z}}$ (recall that $\hat{K}_\ell$ is within 1 of $\bar{K}_n$). This process is demonstrated in Figure 6, where we highlight the contribution to the 4th processor in the 1st column fiber.

The two variants perform the same number of flops and communicate almost the same amount of data (to within a factor of 2). The number of flops is $O(J^{\circledast} K_n / P^{\circledast})$, which is the cost of the local TTM(s).

Table 3. Asymptotic Costs of Algorithms for TTM and Gram with Respect to Mode n for a Tensor with Global Dimensions $J_0 \times J_1 \times \cdots \times J_{N-1}$ and Processor Grid with Dimensions $P_0 \times P_1 \times \cdots \times P_{N-1}$

	Flops	Bandwidth	Latency	Memory
TTM (Reduce-Scatter)	$\frac{J_n^{\circledast}}{P^{\circledast}}K$	$\frac{J_n^{\oslash}}{P_n^{\oslash}}K$	$\log P_n$	$\frac{J_n^{\oslash}}{P_n^{\oslash}}K$
TTM (Reduce)	$\frac{J_n^{\circledast}}{P^{\circledast}}K$	$\frac{J_n^{\oslash}}{P_n^{\oslash}}K$	$P_n \log P_n$	$\frac{J_n^{\oslash}}{P^{\circledast}}K$
Gram (Original)	$\frac{J^{\circledast}}{P^{\circledast}}J_n$	$\frac{J^{\circledast}}{P^{\circledast}}P_n + \frac{J_n^2}{P_n}$	P_n	$\frac{J^{\circledast}}{P^{\circledast}} + \left(\frac{J_n}{P_n}\right)^2$
Gram (New)	$\frac{J^{\circledast}}{P^{\circledast}}J_n$	$\frac{J^{\circledast}}{P^{\circledast}} + J_n^2$	P_n	$\frac{J^{\circledast}}{P^{\circledast}} + J_n^2$

We omit leading constant factors and lower order terms. The columns correspond to per-processor costs: number of floating point operations, number of words communicated, number of messages communicated, and amount of temporary local memory required, respectively. Recall $J^{\circledast} = \prod_{n\in[N]} J_n$ and $J_n^{\oslash} = J^{\circledast}/J_n$, with analogous definitions for $P^{\circledast}$ and $P_n^{\oslash}$.

Fig. 6. Multiple-reduction variant of TTM, used to obtain a smaller memory footprint. The data belonging to five processors (in the same processor fiber) is color-coded for tracking.

The data communication costs differ by only a factor of 2. In the Reduce-Scatter variant, the amount of data communicated is the cost of one Reduce-Scatter collective over the mode-n fiber: $\prod_{m\neq n}\lceil J_m/P_m\rceil\lceil K_n/P_n\rceil(P_n-1) \approx J_n^{\oslash}K_n(P_n-1)/P^{\circledast}$. In the multiple-reduction variant, there are P_n Reduce collectives of a data size that is smaller by a factor of P_n, which yields the same asymptotic cost, but incurs an extra factor of 2 due to the cost of the Reduce collective.

The number of messages is fewer for the Reduce-Scatter variant, which has only one collective, at a cost of $\log P_n$ messages. The multiple-reduction variant involves P_n collectives, at a total cost of $2P_n \log P_n$ messages.

The temporary memory ($\bar{\mathcal{W}}$) of the multiple-reduction variant is much lower: $\lceil K_n/P_n\rceil \bar{J}_n^{\oslash}$ words for the multiple-reduction variant versus $K_n\bar{J}_n^{\oslash}$ words for the Reduce-Scatter variant. As an aside, we note the block size of the blocked algorithm can be chosen arbitrarily, navigating a tradeoff between latency cost and memory footprint; we used a version that corresponded to the result size for simplicity, but it assumes the available remaining memory is at least the size of the required memory for the problem.

7 PARALLEL ST-HOSVD COST ANALYSIS

In this section, we analyze the computation, communication, and temporary memory requirements of the ST-HOSVD algorithm. We note that the computation and bandwidth costs are sensitive to mode order, while the latency cost and memory requirements are not. This analysis assumes that the mode order used by the algorithm is increasing by mode index; the costs for other mode orders can be derived by relabeling modes. If the core ranks are specified *a priori*, then an optimal (in terms of flops or communication) mode ordering can be determined similar to the case of reconstruction, as described in Section 8. We also note that the communication costs are sensitive to the processor grid, but the computation cost and memory requirements are not. In this analysis, we use the new Gram algorithm and allow for the choice of TTM algorithm based on the relative sizes of dimensions as described in Section 6.3. To simplify the analysis, we provide an upper bound on the communication costs by assuming the Reduce version, which sends more messages, is used for each mode.

The nth mode Gram performs $R_n^{\circledcirc} I_n^2 I_n^{\circledcirc} / P^{\circledast}$ flops (exploiting the symmetry of the output), the nth mode eigenvalue computation requires $(10/3)I_n^3$ flops, and the nth mode TTM performs $2R_n^{\circledcirc} R_n I_n I_n^{\circledcirc} / P^{\circledast}$ flops. Thus, the leading order terms in the flop costs are

$$\gamma \cdot \left(\frac{1}{P^{\circledast}} \sum_{n=0}^{N-1} \left(I_n^2 + 2R_n I_n \right) R_n^{\circledcirc} I_n^{\circledcirc} + \frac{10}{3} \sum_{n=0}^{N-1} I_n^3 \right).$$

The communication cost of the nth mode (new) Gram computation is $\beta \cdot (R_n^{\circledcirc} I_n I_n^{\circledcirc} / P^{\circledast} \cdot ((P_n - 1)/P_n) + I_n^2) + \alpha \cdot (P_n - 1 + 2 \log P^{\circledast})$. The communication cost of the nth mode (Reduce) TTM is $\beta \cdot (R_n^{\circledcirc} R_n I_n^{\circledcirc} / P_n^{\oslash} \cdot ((P_n - 1)/P_n)) + \alpha \cdot (2P_n \log P_n)$. Thus, the leading-order terms in the bandwidth costs, assuming we use the new Gram algorithm, are

$$\beta \cdot \left(\frac{1}{P^{\circledast}} \sum_{n=0}^{N-1} \left(I_n \frac{P_n - 1}{P_n} + 2R_n (P_n - 1) \right) R_n^{\circledcirc} I_n^{\circledcirc} + \sum_{n=0}^{N-1} I_n^2 \right).$$

The leading-order terms in the latency costs, conservatively assuming we use the Reduce TTM algorithm at each step, are

$$\alpha \cdot \left(2N \log P^{\circledast} + \sum_{n=0}^{N-1} 2P_n \log P_n \right).$$

The temporary memory required for the nth Gram computation is twice the size of the current local tensor data, which is $R_n^{\circledcirc} I_n I_n^{\circledcirc} / P^{\circledast}$. Because this memory can be re-used and $R_n \leq I_n$ for each n, this cost is dominated by the first mode, requiring $2I^{\circledast}/P^{\circledast}$ words. The Gram computation also requires space for storing the output Gram matrix, which is of size $I_n^2/2$ and can be re-used across modes. The eigenvalue computation requires as much as I_n^2 extra memory if all eigenvectors are computed. The temporary memory required for TTM is guaranteed to be smaller than the input tensor, which is always bigger than the output tensor in the case of ST-HOSVD, so the temporary memory required for the nth TTM never exceeds the size required of the nth Gram computation. Thus, the leading-order terms of the total temporary memory required on each processor is

$$2 \max \left\{ \frac{I^{\circledast}}{P^{\circledast}}, \max_n I_n^2 \right\}.$$

