Parallel Rank-Adaptive Higher Order Orthogonal Iteration

Joao Pinheiro
Wake Forest University
Winston-Salem, USA
deolj19@wfu.edu

Aditya Devarakonda
Wake Forest University
Winston-Salem, USA
devaraa@wfu.edu

Grey Ballard
Wake Forest University
Winston-Salem, USA
ballard@wfu.edu

Abstract

Higher Order Orthogonal Iteration (HOOI) is an iterative algorithm that computes a Tucker decomposition of fixed ranks of an input tensor. In this work we modify HOOI to determine ranks adaptively subject to a fixed approximation error, apply optimizations to reduce the cost of each HOOI iteration, and parallelize the method in order to scale to large dense datasets. We show that HOOI is competitive with the Sequentially Truncated Higher Order Singular Value Decomposition (STHOSVD) algorithm, particularly in cases of high compression ratios. Our proposed rank-adaptive HOOI can achieve comparable approximation error to STHOSVD in less time, sometimes achieving a better compression ratio. We demonstrate that our parallelization scales well over thousands of cores and show using three scientific simulation datasets that HOOI outperforms STHOSVD in high-compression regimes. For example, for a 3D fluid-flow simulation dataset, HOOI computed a Tucker decomposition 82x faster and achieved a compression ratio 50% better than STHOSVD's.

CCS Concepts

• **Mathematics of computing → Mathematical software performance**; • **Theory of computation → Massively parallel algorithms**.

Keywords

Tucker decomposition, Subspace iteration, Block coordinate descent, Higher-Order SVD, Parallel algorithms, Dimension trees

ACM Reference Format:
Joao Pinheiro, Aditya Devarakonda, and Grey Ballard. 2025. Parallel Rank-Adaptive Higher Order Orthogonal Iteration. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3712285.3759865

1 Introduction

The Tucker decomposition is a type of low-rank tensor approximation of multidimensional data that trades off compression ratio with approximation error. Previous work has shown that Tucker is particularly effective at compressing datasets arising from scientific simulations occurring in two or three spatial dimensions and through time, in part because algorithms for computing the Tucker decomposition can scale to high performance computing platforms (see, e.g., [1, 3, 4, 6, 8, 10, 11]). When used as a technique for compression, the Tucker format has an advantage that subtensors can be efficiently decompressed without reconstructing the full tensor, which allows for fast visualization of particular time steps, spatial regions, or quantities of interest. The Tucker decomposition is a generalization of the truncated singular value decomposition (SVD) that consists of a core tensor, with as many modes as the input, and a set of factor matrices. The dimensions of the core tensor are known as the Tucker ranks, and like the truncated SVD, smaller ranks yield higher compression but larger error. In the rank-specified formulation of the Tucker approximation problem, we minimize the error over all Tucker-format tensors of fixed ranks. In the error-specified formulation, we maximize the compression ratio subject to the approximation satisfying an error threshold.

As we describe in § 2, a direct algorithm known as Sequentially Truncated Higher Order SVD (STHOSVD) achieves quasi-optimal accuracy among decompositions of specified ranks, and it can adaptively determine ranks to solve the error-specified formulation [13, 24]. The Higher Order Orthogonal Iteration (HOOI) algorithm is an iterative method that solves the rank-specified formulation of the problem [12, 14, 17]. Conventional wisdom has held that because STHOSVD solves the rank-specified problem to within a small factor of the optimal solution, HOOI is useful only to refine STHOSVD's solution and is typically unnecessary [4, 9, 19]. Based on the observations that (1) a single iteration of HOOI is computationally cheaper than STHOSVD, particularly when the compression ratio is high, and (2) when initialized randomly, HOOI tends to converge to a solution as accurate as that of STHOSVD in as few as one or two iterations, the goal of this work is to evaluate the scalability of HOOI to large tensor datasets and compare its performance with state-of-the-art implementations of STHOSVD.

One of the main limitations of HOOI is that it solves the rank-specified formulation of the Tucker approximation problem, but it does not solve the error-specified formulation. We propose in § 3.2 a rank-adaptive variant of HOOI that does solve the error-specified formulation. Our approach is based on incrementally expanding the Tucker ranks over HOOI iterations in order to satisfy the error threshold and then, once it is satisfied, truncating the ranks to maximize compression. We exploit fast computation of the approximation error of a given Tucker approximation and all its leading subtensors to determine the best truncation. Thus, prior knowledge of the output ranks is no longer required, but the choice of initial ranks affects the number of HOOI iterations performed.

TuckerMPI is a C++/MPI library that implements STHOSVD for large dense tensors [4]. We build our parallelization of HOOI on TuckerMPI, leveraging the existing functionality for the main computational kernels required of both STHOSVD and HOOI, including tensor-times-matrix (TTM) and algorithms for computing the SVD. The efficiency and scalability of HOOI is largely determined by those of the TTM and SVD kernels. We apply two key optimizations,

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759865

one for each kernel, in order to make our rank-adaptive parallel HOOI algorithm more efficient. To reduce the computational costs of the TTM kernel, we use memoization to avoid recomputation of individual TTMs that occur across subiterations of HOOI; see § 3.3. To reduce costs and expose better parallelism of SVD computations, we use subspace iteration within HOOI subiterations. While subspace iteration computes only an approximation to the leading left singular vectors, we show that one subspace iteration is sufficient to obtain the desired accuracy across the full HOOI iteration. Implementation of subspace iteration requires new parallel computational kernels in TuckerMPI, which we describe in § 3.4.

In § 4, we evaluate the efficiency and scalability of HOOI and compare it to TuckerMPI's STHOSVD. We consider synthetic test data to show how the number of modes and the compression ratios affect performance, and we demonstrate the impact of our computational optimizations in different scenarios for the rank-specified approximation problem. We also consider three real datasets generated from scientific simulation of fluid flow and combustion to test the rank-adaptivity of our algorithm. The experimental results demonstrate that HOOI generally scales as well as STHOSVD. In cases of large tensor dimension, STHOSVD becomes bottlenecked by a sequential SVD-related computation, and HOOI scales significantly better than STHOSVD at high core counts. We show that HOOI benefits from the reduction of computational cost, roughly proportional to the compression ratio in a single tensor dimension, compared to STHOSVD, but that it can suffer from lower local kernel efficiency as a result. For scenarios of high compression ratio and initial ranks that are overestimates of the output ranks, we observe that HOOI achieves Tucker approximations faster than STHOSVD, and in many cases, produces Tucker decompositions with better compression ratio.

To summarize, the main contributions of this work are

(1) parallelization of HOOI using the TuckerMPI library;[1]
(2) novel adaptation of HOOI to solve the error-specified formulation of the Tucker approximation problem;
(3) efficient memoization of the TTM computations across subiterations of HOOI;
(4) novel use of subspace iteration to reduce parallel computational cost of the SVD computations within HOOI; and
(5) demonstration of faster time-to-solution of HOOI compared to STHOSVD for scientific simulation datasets.

We conclude in § 5 that HOOI is a viable alternative to STHOSVD for solving both rank-specified and error-specified formulations of the Tucker approximation problem, and it is preferred when the compression ratio is high or individual tensor dimensions are large.

2 Background

Notation. Throughout this work, we use bold lowercase letters (e.g., $\mathbf{v}$) to denote vectors, bold uppercase letters (e.g., $\mathbf{M}$) to denote matrices, and bold script uppercase letters (e.g., $\mathcal{T}$) to denote tensors. A d-way tensor $\mathcal{T} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ has entries $\mathcal{T}_{i_1,i_2,\ldots,i_d}$. We also use the term *modes* to refer to the dimensions of a tensor.

Operations. A d-way tensor $\mathcal{T}$ can be mode-wise unfolded into a matrix in d ways; the resulting mode-j unfolding, denoted as $\mathbf{T}_{(j)}$, is formed so that the columns are the mode-j fibers of the tensor. Another major operation for tensors is the multi-TTM, which computes the product of a tensor $\mathcal{T}$ with j matrices $\{\mathbf{U}_j\}$ in up to d modes. A single TTM along mode j, denoted as $\mathcal{T} \times_j \mathbf{U}_j$, is computed as the matrix multiplication $\mathbf{U}_j\mathbf{T}_{(j)}$. The multi-TTM of $\mathcal{T}$ with d matrices $\mathcal{T} \times_1 \mathbf{U}_1 \cdots \times_d \mathbf{U}_d$ is computed as a series of TTMs. The resulting tensor can be unfolded along mode j, expressed as $\mathbf{U}_j\mathbf{T}_{(j)}(\mathbf{U}_d \otimes \cdots \otimes \mathbf{U}_{j+1} \otimes \mathbf{U}_{j-1} \otimes \cdots \otimes \mathbf{U}_1)^\intercal$. Also let $\|\cdot\|$ denote the tensor norm, which is the natural generalization of the matrix Frobenius norm.

Tucker Decomposition. The Tucker decomposition of a tensor $\mathcal{X} \in \mathbb{R}^{n_1\times\cdots\times n_d}$ for a rank $\mathbf{r} = (r_1 \times \cdots \times r_d)$ approximates $\mathcal{X}$ as the product of a core tensor $\mathcal{G} \in \mathbb{R}^{r_1\times\cdots\times r_d}$ and d factor matrices $\mathbf{U}_j \in \mathbb{R}^{n_j\times r_j}$ where $\mathcal{X} \approx \hat{\mathcal{X}} = \mathcal{G} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_d \mathbf{U}_d$. By taking $r_j = \operatorname{rank}(\mathbf{X}_{(j)})$, we can also obtain an exact representation of tensor $\mathcal{X}$. The optimal rank-$\mathbf{r}$ Tucker decomposition of $\mathcal{X}$ can be expressed as a solution to the rank-specified optimization problem

$$\begin{aligned} &\min \quad \|\mathcal{X} - \mathcal{G} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_d \mathbf{U}_d\| \\ &\text{subject to } \mathcal{G} \in \mathbb{R}^{r_1\times\cdots\times r_d}, \mathbf{U}_j \in \mathbb{R}^{n_j\times r_j} \text{ for } j = 1,\ldots,d. \end{aligned} \tag{1}$$

Alternatively, the error-specified formulation of the Tucker approximation problem is given as

$$\begin{aligned} &\min \quad \prod_{j=1}^{d} r_j + \sum_{j=1}^{d} n_j r_j \\ &\text{subject to } \mathcal{G} \in \mathbb{R}^{r_1\times\cdots\times r_d}, \mathbf{U}_j \in \mathbb{R}^{n_j\times r_j} \text{ for } j = 1,\ldots,d \\ &\text{and} \quad \|\mathcal{X} - \mathcal{G} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_d \mathbf{U}_d\| \le \epsilon\|\mathcal{X}\|. \end{aligned} \tag{2}$$