8 OPTIMIZED RECONSTRUCTION

After the data has been compressed using ST-HOSVD, the user may wish to move the compressed data to another machine and reconstruct an approximation of the original data there. Since the full reconstructed data set would take up just as much space as the original, we provide the user an option to reconstruct sub-tensors of the original tensor. Thanks to the structure of the Tucker model, linear operations can be applied cheaply to each mode. For example, the user may want to downsample one of the spatial dimensions (mode 2) and select a single variable (mode 3) at a single timestep (mode 4) for the purposes of visualization. This can be accomplished through the following series of TTMs:

$$\mathcal{Z} = \mathcal{G} \times_0 \mathbf{U}_0 \times_1 \mathbf{U}_1 \times_2 \mathbf{C}_2{}^T \mathbf{U}_2 \times_3 \mathbf{C}_3{}^T \mathbf{U}_3 \times_4 \mathbf{C}_4{}^T \mathbf{U}_4,$$

where

$$\mathbf{C}_2 = \begin{bmatrix} 0.5 & 0 & \cdots & 0 \\ 0.5 & 0 & \cdots & 0 \\ 0 & 0.5 & \cdots & 0 \\ 0 & 0.5 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix}, \qquad \mathbf{C}_3 = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}, \qquad \mathbf{C}_4 = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}.$$

The five TTMs can be done in any order to obtain the result, but the Multi-TTM ordering can have a large effect on computational cost, memory footprint, and communication cost (in the parallel case). We demonstrate the effects of reconstruction mode ordering on run time and memory in Section 9.5. In the code, the user can specify the mode ordering or allow the software to automatically select the ordering that minimizes either computational cost or memory footprint. In the case of parallel reconstruction with linear operations applied to the factor matrices, each processor redundantly stores each $\mathbf{C}_n$ and computes $\mathbf{C}_n{}^T\mathbf{U}_n$ locally.

To determine the optimal mode ordering, the TuckerMPI code exhaustively searches over all $N!$ permutations to obtain the optimal ordering, but the optimal ordering can be determined in $O(N \log N)$ time by sorting with a specific comparator [7]. For example, consider the $N = 2$ case, which corresponds to a product of three matrices, with input core matrix dimensions $R_0 \times R_1$ and output subtensor dimensions $K_0 \times K_1$. Computing the mode-0 product followed by the mode-1 product requires $K_0R_0R_1 + K_0R_1K_1$ scalar multiplications, while computing the products in the opposite order requires $R_0R_1K_1 + R_0K_0K_1$ scalar multiplications, so to minimize flops we order the products based on the comparison of these two costs.

More generally, let the input core tensor $\mathcal{G}$ have dimensions $R_0 \times R_1 \times \cdots \times R_{N-1}$ and the output subtensor $\mathcal{Z}$ have dimensions $K_0 \times K_1 \times \cdots \times K_{N-1}$. The key insight is that optimal ordering of all N modes is the one such that every pair of modes is ordered correctly according to the $N = 2$ case. That is, to minimize flops, mode i should precede mode j in the ordering if $K_iR_iR_j + K_iR_jK_j < R_iR_jK_j + R_iK_iK_j$. This can be shown, as argued by Chakravarthy [7], by recognizing that this comparator yields a total ordering on modes and arguing by contradiction. Suppose the optimal ordering is not sorted by this comparator, then there exists two consecutive modes n and $n + 1$ that are out of order. Swapping the two consecutive modes will reduce the overall cost, because the costs of the first $n - 1$ TTMs and the last $N - n - 1$ TTMs are equivalent, but the cost of the n and $n + 1$ TTMs are reduced by the property of the comparator, and we have a contradiction.

Similar arguments can be made for communication (bandwidth) cost and memory footprint. Given the bandwidth cost of Algorithm 5, to minimize words moved, mode i should precede mode j if $R_jK_i(P_i - 1) + K_iK_j(P_j - 1) < R_iK_j(P_j - 1) + K_iK_j(P_i - 1)$. To minimize temporary memory, mode i should precede mode j if $R_i/K_i < R_j/K_j$.

9 EXPERIMENTAL RESULTS

9.1 Experimental Platform

We run all experiments on Skybridge, a Sandia supercomputer consisting of 1,848 dual-socket 8-core Intel Sandy Bridge (2.6 GHz) compute nodes. Each node has 64 GB of memory, a peak flop rate of 332.8 GFLOPS (i.e., 20.8 GFLOPS per core), and the nodes are connected by an Infiniband interconnect. We use Intel compilers and the MKL for BLAS and LAPACK subroutines. We execute 16 MPI processes per node with 1 thread per process unless otherwise stated. Data files are stored on a Lustre file system, and Skybridge's I/O is shared with other clusters. All reported timings in this section are averages over multiple runs. All reported memory requirements are computed by the application, using a wrapper around memory allocation calls, and do not include memory used by the operating system or by the MPI implementation.

9.2 Data Description

We consider two large-scale simulation datasets that were produced by S3D [9], a massively parallel direct numerical simulation of compressible reacting flows, developed at Sandia National Laboratories. The datasets are as follows:

Table 4. Experimental Setup

Dataset Name	Overall Tensor Size	Total Storage	Number of Nodes	Number of Processes	Storage per Node	Storage per Process
SP	500 × 500 × 500 × 11 × 400	4.4 TB	250	4,000	17.6 GB	1.1 GB
JICF	1,500 × 2,080 × 1,500 × 18 × 10	6.7 TB	350	5,600	19.3 GB	1.2 GB

(a) Data sets to be compressed and the number of parallel nodes to be used in the compression experiments.

Dataset	Proc. Config. Name	Processor Grid	Local Tensor Size
SP	A	1 × 1 × 40 × 1 × 100	500 × 500 × 13 × 11 × 4
	B	10 × 8 × 5 × 1 × 10	50 × 63 × 100 × 11 × 40
	C	40 × 10 × 1 × 1 × 10	13 × 50 × 500 × 11 × 40
JICF	A	1 × 16 × 35 × 1 × 10	1,500 × 130 × 43 × 18 × 1
	B	10 × 8 × 7 × 1 × 10	150 × 260 × 215 × 18 × 1
	C	35 × 16 × 1 × 1 × 10	43 × 130 × 1,500 × 18 × 1

(b) Three different processor grid configurations per dataset, to test the efficiency of the compression. The local tensor size may vary, but here we list the largest local size in each dimension.

- **SP:** This 5-way data tensor is of size $500 \times 500 \times 500 \times 11 \times 400$ and corresponds to a cubic $500 \times 500 \times 500$ spatial grid for 11 variables over 400 time steps. Each time step requires 11 GB, so the entire dataset is 4.4 TB. The SP dataset is from the simulation of a 3D statistically steady planar turbulent premixed flame of methane-air combustion [21]. The first 50 time steps (a 550 GB dataset) was used in previous work [2].
- **JICF:** This five-way data tensor of size $1,500 \times 2,080 \times 1,500 \times 18 \times 10$ comes from a $1,500 \times 2,080 \times 1,500$ spatial grid with 18 variables over 10 time steps. Each time step requires 674 GB storage, so the entire dataset is 6.7 TB. The JICF dataset is from a jet in cross-flow simulation, which is a canonical configuration for many combustion systems [25].

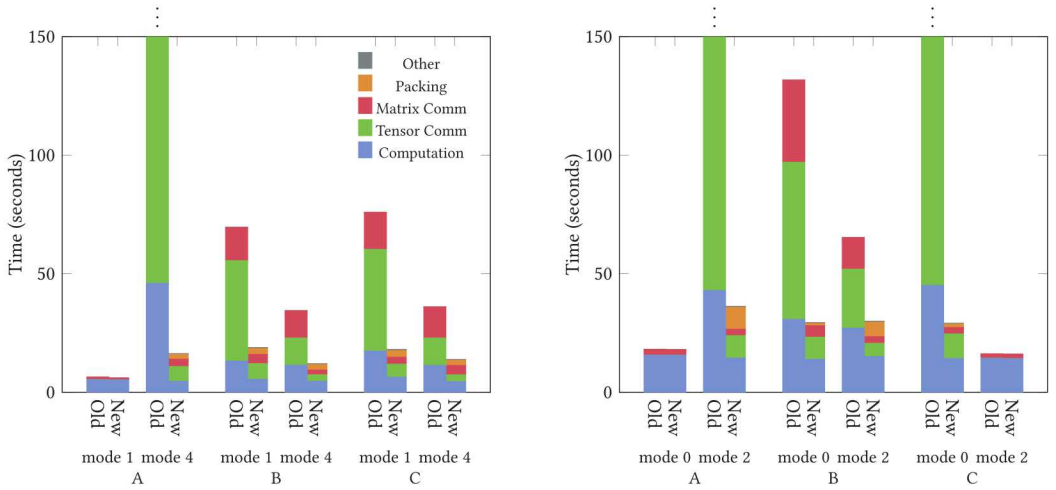
The general experimental setup is described in Table 4. For each dataset, we use the same number of nodes for all experiments: 250 for SP and 350 JICF. Since each node has 64 GB of RAM and runs 16 threads/processes, storage of the full tensor requires a little more than 1/4 of the memory per node (our data is stored in double precision). For each dataset, we consider three different processor grid configurations (A/B/C), as specified in Table 4(b). They vary primarily in how the processors are distributed in the first three modes with scenario A having fewer processors in mode 0, scenario B being more evenly divided on modes 0–2, and scenario C having more processors on mode 0.

For testing and debugging, TuckerMPI also provides both sequential and parallel synthetic tensor generators for the users' convenience. The user specifies the desired size and rank of a tensor, denoted by $(I_0, I_1, \ldots, I_{N-1})$ and $(R_0, R_1, \ldots, R_{N-1})$, respectively, and the amount of relative noise to be added, denoted η. We construct a core tensor $\mathcal{G}$ with dimensions defined by $(R_0, R_1, \ldots, R_{N-1})$; the numbers are drawn from a standard normal distribution. The factor matrices $\mathbf{U}_0 \cdots \mathbf{U}_{N-1}$ are generated in a similar fashion, where $\mathbf{U}_n$ is $I_n \times R_n$. By performing the tensor times matrix mulitplications $\mathcal{G} \times_0 \mathbf{U}_0 \times_1 \mathbf{U}_1 \cdots \times_{N-1} \mathbf{U}_{N-1}$, we obtain a tensor of size $(I_0, I_1, \ldots, I_{N-1})$ with rank $(R_0, R_1, \ldots, R_{N-1})$, which we refer to as $\mathcal{M}$.

After obtaining the tensor $\mathcal{M}$, we wish to add noise to it so that it is not exactly rank $\mathbf{R}$. Let $\mathcal{X} = \mathcal{M} + \eta \frac{\|\mathcal{M}\|}{\|\mathcal{N}\|}\mathcal{N}$, where $\mathcal{N}$ is a randomly generated tensor of noise whose values are also obtained from a standard normal distribution. This gives us $\frac{\|\mathcal{X}-\mathcal{M}\|}{\|\mathcal{X}\|} \approx \eta$. Note that we do not explicitly construct and store $\mathcal{N}$ due to its large size, so we approximate its norm using its expected value: $\|\mathcal{N}\| \approx \sqrt{I^{\circledast}}$.

Dataset	Gram Mode	Proc. Config.	Run Times (sec) Old	New	Ratio	Memory Usage (GB) Old	New	Ratio
SP	1	A	6.3	6.0	1.0	1.15	1.15	1.0
		B	69.6	**18.9**	3.7	2.22	3.32	1.5
		C	75.9	**18.1**	4.2	2.29	3.34	1.5
	4	A	788.9	**16.3**	48.4	2.29	3.43	1.5
		B	34.4	**12.1**	2.8	2.22	3.33	1.5
		C	36.0	**13.8**	2.6	2.29	3.43	1.5
JICF	0	A	18.1	18.0	1.0	1.26	1.24	1.0
		B	131.7	**29.4**	4.5	2.44	3.62	1.5
		C	434.2	**29.1**	14.9	2.43	3.61	1.5
	2	A	390.5	**36.2**	10.8	2.43	3.62	1.5
		B	65.3	**30.0**	2.2	2.44	3.62	1.5
		C	16.2	16.0	1.0	1.26	1.24	1.0

(a) Total times and memory usage per process.



(b) Breakdown for SP dataset

(c) Breakdown JICF dataset

Fig. 7. Gram total run time and breakdowns for old (round-robin variant from Reference [2]) and new (redistribution variant) Gram algorithms on two datasets with different processor configurations as detailed in Table 4(b). The "Gram Mode" or "Mode" refers to the unfolding mode, i.e., n. In the table, differences of more that 2× between the old and new are highlighted in boldface. In the breakdowns, "Packing" only occurs for the new method and refers to the reordering of the data in memory before communication with the other processors, "Tensor Comm" refers to communication of the tensor data, which happens every step of the round-robin procedure for the old method and only once in the redistribution for the new method, and "Matrix Comm" refers to the all-reduce, which happens after each step in the round-robin procedure for the old method and just once in the new method. Note that some bars go past the y-axis limit as indicated by vertical dots.

9.3 Comparison of Gram Algorithms

We compare the two versions of the Gram algorithms described in Section 6.2: the old round-robin variant from Reference [2] and the new redistribution variant from Algorithm 4. We use the two datasets and pick two representative modes, with corresponding experimental conditions detailed in Table 4. The results are shown in Figure 7(a). The new algorithm is up to 48 times faster than the old algorithm and never slower. The least speedup occurs when there is only one processor in the

Gram mode fiber and no communication is performed in either case. The most speedup is when there are 100 processors in each fiber of the Gram mode. Ignoring the cost of the All-Reduce, the communication cost ratio between the old and new algorithms for a Gram operation in mode n is P_n, which is an upper bound on the possible speedup. We see speedups of around 20–50% of that upper bound because of time that both algorithms spend on computation and the increased cost of the All-Reduce for the new algorithm. With respect to processor configurations, there is much less variance in the new algorithm than in the old. The timing for the new algorithm varies by no more than 3×, whereas the timing for the old algorithm varies between 12–62× depending on the processor configuration.