2.1 TuckerMPI's STHOSVD

Already implemented in TuckerMPI [4] prior to this work is the STHOSVD method, which approximately solves (2) by unfolding the j^{th} mode, computing its leading left singular vectors (LLSV), and performing a TTM with the singular vectors to truncate the j^{th} mode to rank r_j. Once all factor matrices have been computed, the truncated tensor has rank $\mathbf{r}$ and is the core tensor, $\mathcal{G}$, corresponding to a Tucker decomposition of $\mathcal{X}$ with approximation error $\|\hat{\mathcal{X}} - \mathcal{X}\| \le \varepsilon\|\mathcal{X}\|$, where $\hat{\mathcal{X}} = \mathcal{G} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \cdots \times_d \mathbf{U}_d$.

A relative error of ε in the Tucker decomposition can be achieved by selecting r_j in the LLSV computation such that $\sum_{i=r_j+1}^{n_j} \sigma_i^2 \le \varepsilon^2\|\mathcal{X}\|^2/d$, where σ_i is the i^{th} largest singular value of the j^{th} unfolding. That is, the LLSV computation can be performed to compute a specified number of singular vectors or to achieve a specified error; see [5] for more details. Algorithm 1 shows the steps of STHOSVD to obtain a Tucker decomposition of $\mathcal{X}$. There are several algorithms that one could choose in line 4 of Alg. 1 to compute $\mathbf{U}_j$. As described in [4], TuckerMPI computes the eigenvalue decomposition (EVD) of the Gram matrix, $\mathbf{Y}_{(j)}\mathbf{Y}_{(j)}^\intercal$. Alternative choices include QR-based approaches such as LQ decomposition of $\mathbf{Y}_{(j)}$ followed by an SVD of $\mathbf{L}$ [18] and randomized approaches such as the randomized range finder method [20, 21]. The following complexity analysis of TuckerMPI's STHOSVD follows [4].

[1] Our TuckerMPI extension is available at https://doi.org/10.5281/zenodo.16752647.

Algorithm 1 Sequentially-Truncated High-Order SVD

```
1: function [ $\mathcal{G}, \{\mathbf{U}_j\}$ ] = STHOSVD($\mathcal{X}, \varepsilon$)
2:     $\mathcal{Y} = \mathcal{X}$
3:     for $j = 1 : d$ do
4:         $\mathbf{U}_j = \text{LLSV}(\mathbf{Y}_{(j)}, \varepsilon\|\mathcal{X}\|/\sqrt{d})$        ▷ Error-specified
5:         $\mathcal{Y} = \mathcal{Y} \times_j \mathbf{U}_j^\mathsf{T}$
6:     end for
7:     $\mathcal{G} = \mathcal{Y}$
8: end function
```

STHOSVD's Computational Complexity. TuckerMPI uses P processors organized into a d-dimensional grid such that $P = (P_1 \times \cdots \times P_d)$ and that each processor stores a $1/P$ fraction of $\mathcal{X}$. Our analysis will assume $\mathcal{X} \in \mathbb{R}^{n \times \cdots \times n}$ and $\mathcal{G} \in \mathbb{R}^{r \times \cdots \times r}$ to simplify cost comparison across algorithms. Under these simplifying assumptions, the cost of LLSV in line 4 is given by

$$\sum_{j=1}^{d} \left(\frac{r^{j-1} n^{d-j+2}}{P} + O(n^3) \right) \approx \frac{n^{d+1}}{P} + O(dn^3),$$

where the first term is the cost of computing the $n \times n$ Gram matrix and the second term is the cost of sequentially computing the EVDs to leading order. After $\mathbf{U}j$ is computed, $\mathcal{Y}$ is truncated by performing the TTM in line 5, which costs

$$2 \sum_{j=1}^{d} \frac{r^j n^{d-j+1}}{P} \approx 2 \frac{rn^d}{P}.$$

Computing the Gram matrix is a factor of $n/2r$ more expensive than the TTM and is the dominant cost for $n \gg r$. Sequentially truncating $\mathcal{Y}$ leads to decreasing dimensions, so the algorithm is typically dominated by the first Gram matrix computation. Note that the EVD is not parallelized, which can be a barrier to parallel scaling when a single tensor dimension is large. We summarize the leading order STHOSVD flops cost in Tab. 1 (shown in red).

STHOSVD's Communication Complexity. TuckerMPI's parallel algorithm for LLSV explicitly forms the Gram matrix, $\mathbf{G} = \mathbf{Y}_{(j)} \mathbf{Y}_{(j)}^\mathsf{T}$, where $\mathbf{Y}_{(j)}$ is redistributed (if necessary) to a 1D column layout across P processors, and then sequentially computes the EVD of $\mathbf{G}$. After redistribution of $\mathbf{G}$, each processor computes a local Gram matrix which can be sum-reduced (or all-reduced) prior to the EVD. At iteration j, the number of entries in $\mathcal{Y}$ is $r^{j-1} n^{d-j+1}$. The Gram matrix that is computed in each mode is of size $n \times n$, so the total communication cost is dn^2 for the all-reduce. Thus, the communication cost is given by

$$\sum_{j=1}^{d} \left(\frac{r^{j-1} n^{d-j+1}}{P} \cdot \frac{P_j - 1}{P_j} + O(n^2) \right) \approx \frac{n^d}{P} \cdot \frac{P_1 - 1}{P_1} + dn^2,$$

where we assume the redistribution cost is dominated by the first mode. However, note that there is no redistribution cost in mode j if $P_j = 1$. Finally, the parallel TTM also requires communication to perform a sum-reduce of local TTM results. Since the output of the TTM is largest in the first mode (of size rn^{d-1}/P), the communication cost of TTMs to leading order cost is

$$\sum_{j=1}^{d} \frac{r^j n^{d-j}}{P} (P_j - 1) \approx \frac{rn^{d-1}}{P} (P_1 - 1).$$