A breakdown of the run time for each of the experiments is shown in Figure 7. We see that the main speedup comes from a reduction in communication of the tensor, but we also see a reduction in computation time. This is the result of the new algorithm making a single BLAS call rather than the old algorithm's P_n BLAS calls on smaller subproblems. The new algorithm does include some overhead for packing and unpacking the data, but it is negligible compared to the benefits of reduced communication.

Figure 7(a) also reports the per-process memory requirement of each algorithm. Because the new algorithm requires re-packing the data and performing an All-to-All collective, it requires space for two extra copies of the local tensor data. The old algorithm requires temporary space for only one extra copy of the local tensor data, to perform its round-robin exchange of data. The ratio of three total copies to two total copies yields the memory footprint ratio of 1.5 as reported in the table. When the number of processors in the mode of the Gram computation is one, no communication is necessary and no extra memory is required, so the memory footprint ratio is 1 in those cases.

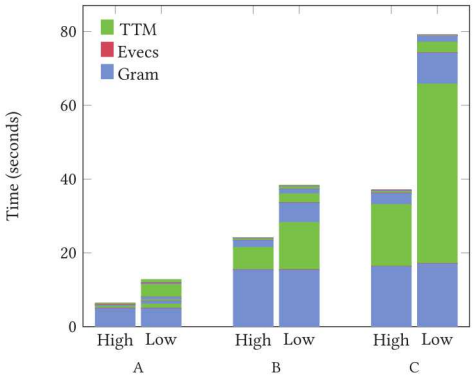
9.4 ST-HOSVD

We analyze the parallel ST-HOSVD (Algorithm 1) using the new redistribution version of the Gram algorithm. First, we consider its performance on two real-world datasets, varying the processor grid but not the number of processors. Second, we do strong and weak scaling studies varying the number of processors.

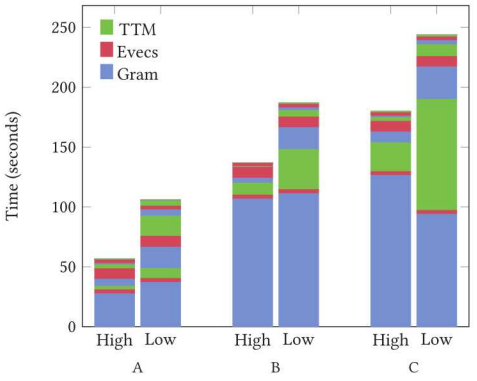
9.4.1 Compression of Combustion Data. We use TuckerMPI to compress the SP and JICF datasets described in Table 4(a) using the number of processors and grids specified in Table 4(b). In terms of the pre-processing described in Section 3.5, there was no preprocessing on the SP data, since it was already scaled, and max scaling was applied to the JICF data. For each dataset, we consider two relative error tolerances: 1e-2 and 1e-4, which we refer to as "High" and "Low" compression, respectively. The scenarios are summarized in Figure 8(a) along with the compressed core size, total storage for the core and factor matrices, and overall compression ratio. For real-world datasets such as these, the compression potential depends on the amount of redundancy that is inherent in the data. For time-evolving simulations, there may be spatial regions with minimal change, and so these parts can be highly compressed. In our code, the user specifies the desired relative error tolerance (ϵ), from which the level of compression is determined on the fly. (Alternatively, the software allows the user to specify the desired final core size, from which the final relative error is determined.) For our two datasets, the high-compression scenario yields reductions in size of 4–5 orders of magnitude. We note that our subject-matter experts have deemed the high compression datasets to be faithful representations that are scientifically useful. For instance, Figure 9 shows a visual comparison between the original and compressed versions. Specifically, Figure 9 shows a temperature isosurface at the 201st (middle) time point, used to track the boundaries of a flame during the simulation. The errors in the reconstructions of the compressed versions

Dataset	Scenario Name	Relative Error	Compressed Core Size	Total Storage	Compression Ratio
SP	High	1e-2	$30 \times 38 \times 35 \times 6 \times 11$	21.5 MB	2×10^5
	Low	1e-4	$95 \times 129 \times 125 \times 7 \times 125$	10.7 GB	4×10^2
JICF	High	1e-2	$90 \times 61 \times 48 \times 13 \times 6$	167 MB	4×10^4
	Low	1e-4	$424 \times 387 \times 261 \times 18 \times 10$	45.7 GB	1×10^2

(a) ST-HOSVD compression results (independent of processor grid)



(b) ST-HOSVD on SP dataset with different processor grids

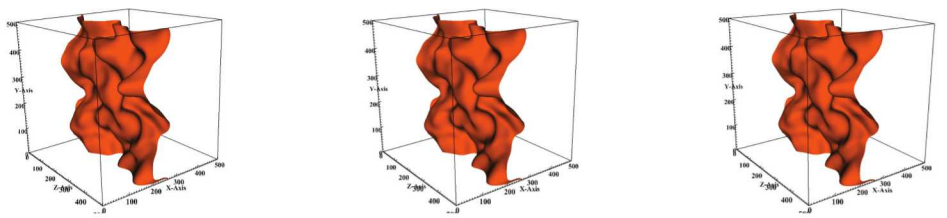


(c) ST-HOSVD on JICF dataset with different processor grids

Proc. Config.	Compression Scenario	SP dataset Memory Usage (GB)	SP dataset I/O Time (s) Input	SP dataset I/O Time (s) Output	SP dataset Comp. Time (s)	JICF dataset Memory Usage (GB)	JICF dataset I/O Time (s) Input	JICF dataset I/O Time (s) Output	JICF dataset Comp. Time (s)
A	High ϵ=1e-2	1.22	370	0.2	6	1.43	2308	2	57
	Low ϵ=1e-4	1.42		22	13	2.24		424	106
B	High ϵ=1e-2	3.33	877	0.9	24	3.62	2187	4	137
	Low ϵ=1e-4	3.33		983	38	3.62		18517	187
C	High ϵ=1e-2	3.34	861	0.8	37	3.61	2077	5	180
	Low ϵ=1e-4	3.34		13470	79	3.61		DNC	244

(d) STHOSVD per-core memory usage, I/O time, and computation time

Fig. 8. ST-HOSVD compression, run time breakdown, memory usage, and I/O cost for different choices of ϵ (in in Algorithm 1) and processor grids. The different processor grids (A/B/C) are given in Table 4(b). In the breakdowns, since these are five-way datasets, there are five iterations of ST-HOSVD, each of which calls Gram, local eigenvalue decomposition (Evecs), and TTM. These are stacked from mode 0 (bottom) to mode 4 (top). The mode-0 calls are the most expensive, because the tensor is reduced in size for subsequent iterations, and Evecs is never a bottleneck.



(a) Original (b) Low compression (ϵ=1e-4) (c) High compression (ϵ=1e-2)

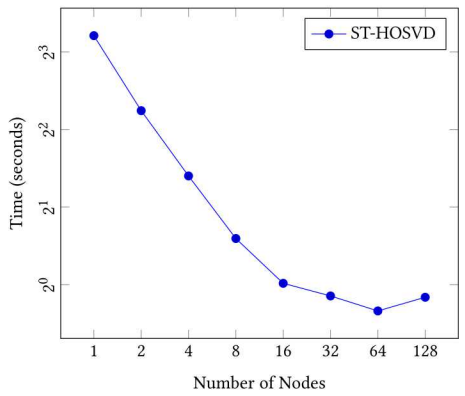
Fig. 9. Temperature isosurfaces at a given timestep in SP data computed from the original and compressed (and reconstructed) data sets.

are unobservable in this visualization. As we see from Figure 8, the resulting compressed data sets are small enough to be easily shared across high-speed networks and/or analyzed on workstations as described in Section 9.5.

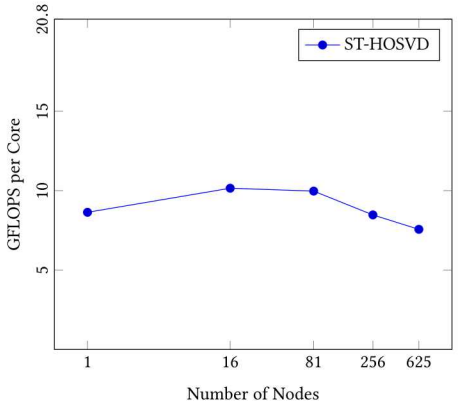
Figure 8 shows the run time results of ST-HOSVD using the different compression scenarios in Figure 8(a) and processor grid configurations in Table 4(b). Note that the degree of compression has no dependence whatsoever on the processors grid configuration; the configuration impacts only the run time. In the figure, the run times are broken down into color-coded segments corresponding to the three main kernels: Gram, local eigenvalue decomposition (Evecs), and TTM. As these are five-way tensors, there are five calls to each kernel. The times corresponding to mode zero are at the bottom and the most expensive. Since the tensor is reduced at each iteration (via the call to TTM), the calls become significantly cheaper so that the breakdown is hardly apparent past the first 2-3 iterations/modes. Comparing the "high" and "low" scenarios, the initial Gram computations are roughly equivalent, but all other computations in the 1e-2 case finish more quickly than the 1e-4 case, because the data is compressed more drastically at each step. Comparing SP and JICF datasets, the Evecs computation is negligible for SP but is noticeable for JICF, because its largest mode sizes are 3–4 times larger than for SP (the eigenvector computation is not parallelized). Comparing across processor grid configurations (A/B/C), loading more processors onto later modes generally improves run time, as the heavy communication steps are performed on data that can be orders of magnitude smaller than the initial tensor. The fastest of the processor grids are 3–6× faster than the slowest of the processor grids. This is not included in the figure, but the average time taken for preprocessing the JICF data (scaling each mode-3 slice by the inverse of the maximum entry) is 8 seconds and varied minimally across processor grids. This amounts to at most 15% of the run time of ST-HOSVD after the tensor is loaded in memory. The SP data in our experiments has already been scaled, so TuckerMPI does no pre-processing.

We note that this experiment does not consider alternative mode orderings, which can have a significant effect on run time. In the case of ST-HOSVD with a specified tolerance, the core tensor size is not known *a priori*, so it is not possible to pick an optimal ordering as discussed in Section 8. However, one can use some heuristics to pick a mode ordering. For example, starting with a mode whose processor grid dimension is 1 avoids communication of the tensor in the first Gram (which is typically the most expensive operation). Indeed, in this experiment, the fastest processor grid (A) has one processor in mode 0 for both SP and JICF data sets, but using the natural mode ordering 01234 is not necessarily optimal, even for that processor grid.

Figure 8(d) shows the memory footprints and I/O times of the different cases. The input SP tensor requires about 1.1 GB per core on 4,000 cores, and TuckerMPI requires about 3× that much memory to complete its computations, which is dominated by the memory required by the initial Gram computation. When a processor grid with only one processor in mode 0 is used, the memory footprint is only about 10–30% more than the initial data, depending on how much compression is achieved through the algorithm. The JICF data on 5,600 cores requires about 1.2 GB per core, and again we see a memory footprint of about 3× that amount. In terms of I/O time, the TuckerMPI code uses the MPI I/O interface for reading and writing binary files of multidimensional arrays. From the input results, we see a file reading bandwidth of 3–10 GB/s, requiring $O(10)$ minutes to read the input tensors from disk. In comparison, computing the ST-HOSVD takes $O(1)$ minute per Figure 8(d). For instance, the SP-High-A scenario is 100× faster than reading the data from from disk. Writing to disk is much more expensive than reading. As a result, writing the compressed representation takes a disproportionate amount of time. We believe this is an artifact of the underlying MPI I/O implementation with the Lustre filesystem on Skybridge. One experiment, labeled "DNC," did not complete due to an error within the MPI I/O implementation. Improving I/O performance for the low compression scenario is a topic of future work.



(a) Strong scaling performance of ST-HOSVD for $256 \times 256 \times 256 \times 256$ tensor compressed to size $32 \times 32 \times 32 \times 32$ (4096× compression), using 2^k nodes for $0 \leq k \leq 7$. The processor grids are 1×1×2×8, 1×1×2×16, 1×1×4×16, 1×1×8×16, 1×1×16×16, 1×2×16×16, 1×4×16×16, 1×4×16×32.



(b) Weak scaling performance of ST-HOSVD for $200k \times 200k \times 200k \times 200k$ tensor with reduced size $20k \times 20k \times 20k \times 20k$, using k^4 nodes and a $1 \times 1 \times 4k^2 \times 4k^2$ processor grid for $1 \leq k \leq 5$. For reference, the peak GFLOPS per core is 20.8.

Fig. 10. Parallel scaling experiments on synthetic data. There are 16 cores per node.

9.4.2 Strong Scaling with Synthetic Data. In this section, we demonstrate the scalability of our code on a synthetic 4D tensor with dimensions $256 \times 256 \times 256 \times 256$. The experimental setup is as follows. We fix the core dimensions to $32 \times 32 \times 32 \times 32$ (4,096× compression) so that all mode orderings are equivalent, and we scale from 1 node up to 128 nodes (2,048 cores), using 1 MPI process per core. Since all the dimensions are the same, the mode ordering is irrelevant. We used the following processor grids: 1×1×2×8, 1×1×2×16, 1×1×4×16, 1×1×8×16, 1×1×16×16, 1×2×16×16, 1×4×16×16, 1×4×16×32, chosen heuristically to have fewer processors in the first and second modes to minimize communication in the early Gram computations, which dominate the run time.

The run times are reported in Figure 10(a). The input tensor is 32 GB in size, about half the memory available on a single node. The performance scales well up to 16 nodes (256 cores), achieving 9× speedup over the single node, because the run time is dominated by the local Gram computation in the first node, which is perfectly parallelized. The degradation of performance after 16 nodes is caused in large part by the local computation in the first Gram computation (a single call to `syrk`) failing to scale perfectly. This is due to the local dimensions becoming too small and performance variability across nodes. We note that performance reported in Reference [2] shows strong scaling of nearly the same algorithm on nearly the same problem to 256 nodes, but it was benchmarked on a different parallel computer.