Again, note there is no communication cost in mode j if $P_j = 1$. Because the largest data communicated occurs in mode 1, processor grids with $P_1 = 1$ are typically the fastest for STHOSVD (as we observe in our experiments). We summarize the STHOSVD communication costs in Tab. 2 (shown in red).

2.2 TuckerMPI's HOOI

Algorithm 2 HOOI

```
function [ $\mathcal{G}, \{\mathbf{U}_j\}$ ] = HOOI($\mathcal{X}, \mathbf{r}$)
    Initialize factor matrices $\mathbf{U}_j$, $j = 1, \ldots, d$
    while not converged do
        for $j = 1 : d$ do
            $\mathcal{Y} = \text{MULTI-TTM}(\mathcal{X}, \{\mathbf{U}_1^\mathsf{T}, \ldots, \mathbf{U}_{j-1}^\mathsf{T}, \mathbf{U}_{j+1}^\mathsf{T}, \ldots, \mathbf{U}_d^\mathsf{T}\})$
            $\mathbf{U}_j = \text{LLSV}(\mathbf{Y}_{(j)}, r_j)$        ▷ Rank-specified
        end for
    end while
    $\mathcal{G} = \mathcal{Y} \times_d \mathbf{U}_d^\mathsf{T}$
end function
```

Higher Order Orthogonal Iteration (HOOI) is given in Alg. 2 and is an alternative method for solving the rank-specified formulation of the Tucker approximation problem [12, 14, 17]. HOOI is a block coordinate descent method and can be initialized from random initial factor matrices $\{\mathbf{U}_j \in \mathbb{R}^{n_j \times r_j}\}_{j=1}^{d}$. This algorithm iteratively updates each factor matrix $\mathbf{U}_j$ by performing a TTM with all but the j^{th} factor matrix to obtain an intermediate tensor $\mathcal{Y}$ and computing the leading left singular vectors of $\mathbf{Y}_{(j)}$. The core tensor $\mathcal{G}$ can be computed once, at the end, or at the end of every iteration in order to compute the approximation error. In § 3.2, we will discuss a technique that allows HOOI to adapt ranks by performing analysis on $\mathcal{G}$ every d iterations. The following analysis of TuckerMPI's HOOI follows [1].

HOOI's Computational Complexity. Since HOOI is an iterative algorithm for Tucker decomposition, we analyze the cost of one HOOI iteration. Each HOOI iteration requires d multi-TTMs, in all modes but mode-j, and d LLSV computations to update factors matrices, in all modes. Once the factor matrices have been updated, the core tensor $\mathcal{G}$ is obtained by performing a TTM with the last factor matrix $\mathbf{U}_d$. The cost of computing d multi-TTMs is given by

$$2d \sum_{i=1}^{d} \frac{r^i n^{d-i+1}}{P} \approx 2d \frac{rn^d}{P}.$$

The cost of each TTM decreases, so the first term in the summation (i.e. the first TTM) dominates. Multiplying the cost of the first TTM by d yields the cost of d multi-TTMs (i.e. one HOOI iteration). The cost of computing LLSV is given by

$$d \frac{r^{d-1} n^2}{P} + O(dn^3),$$

where the first term is the cost of computing the Gram matrix $\mathbf{Y}_{(j)} \mathbf{Y}_{(j)}^\mathsf{T}$ and the second term is the cost of computing the EVD. Finally, the core tensor at the end of each HOOI iteration is obtained by performing a TTM in mode-d with the intermediate tensor $\mathcal{Y}$ and

$\mathbf{U}_d$, which has a cost of $2nr^d/P$ and is a lower order term. We summarize the leading order cost per HOOI iteration as implemented by TuckerMPI in Tab. 1 (shown in red).

HOOI's Communication Complexity. The communication cost of each iteration of HOOI is dominated by multi-TTMs and LLSV computations. Each TTM in the multi-TTM requires communication to perform a sum reduction to form $\mathcal{Y}$. Communication is required along the processor dimension corresponding to the mode in which a TTM is performed. The size of $\mathcal{Y}$ decreases with each TTM, so the communication cost of a multi-TTM is dominated by the first TTM. Each HOOI iteration performs d multi-TTMs, where one iteration updates the factor matrix in the first mode. The cost of communication for the multi-TTMs is given by

$$\sum_{j=1}^{d}\left(\sum_{i=1}^{j-1}\frac{r^i n^{d-i+2}}{P}(P_i-1)+\sum_{i=j+1}^{d}\frac{r^{i-1}n^{d-1+1}}{P}(P_i-1)\right)$$
$$\approx (d-1)\frac{rn^{d-1}}{P}(P_1-1)+\frac{rn^{d-1}}{P}(P_2-1).$$

The first term corresponds to the $d-1$ TTMs performed in the 1st mode and the second term corresponds to TTMs performed in the 2nd mode (for the multi-TTM in all but the 1st mode).