9.4.3 Weak Scaling with Synthetic Data. In this section, we demonstrate the weak scalability of our code on synthetic 4D tensors whose sizes scale with the number of processors so that the portion per processor remains constant. We ran our code on 2^k nodes (for $1 \leq k \leq 5$), using 16 MPI processes per node, with a $1 \times 1 \times 4k^2 \times 4k^2$ processor grid. We take a tensor of order $200k \times 200k \times 200k \times 200k$ and reduce it to a core size of $20k \times 20k \times 20k \times 20k$. This means that the local data on each node is approximately 0.8 GB. The processor grid was chosen as the best performing among $2k \times 2k \times 2k \times 2k$, $k \times k \times 4k \times 4k$ and $1 \times k \times 4k \times 4k^2$. The largest speedup of the $1 \times 1 \times 4k^2 \times 4k^2$ grid over the other grids was about 70%, and the $1 \times k \times 4k \times 4k^2$ grid performed nearly as well.

Table 5. Intermediate Tensor Sizes in the Partial Reconstruction for Two Different TTM Orderings, Using the High Compression Scenario (ϵ=1e-2) for the SP Dataset

	Ordering 01234		Ordering 43120	
TTM	**Dimensions**	**Intermed. Size (GB)**	**Dimensions**	**Intermed. Size (GB)**
None	$30 \times 38 \times 35 \times 6 \times 11$	0.02	$30 \times 38 \times 35 \times 6 \times 11$	0.02
1st	$500 \times 38 \times 35 \times 6 \times 11$	0.35	$30 \times 38 \times 35 \times 6 \times 1$	<0.01
2nd	$500 \times 500 \times 35 \times 6 \times 11$	4.62	$30 \times 38 \times 35 \times 1 \times 1$	<0.01
3rd	$500 \times 500 \times 500 \times 6 \times 11$	66.0	$30 \times 500 \times 35 \times 1 \times 1$	<0.01
4th	$500 \times 500 \times 500 \times 1 \times 11$	11.0	$30 \times 500 \times 500 \times 1 \times 1$	0.06
5th	$500 \times 500 \times 500 \times 1 \times 1$	1.00	$500 \times 500 \times 500 \times 1 \times 1$	1.00

The TTM ordering can make a dramatic different in memory usage.

Figure 10(b) reports the performance in terms of GFLOPS per core. We observe that weak-scaling performance is generally preserved up to 625 nodes, with the code achieving 40–50% of peak performance throughout. In this experiment, the amount of local computation is held fixed, while the communication is increased with the number of processors. The weak scaling is possible, because the run time remains dominated by computation. On 1 node ($k = 1$), the tensor size is 12.8 GB, and on 625 nodes ($k = 5$) it is 8 TB. Compared to the performance reported in Reference [2], we observe similar overall performance relative to the architecture.

9.5 Reconstruction

Compressing data makes it cheaper to store and to transmit, but ultimately it needs to be reconstructed to be useful. We expect that most users will do only *partial* reconstructions. These can be used to visualize a portion of the data, extract summary statistics, and so on. Moreover, this can oftentimes be done on a workstation rather than a parallel computer, making analysis much simpler. We can also reconstruct the *entire* dataset, which is primarily useful for comparison to the original dataset for quality control. In either case, reconstruction is a Multi-TTM operation as described in Section 8.

9.5.1 Partial Reconstruction. Our aim in this section is to show that we can do *partial* reconstructions on a laptop or workstation, so we run all experiments sequentially using only one MPI process (and thus only one node and one thread). We note that each node on Skybridge has 64 GB, and that the sequential execution has access to all 64 GB. We consider the SP dataset, as described in Section 9.2.

Visualization of Single Time Step. One partial reconstruction for visualization is to extract the entire $500 \times 500 \times 500$ grid corresponding to a *single* variable (out of 11) at a *single* timestep (out of 400). This results in a tensor of size $500 \times 500 \times 500 \times 1 \times 1$, which requires 1 GB of storage. This is the first step used to generate the visualizations of the compressed data in Figure 9, for example.

As discussed in Section 8, the mode ordering is important in the reconstruction. We never want the intermediate size to be bigger than the larger of the input and output tensor sizes, but this can happen for the wrong mode ordering. Consider the high compression scenario and the intermediate tensor sizes that result as shown in Table 5. Using the ordering 01234 results in a 66 GB tensor after the third TTM in the reconstruction. But ordering 43120 never has an intermediate result larger than the final reconstruction. We stress that any ordering produces the same result (up to floating point error) and the difference is in the size of the intermediate results and the run time.

In Table 6, we report the maximum memory usage, the compute time (Multi-TTM), and the I/O times for different mode orders. The sizes of the core tensors on disk are 21 MB and 11 GB per Figure 8(a), and the size of the reconstructed output is 1 GB. The I/O time shows a bandwidth rate

Table 6. Partial Reconstruction Results on the SP Dataset with a *Single* MPI Process, Computing a Partial Reconstruction of Size $500 \times 500 \times 500 \times 1 \times 1$ (1 GB)

Compression Scenario	Mode Order	Max Memory Usage (GB)	Compute Time (s)	I/O Time (s) Input	I/O Time (s) Output
High ϵ=1e-2	01234	*out of memory*		0.05 (21.5 MB)	1.01 (1 GB)
	03421	1.08	1.55		
	24013	7.00	8.88		
	34021	7.00	1.19		
	43120	1.08	1.05		
Low ϵ=1e-4	01234	*out of memory*		12.23 (10.7 GB)	
	03421	*out of memory*			
	24013	53.62	129.70		
	34021	12.26	5.83		
	43120	10.81	4.72		

Table 7. Summary Statistic Results on the SP Dataset with a *Single* MPI Process, Computing the Average Fraction of Carbon Dioxide Across All Time Steps and Entire Physical Grid

Compression Scenario	Mode Order	Max Memory Usage (GB)	Compute Time (s)	I/O Time (s) Input	I/O Time (s) Output	Relative Error
High (ϵ=1e-2)	01234	0.022	0.16	0.69 (21.5 MB)	0.04 (8 B)	6.7e-6
Low (ϵ=1e-4)	01234	10.84	2.63	13.02 (10.7 GB)		5.7e-8

Relative error is in comparison to the value computed based on the original data.

of about 1 GB/sec. In the high compression case, the partial reconstruction is larger than then the input; in contrast, the reverse is true for the low compression scenario. For both the high and low compression scenarios, the unique minimizer of the flops is mode order 43120, and it is one of the memory minimizers. Compared with the other orders benchmarked in the experiment, the optimal order runs as much as an order of magnitude faster, and some orders, including the straightforward 01234 order, fail due to out-of-memory errors.

Computing a Summary Statistic. As an example summary statistic for the SP data, we can compute the average fraction of carbon dioxide across all space and time in a few seconds. Mathematically, in the notation of Section 8, we set $\mathbf{C}_0$, $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_4$ to be all-ones vectors and $\mathbf{C}_3$ (the mode corresponding to variables) to be all zeros except for a one in the index that corresponds to carbon dioxide. Table 7 shows the results of this computation on a single node for both compression scenarios. The average fraction of carbon dioxide across all space and time is 0.0534 (calculated using the original data).[1] In comparison, the relative error of the averages computed from the compressed versions are 6.7e-6 (high compression) and 5.7e-8 (low compression). The memory required is only that of storing the compressed data, and the time is dominated by the cost of reading the core tensor from disk. For the high compression case, the computation took less than 1 s in total; for the low compression case, it required less than 16 s. Recall that working with the original data set requires a parallel computer and 100 s of nodes just to read the data.

9.5.2 Full Reconstruction of the SP Data. As mentioned above, we do not expect users to employ full reconstruction very often, but it is useful as a diagnostic tool to check the quality of the

[1]In preprocessing, the data was rescaled to have a maximum value of 0.5, and the mean based on the rescaled data was 0.3014.

Table 8. Full Reconstruction Results of the SP Dataset Using 250 Nodes/4000 MPI Processes

Compression Scenario	Mode Order	Max Memory Usage (GB)	Compute Time (s)	I/O Time (s) Input	I/O Time (s) Output
High ϵ=1e-2	12034	2.57	320.34	4.02 (21.5 MB)	3118.57 (4.4 TB)
	34120	1.21	27.51		
	41203	1.77	16.39		
	42103	1.77	5.04		
Low ϵ=1e-4	12034	*out of memory*		15.24 (10.7 GB)	
	34120	1.37	82.46		
	41203	1.88	53.74		
	42103	1.88	19.43		

The processor grid is size $1 \times 1 \times 40 \times 1 \times 100$. The "Max Memory Usage" is per process.

approximation. Hence, we reconstructed the full SP dataset for both the high and low compression scenarios using a variety of mode orderings and processor configuration A ($1 \times 1 \times 40 \times 1 \times 100$), since it resulted in the fastest compression time. The results are reported in Table 8.

The I/O timings are averaged over the runs with different mode orderings. As the output is about 400 times larger than the larger of the two inputs, the overall time for the experiment is dominated by writing the output to disk, which takes almost an hour. (This supports the idea of avoiding reading and writing the full data sets to disk.) As compared with the bandwidth rate of one MPI process (see Table 6), the parallel I/O bandwidth rate is slower for the smaller inputs and about 50% faster for the larger output.

As discussed in Section 8, the computational and communication costs for the Multi-TTM, as well as the temporary memory footprint, are all dependent on the mode ordering of the individual TTMs. Furthermore, the mode ordering that minimizes computation need not be the same as the one that minimizes communication or memory. For the high compression scenario, mode order 41203 minimizes computation, 42103 minimizes communication, and 34120 minimizes memory. We also experiment with ordering 12034, which requires about 60% more temporary memory than the optimal orders and yields an out-of-memory error in the high compression scenario. The 34120 order yields minimum max memory usage for both scenarios, and 42103 is the fastest for both scenarios.

10 CONCLUSION

The Tucker tensor decomposition is useful for compression of many large-scale datasets, because it uncovers latent low-dimensional structure. For multi-terabyte datasets that arise in direct numerical simulation of combustion reactions, we demonstrate that the Tucker decomposition can obtain five orders of magnitude in compression by reducing a 4.4 TB dataset to 21.5 MB. The compression time is an order of magnitude faster than simply reading the data. Austin, Bader, and Kolda [2] proposed the first parallel Tucker implementation. In this article, we build upon their work, explaining the details of the parallel and serial algorithms as well as the local and distributed data layouts. Additionally, we have improved upon the algorithm in Reference [2] and so can now run on much larger datasets. We also explain in detail how to reconstruct portions of the data, one of the most important benefits of the Tucker decomposition, without requiring any parallel resources. We show that it is possible to reconstruct portions of the data on a single processor in only a few seconds and using no more memory than the size of the input or output (whichever is larger).

Other parallel algorithms and implementations for computing Tucker decompositions of dense tensors using Higher-Order Orthogonal Iteration include Reference [6], which uses TTM-trees

and performs dynamic regridding to optimize computation and communication costs, and Reference [26], which uses TTM-trees and approximate information to update factor matrices in each iteration. Choi, Lui, and Chakaravarthy [10] implement ST-HOSVD for dense tensors on a cluster equipped with GPUs, and they also use randomized SVD algorithms to improve run time. In the sparse case, there have also been several algorithmic innovations to improve the parallel performance of Higher-Order Orthogonal Iteration [19, 27, 30]; these optimizations exploit the sparsity of the data tensor and avoid memory overhead of temporary dense data structures.

In future work, we hope to create a library for *in situ* compression that can compress at each time step in a simulation. Ideally, the simulation would never write the full dataset to disk but only compressed versions. Although these compressed versions cannot be used for restart in the event of a failure, they are a sort of "thumbnail" of the full simulation. This would save both time (for I/O) and disk space, not to mention making the sharing of data much easier.

We have not compared our approach to other methods of compression but hope to do so in future studies. Ballester-Ripoll, Lindstrom, and Pajarola [4] have recently compared Tucker compression (computed in serial) to the ZFP, SZ, and SQ compression methods and determined that this method "typically produces renderings that are already close to visually indistinguishable to the original data set." Most other compression methods are focused on pointwise errors and achieve only about $O(10)$ compression and moreover divide up the space into blocks and compress the blocks individually. The Tucker approach we use here bounds the *overall* error and looks for large-scale patterns with the advantage being that it can get much higher compression, e.g., up to $O(10^5)$. A comparison study would need to determine how to compare across different types of compression metrics (e.g., overall versus pointwise) and have a common large-scale dataset for the comparisons. We note that we can also potentially combine compression methods, using other methods to compress the core that results from the Tucker compression. Indeed, the "readme" file for the SZ Fast Error-Bounded Scientific Data Compressor lists Tucker compression as an option.[2]

Li et al. [23] evaluate the impact of wavelet compression (up to 512×) for turbulent-flow data on visualization and analysis tasks, and such analysis would be interesting to consider also in the case of Tucker compression.

The data we compress in our study corresponds to a regular rectilinear grid. Another topic for future work is dealing with unstructured and/or non-rectilinear grids that are not neatly represented as a tensor. We also assume that the data is dense, but there are applications in data mining that have sparse tensors so this is another potential topic for study.

ACKNOWLEDGMENTS

We thank the anonymous referees for their comments and suggestions on the article. We are indebted to Woody Austin for his work on the original prototype software on which this work is based. We thank Hemanth Kolla for creating Figure 9 and his overall support of this work. We are also grateful to our colleagues Gavin Baker, Casey Battaglino, Cannada (Drew) Lewis, Jed Duersch, Alex Gorodetsky, and Prashant Rai for discussions about this software.

REFERENCES

[1] S. Afra, E. Gildin, and M. Tarrahi. 2014. Heterogeneous reservoir characterization using efficient parameterization through higher order SVD (HOSVD). In *Proceedings of the American Control Conference*. 147–152. DOI: https://doi.org/10.1109/ACC.2014.6859246

[2] Woody Austin, Grey Ballard, and Tamara G. Kolda. 2016. Parallel tensor compression for large-scale scientific data. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. 912–922. DOI: https://doi.org/10.1109/IPDPS.2016.67 arXiv:1510.06689

[2] See https://github.com/disheng222/SZ/blob/master/README (ver. 12c9356), line 133.

[3] Grey Ballard, Koby Hayashi, and Ramakrishnan Kannan. 2018. *Parallel Nonnegative CP Decomposition of Dense Tensors.* Technical Report 1806.07985. Retrieved from https://arxiv.org/abs/1806.07985.