Communication is also required when computing the LLSV in each mode. Using the same LLSV algorithm as in STHOSVD, the Gram matrix is computed in parallel followed by a sequential EVD. Computing the Gram matrix requires an all-to-all to redistribute $\mathbf{Y}_{(j)}$ so that it is stored in 1D-column layout. After redistribution $\mathbf{Y}_{(j)}\mathbf{Y}_{(j)}^{\mathsf{T}}$ is computed in parallel by performing local matrix-matrix multiplications that are sum-reduced to obtain the Gram matrix. The cost of communication for the LLSV is given by

$$\frac{r^{d-1}n}{P}\sum_{i=1}^{d}\left(\frac{P_i-1}{P_i}\right)+dn^2,$$

where the first term is the cost of all-to-all communication and the second term is the cost of sum reduction of the Gram matrix for one HOOI iteration (i.e. d calls to LLSV). We summarize the HOOI communication costs as implemented by TuckerMPI in Table 2 (shown in red).

2.3 Other Related Work

STHOSVD Optimizations. Many previous parallelizations of the Tucker approximation algorithms have been developed. Austin et al. [1] developed the first parallel implementation of STHOSVD and HOOI for dense tensors. Chakaravarthy et al. [10] proposed optimizations to help port the computational kernels of STHOSVD to GPUs, including using randomized algorithms. Li et al. [18] propose a more numerically stable version of STHOSVD using a parallel QR-SVD and proposed running in lower working precision to achieve speed up. Minster et al. [20] introduced algorithms to perform randomized sketches with random matrices that have internal structure to reduce computation further. These algorithms were implemented in the TuckerMPI library, but we do not compare against them. One can view HOOI with initial randomization as a form of the structured random sketches from that algorithm.

HOOI Optimizations. Other related work has improved the HOOI algorithm in similar ways to our proposed algorithm. Xiao and Yang [26] propose a related mode-wise adaptive-rank strategy for HOOI that allows for both expansion and contraction of ranks across iterations, though their approach truncates mode by mode. They show their MATLAB implementation is competitive with Tensorlab's implementation [25] of STHOSVD. Sun and Huang [23] proposed HOQRI for solving the fixed-rank formulation for sparse input tensors. It uses subspace iteration without forming intermediate quantities to avoid the intermediate memory blowup problem that occurs for sparse Tucker approximations. They show their MATLAB implementation achieves better accuracy more quickly than Tensor Toolbox's implementation [2] of (sparse) HOOI. Finally, Kaya and Robert [15] study dimension tree memoization for CP-ALS (a block coordinate descent method for computing a CP decomposition) and HOOI algorithms applied to dense tensors, and they provide efficient algorithms for selecting the optimal tree structure. We use a heuristic for determining our dimension tree structure, so they may not be optimal. Chakaravarthy et al. [9] employ dimension trees and other optimizations within HOOI in a parallel setting, but they do not show improvement over contemporary parallel implementations of STHOSVD.

3 Algorithm Design

3.1 Motivation

Comparing TuckerMPI's STHOSVD and HOOI flops costs (shown in red in Tab. 1), we observe that for most problems, STHOSVD's dominant cost is n^{d+1}/P and HOOI's dominant cost is $2d\ell \cdot rn^d/P$, where ℓ is the number of HOOI iterations to achieve the same approximation error as STHOSVD. This implies that HOOI is computationally cheaper if $n/r > 2d\ell$, i.e., if the dimension reduction in each mode of the tensor is at least twice the number of dimensions times the number of HOOI iterations. As claimed in § 1 and empirically corroborated in § 4, when initialized randomly, HOOI often converges to an error comparable to STHOSVD in as few as 2 iterations. Under this assumption, the dimension reduction requirement for HOOI to be cheaper becomes $n/r > 4d$. Although such a high degree of dimension reduction has been observed for some Tucker compression problems, for reasonably large d (e.g., 4-way or 5-way tensors), it is rare. In cases of small d and large n and number of processors P, TuckerMPI's STHOSVD and HOOI can be bottlenecked by the sequential EVD computations, which are $O(dn^3)$ and $O(d\ell n^3)$ respectively. In this case, we never expect HOOI to be cheaper, which we in fact observe and discuss in § 4.1.

However, we propose two means of reducing the computational cost of HOOI iterations to make the method much more competitive with STHOSVD even in cases of smaller dimension reduction. The first optimization is dimension tree memoization of the TTMs that usually dominate the cost of HOOI iterations. As shown in § 3.3, this reduces the TTM computational cost by a factor of $d/2$, and implies that the dimension reduction requirement for HOOI to be cheaper becomes $n/r > 8$ (assuming 2 iterations). The second optimization is the use of subspace iteration to reduce the dominant cost of computing LLSV by a factor of $1/4 \cdot n/r$ and reduce the sequential computation within TuckerMPI by a factor of $O\big((n/r)^2\big)$. The details of this optimization are given in § 3.4.

Before describing the computational optimizations, we show in § 3.2 how we modify the HOOI algorithm to solve the error-specified formulation of the Tucker approximation problem. Given this modification, along with the computational optimizations, HOOI becomes much more competitive with STHOSVD and significantly outperforms it in certain cases of high dimension reduction.

Our proposed algorithm RA-HOSI-DT, with all three modifications/optimizations, is given in Alg. 3. Tables 1 and 2 summarize the cost analysis, showing the benefits of each of the two optimizations and comparing the final costs of RA-HOSI-DT with STHOSVD. As argued above, RA-HOSI-DT is computationally cheaper roughly when $n/r > 8$, and it also alleviates a $O(dn^3)$ sequential bottleneck in STHOSVD. It also requires less communication roughly when $n/r > 2(P_1 + P_d - 2)$.

Algorithm 3 Rank-Adaptive HOOI with dim. trees and sub. iter.

```
1:  function [$\mathcal{G}, \{\mathbf{U}_j\}$] = RA-HOSI-DT($\mathcal{X}, \varepsilon, \mathbf{r}, \alpha$)
2:      Initialize factor matrices $\mathbf{U}_j$, $j = 1, \ldots, d$
3:      while not converged do
4:          [$\mathcal{G}, \{\mathbf{U}_j\}$] = HOSI-DT($\mathcal{X}, \{\mathbf{U}_j\}, 1 : d, \mathbf{r}$)
5:          if $\|\mathcal{G}\|^2 \geq (1 - \varepsilon^2)\|\mathcal{X}\|^2$ then
6:              Find $\mathbf{r}$ by solving (3)
7:              $\mathcal{G} = \mathcal{G}(\mathbf{1} : \mathbf{r}), \mathbf{U}_j = \mathbf{U}_j(:, 1 : r_j)$ for $j = 1, \ldots, d$
8:          else
9:              $\mathbf{r} = \alpha\mathbf{R}$          ▷ increase ranks by constant factor
10:         end if
11:     end while
12: end function
```

3.2 Rank-Adaptive HOOI

A significant disadvantage of HOOI is that it solves only the rank-specified formulation of the Tucker approximation problem, whereas STHOSVD can adaptively select ranks based on a relative error tolerance. We propose a technique that allows HOOI to automatically adapt ranks to meet a user-specified relative error tolerance, as shown in Alg. 3. Line 4 performs an update of all factor matrices and the core using optimizations described in § 3.3 and § 3.4.

Recall that for the error-specified formulation, given an error tolerance ε and an initial rank estimate $\mathbf{r}$, our method adaptively finds a Tucker decomposition $\hat{\mathcal{X}} = [\mathcal{G}; \mathbf{U}_1, \ldots, \mathbf{U}_d]$ for a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ such that $\|\hat{\mathcal{X}} - \mathcal{X}\| \leq \varepsilon\|\mathcal{X}\|$. Whereas in clasical HOOI the core is only updated after the iterations, here we compute the core tensor at the end of every iteration and perform error analysis on it. To check the error, we use the identity that for orthonormal matrices $\mathbf{U}_1, \ldots, \mathbf{U}_d$ and $\mathcal{G} = \mathcal{X} \times_1 \mathbf{U}_1^\mathsf{T} \times \cdots \times_d \mathbf{U}_d^\mathsf{T}$, the approximation error can be written as $\|\mathcal{X} - \hat{\mathcal{X}}\|^2 = \|\mathcal{X} - \mathcal{G} \times_1 \mathbf{U}_1 \times \cdots \times_d \mathbf{U}_d\|^2 = \|\mathcal{X}\|^2 - \|\mathcal{G}\|^2$ ([5, Proposition 6.3]). If the current Tucker approximation is not sufficiently accurate, we increase all ranks by a constant factor α and perform the next HOOI iteration. The tunable parameter α trades off how many iterations are required in order to reach ranks that will satisfy the approximation error with how large the overestimate is once the error is achieved; we typically use 1.5 or 2. If the current approximation satisfies the error threshold, then we can optimize over all rank truncations by analyzing the core tensor's entries. Specifically, we solve the optimization problem

$$\min_{\mathbf{r}} \quad \Pi_{j=1}^{d} r_j + \Sigma_{j=1}^{d} n_j r_j, \qquad \text{subject to } \|\mathcal{G}(\mathbf{1} : \mathbf{r})\|^2 \geq (1 - \varepsilon^2)\|\mathcal{X}\|^2. \tag{3}$$

This computes the leading subtensor of $\mathcal{G}$ that minimizes the size of the Tucker approximation and also satisfies the error threshold. Note that any subtensor of the core, along with the corresponding columns of the factor matrices, is a valid Tucker approximation with error determined by the norm of the core subtensor. The optimal subtensor need not be a leading one, but we order factor matrix columns to concentrate the weight of $\mathcal{G}$ towards the entry of smallest index value so that the heuristic of searching over only leading subtensors is reasonable.

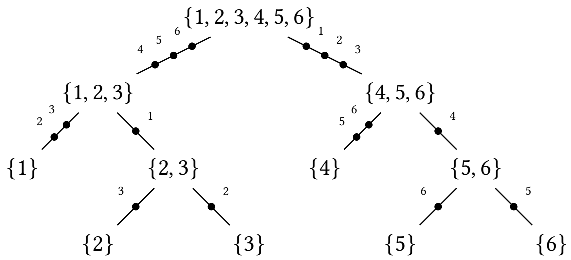

Figure 1: Illustration of multi-TTM memoization for an order-6 tensor. Each node in the tree shows the set of modes in which multiplication has not been performed. Each notch in an edge is a TTM in the labeled mode. Factor matrices are computed at each leaf node in the mode shown. $\mathcal{G}$ is updated in the last leaf node.