[4] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2019. TTHRESH: Tensor compression for multidimensional visual data. *IEEE Trans. Visual. Comput. Graph.* (2019). DOI: https://doi.org/10.1109/TVCG.2019.2904063

[5] Rafael Ballester-Ripoll and Renato Pajarola. 2015. Lossy volume compression using Tucker truncation and thresholding. *Vis. Comput.* 32 (May 2015), 1433–1446. DOI: https://doi.org/10.1007/s00371-015-1130-y

[6] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar. 2017. On optimizing distributed Tucker decomposition for dense tensors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17).* 1038–1047. DOI: https://doi.org/10.1109/IPDPS.2017.86

[7] Venkatesan Chakravarthy. 2017. Personal communication.

[8] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. 2007. Collective communication: Theory, practice, and experience. *Concurr. Comput.: Pract. Exper.* 19, 13 (2007), 1749–1783. DOI: https://doi.org/10.1002/cpe.1206

[9] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. 2009. Terascale direct numerical simulations of turbulent combustion using S3D. *Comput. Sci. Discov.* 2, 1 (2009), 015001. DOI: https://doi.org/10.1088/1749-4699/2/1/015001

[10] Jee Choi, Xing Liu, and Venkatesan Chakaravarthy. 2018. High-performance dense Tucker decomposition on GPU clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18).* IEEE Press, Piscataway, NJ. Retrieved from http://dl.acm.org/citation.cfm?id=3291656.3291712.

[11] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.* 21, 4 (2000), 1253–1278. DOI: https://doi.org/10.1137/S0895479896305696

[12] S. Di and F. Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'16).* 730–739. DOI: https://doi.org/10.1109/IPDPS.2016.11

[13] Nathaniel Fout, Kwan-Liu Ma, and James Ahrens. 2005. Time-varying, multivariate volume data reduction. In *Proceedings of the ACM Symposium on Applied Computing (SAC'05).* ACM, New York, NY, 1224–1230. DOI: https://doi.org/10.1145/1066677.1066953

[14] A. García-Magariño, S. Sor, and A. Velazquez. 2016. Data reduction method for droplet deformation experiments based on high order singular value decomposition. *Exper. Therm. Fluid Sci.* 79 (Dec. 2016), 13–24. DOI: https://doi.org/10.1016/j.expthermflusci.2016.06.017

[15] Wolfgang Hackbusch. 2014. Numerical tensor calculus. *Acta Numerica* 23 (2014), 651–742. DOI: https://doi.org/10.1017/S0962492914000087

[16] D. R. Hatch, D. del Castillo-Negrete, and P. W. Terry. 2012. Analysis and compression of six-dimensional gyrokinetic datasets using higher order singular value decomposition. *J. Comput. Phys.* 231, 11 (June 2012), 4234–4256. DOI: https://doi.org/10.1016/j.jcp.2012.02.007

[17] Koby Hayashi, Grey Ballard, Yujie Jiang, and Michael J. Tobia. 2018. Shared-memory parallelization of MTTKRP for dense tensors. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18).* ACM, New York, NY, 393–394. DOI: https://doi.org/10.1145/3178487.3178522

[18] A. Karami, M. Yazdi, and G. Mercier. 2012. Compression of hyperspectral images using discrete wavelet transform and Tucker decomposition. *IEEE J. Select. Top. Appl. Earth Observ. Remote Sens.* 5, 2 (April 2012), 444–450. DOI: https://doi.org/10.1109/JSTARS.2012.2189200

[19] O. Kaya and B. Uçar. 2016. High performance parallel algorithms for the Tucker decomposition of sparse tensors. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP'16).* 103–112. DOI: https://doi.org/10.1109/ICPP.2016.19

[20] Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51, 3 (Sept. 2009), 455–500. DOI: https://doi.org/10.1137/07070111X

[21] Hemanth Kolla, Xin-Yu Zhao, Jacqueline H. Chen, and N. Swaminathan. 2016. Velocity and reactive scalar dissipation spectra in turbulent premixed flames. *Combust. Sci. Technol.* 188, 9 (2016), 1424–1439. DOI: https://doi.org/10.1080/00102202.2016.1197211

[22] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).* ACM, New York, NY. DOI: https://doi.org/10.1145/2807591.2807671

[23] S. Li, K. Gruchalla, K. Potter, J. Clyne, and H. Childs. 2015. Evaluating the efficacy of wavelet configurations on turbulent-flow data. In *Proceedings of the IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV'15).* 81–89. DOI: https://doi.org/10.1109/LDAV.2015.7348075

[24] P. Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Trans. Visual. Comput. Graph.* 20, 12 (2014), 2674–2683. DOI: https://doi.org/10.1109/TVCG.2014.2346458

[25] Sgouria Lyra, Benjamin Wilde, Hemanth Kolla, Jerry M. Seitzman, Timothy C. Lieuwen, and Jacqueline H. Chen. 2015. Structure of hydrogen-rich transverse jets in a vitiated turbulent flow. *Combust. Flame* 162, 4 (2015), 1234–1248. DOI: https://doi.org/10.1016/j.combustflame.2014.10.014

[26] Linjian Ma and Edgar Solomonik. 2018. *Accelerating Alternating Least Squares for Tensor Decomposition by Pairwise Perturbation.* Technical Report 1811.10573. Retrieved from https://arxiv.org/abs/1811.10573.

[27] S. Oh, N. Park, S. Lee, and U. Kang. 2018. Scalable Tucker factorization for sparse tensors—Algorithms and discoveries. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE'18).* 1120–1131. DOI: https://doi.org/10.1109/ICDE.2018.00104

[28] Anh-Huy Phan, Petr Tichavsky, and Andrzej Cichocki. 2013. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Trans. Signal Process.* 61, 19 (Oct. 2013), 4834–4846. DOI: https://doi.org/10.1109/TSP.2013.2269903

[29] Shaden Smith and George Karypis. 2016. A medium-grained algorithm for distributed sparse tensor factorization. In *Proceedings of the IEEE 30th International Parallel and Distributed Processing Symposium.* 902–911. DOI: https://doi.org/10.1109/IPDPS.2016.113

[30] Shaden Smith and George Karypis. 2017. Accelerating the Tucker decomposition with compressed sparse tensors. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par'17)*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 653–668. DOI: https://doi.org/10.1007/978-3-319-64203-1_47

[31] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (2005), 49–66. DOI: https://doi.org/10.1177/1094342005051521

[32] Ledyard R. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31 (1966), 279–311. DOI: https://doi.org/10.1007/BF02289464

[33] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. 2012. A new truncation strategy for the higher-order singular value decomposition. *SIAM J. Sci. Comput.* 34, 2 (Jan. 2012), A1027–A1052. DOI: https://doi.org/10.1137/110836067

[34] M. A. O. Vasilescu and D. Terzopoulos. 2002. Multilinear analysis of image ensembles: TensorFaces. In *Proceedings of the 7th European Conference on Computer Vision (ECCV'02) (Lecture Notes in Computer Science)*, Vol. 2350. Springer, 447–460. DOI: https://doi.org/10.1007/3-540-47969-4_30

Received January 2019; revised January 2020; accepted January 2020