Core Analysis Computational Complexity. The cost of one RA-HOOI iteration is the same as one iteration of HOOI given in Tab. 1, but with the possible additional cost of performing analysis on the core tensor $\mathcal{G}$ to adapt the ranks for the next iteration. We solve the optimization problem given in eq. (3) exhaustively by computing the norm and corresponding size of every leading subtensor. This can be done using only $O(dr^d)$ operations by employing a multidimensional prefix sum computation across the squares of the core entries. Because computational cost tends to be dominated by the rest of the HOOI iteration, we perform the core analysis sequentially, though the prefix sums are readily parallelizable.

Core Analysis Communication Complexity. At the end of a HOOI iteration, $\mathcal{G}$ is distributed across all processors, so it must be gathered on a single processor in order to perform analysis. Since the entire core tensor must be communicated, the all-gather cost is r^d per HOOI iteration. We demonstrate in § 4 that the sequential cost of core analysis is typically negligible.

3.3 Dimension Tree Memoization

Adapting ranks in each HOOI iteration is a low order cost, however, the cost of TTMs is a factor of d more expensive than in STHOSVD. We can reduce the cost of TTMs by avoiding redundant computations. Notice that for $j = 1$ in Alg. 2 the following multi-TTM is computed $\mathcal{Y} = \mathcal{X} \times_2 \mathbf{U}_2^\mathsf{T} \times_3 \mathbf{U}_3^\mathsf{T} \ldots \times_d \mathbf{U}_d^\mathsf{T}$. At $j = 2$ the multi-TTM is $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}_1^\mathsf{T} \times_3 \mathbf{U}_3^\mathsf{T} \ldots \times_d \mathbf{U}_d^\mathsf{T}$. By comparing the two multi-TTMs we can see that $d - 2$ TTMs are the same (namely 3 to d). So we can reuse results from one multi-TTM to the next by memoizing

Algorithm	LLSV		TTM		Core Analysis
HOOI iteration	Gram + Eig	$d\frac{n^2r^{d-1}}{P} + O(dn^3)$	Direct	$2d\frac{rn^d}{P}$	$O(dr^d)$
	Sub. Iter.	$4d\frac{nr^d}{P} + O(dnr^2)$	Dim. Tree	$4\frac{rn^d}{P}$	
STHOSVD	$\frac{n^{d+1}}{P} + O(dn^3)$		$2\frac{rn^d}{P}$		-
RA-HOSI-DT	$\ell\left(4d\frac{nr^d}{P} + O(dnr^2)\right)$		$\ell\left(4\frac{rn^d}{P}\right)$		$\ell\left(O(dr^d)\right)$

Table 1: Leading order flops costs of LLSV (Gram + Eig and Subspace Iteration), multi-TTM (Direct and Dimension Trees) and Core Analysis algorithmic choices for HOOI and a comparison between STHOSVD and HOOI with Subspace Iteration and Dimension Trees (HOSI-DT) optimizations. We assume ℓ iterations of HOSI-DT are performed.

Algorithm	LLSV		TTM		Core Analysis
HOOI iteration	Gram + Eig	$\frac{nr^{d-1}}{P}\sum_{i=1}^{d}\frac{P_i-1}{P_i} + dn^2$	Direct	$(d-1)\frac{rn^{d-1}}{P}(P_1-1) + \frac{rn^{d-1}}{P}(P_2-1)$	r^d
	Sub. Iter.	$\frac{r^d}{P}\sum_{i=1}^{d}(P_i-1) + 2dnr$	Dim. Tree.	$\frac{rn^{d-1}}{P}(P_1-1) + \frac{rn^{d-1}}{P}(P_d-1)$	
STHOSVD	$\frac{n^d}{P}\frac{P_1-1}{P_1} + dn^2$		$\frac{rn^{d-1}}{P}(P_1-1)$		-
RA-HOSI-DT	$t\left(\frac{r^d}{P}\sum_{i=1}^{d}(P_i-1) + 2dnr\right)$		$t\left(\frac{rn^{d-1}}{P}(P_1+P_d-2)\right)$		$\ell\left(r^d\right)$

Table 2: Leading order bandwidth costs of LLSV (Gram + Eig and Subspace Iteration), multi-TTM (Direct and Dimension Trees) and Core Analysis algorithmic choices for HOOI. For reference, we include a comparison between STHOSVD and HOOI with Subspace Iteration and Dimension Trees (HOSI-DT). We assume a processor grid of $P = (P_1 \times \cdots \times P_d)$ and that ℓ iterations of HOSI-DT are performed.

intermediate results. This idea, organized using so-called "dimension trees," was first used in the context of CP decompositions [22] and has been applied to Tucker computations as well [15, 20]. Figure 1 shows an example dimension tree as we implement them for an order-6 tensor where each node represents the set of modes in which a TTM has not been performed. At the root of the tree, no TTMs have been performed, so the tensor is $\mathcal{X}$. Each notch in an edge of the tree represents a TTM in the labeled mode. At each leaf node, TTMs in all modes but one have been performed, so we update the factor matrix in that mode by performing LLSV. The core tensor $\mathcal{G}$ is updated at the last leaf node by perform a TTM between the (memoized) intermediate tensor and the factor matrix corresponding to the last leaf node. Algorithm 4 shows the HOOI iteration using dimension tree memoization implemented recursively.

Dimension Tree Computational Complexity. The flops cost of performing multi-TTMs using dimension trees is given by

$$4\sum_{i=1}^{d/2}\frac{r^i n^{d-i+1}}{P} + O\left(d\sum_{i=d/2+1}^{d} r^i n^{d-i+1}\right) \approx 4\frac{rn^d}{P},$$

where the first term is the cost of computing the TTMs in the first two branches (left and right of the root) in the dimension tree and the second term is the cost of computing the TTMs in all remaining branches. The largest TTMs in the first two branches dominate, so the cost of multi-TTMs is $4 \cdot rn^d/P$ (i.e. the first TTM in each branch), which is a factor of $d/2$ improvement over computing multi-TTMs directly. This cost is summarized in Table 1.

Dimension Tree Communication Complexity. Since the first TTM in each of the two multi-TTMs off the root dominate, the communication cost of multi-TTMs is given by

$$\sum_{i=1}^{d/2}\frac{r^i n^{d-i-1}}{P}\left(P_i - 1 + P_{d-i+1} - 1\right) \approx \frac{rn^{d-1}}{P}\left(P_1 + P_d - 2\right).$$

Algorithm 4 Recursive HOOI iteration via dimension trees

```
function [ $\mathcal{G}$, $\{\mathbf{U}_j\}$] =HOSI-DT($\mathcal{X}$, $\{\mathbf{U}_j\}$, $\mathbf{m}$, $\mathbf{r}$)
    if length($\mathbf{m}$) = 1 then
        $\mathbf{U}_m$ = LLSV-SI($\mathbf{X}_{(m)}$, $\mathbf{U}_m$, $r_m$)
        if $m = d$ then
            $\mathcal{G} = \mathcal{X} \times_m \mathbf{U}_m^\mathsf{T}$
        end if
    else
        Partition $\mathbf{m} = [\mu, \eta]$
        $\mathcal{X} = \mathcal{X} \bigtimes_{j\in\mu} \mathbf{U}_j^\mathsf{T}$
        [$\mathcal{G}$, $\{\mathbf{U}_j\}$] = HOSI-DT($\mathcal{X}$, $\{\mathbf{U}_j\}$, $\eta$, $\mathbf{r}$)
        $\mathcal{X} = \mathcal{X} \bigtimes_{j\in\eta} \mathbf{U}_j^\mathsf{T}$
        [$\mathcal{G}$, $\{\mathbf{U}_j\}$] = HOSI-DT($\mathcal{X}$, $\{\mathbf{U}_j\}$, $\mu$, $\mathbf{r}$)
    end if
end function
```

When traversing the right branch in the dimension tree shown in Fig. 1, TTMs are performed in the first $d/2$ modes starting with mode 1. The communication cost associated with TTMs in the right branch is the cost of a reduce-scatter on local data of size $rn^{d-1}/P \cdot (P_1 - 1)$, which yields the first term. The second term is due to the communication cost associated with traversing the left branch in Fig. 1. TTMs in the left branch are performed in the last $d/2$ modes starting with mode d. We perform left branch TTMs in reverse order because the mode d TTM achieves higher local TTM performance due to the layout of the local tensor in memory. The communication cost associated with TTMs in the left branch is the same as the first term, except that the reduce-scatter is performed in the P_d processor grid dimension. Therefore, processor grids with $P_1 = P_d = 1$ are typically the fastest for HOOI algorithms employing the dimension tree optimization (as we observe in our experiments).

As shown in Tabs. 1 and 2, introducing dimension trees memoization reduces the flops cost of TTMs in HOOI by a factor of $d/2$ and the communication cost by a factor of $d - 1$ in the first term.

3.4 Subspace Iteration

So far, we have assumed that the LLSVs of a matrix $\mathbf{A}$ are obtained as the eigenvectors of the Gram matrix, $\mathbf{A}\mathbf{A}^\intercal$. The next algorithmic improvement we introduce is to compute the leading left singular vectors by using subspace iteration. Algorithm 5 shows a single subspace iteration, but in principle, the computations could be repeated to improve accuracy.

Algorithm 5 LLSV via Subspace Iteration

```
1: function $\mathbf{Q}$ = LLSV-SI($\mathbf{A}, \mathbf{U}, r$)
2:     $\mathbf{G} = \mathbf{U}^\intercal \mathbf{A}$
3:     $\mathbf{Z} = \mathbf{A}\mathbf{G}^\intercal$
4:     $[\mathbf{Q}, \sim, \sim]$ = QRCP($\mathbf{Z}$)        ▷ QR with column pivoting
5: end function
```

We note that the input matrix $\mathbf{A}$ is $\mathbf{Y}_{(j)}$ from Alg. 2 or $\mathbf{X}_m$ from Alg. 3, which is the result of an all-but-one multi-TTM, and the input matrix $\mathbf{U}$ is the factor matrix from the previous HOOI iteration. This implies that the temporary matrix $\mathbf{G}$ in Alg. 5 is an unfolding of the core tensor corresponding to the current set of factor matrices. That is, the matrix multiplication in line 2 is a TTM, which we implement using existing TuckerMPI subroutines. The multiplication in line 3 is a tensor contraction in all modes but one between the core tensor and the result of an all-but-one multi-TTM, which is not implemented in TuckerMPI. Our parallel algorithm mimics the computation of the Gram matrix of a tensor unfolding, but it is a nonsymmetric operation and has different costs. Finally, we perform QR with column pivoting in line 4 to orthonormalize the subspace iteration result and also order the columns to aid in core analysis, which is discussed in § 3.2. We choose to do only a single subspace iteration because we use an accurate initialization (from the previous HOOI iteration) and because high accuracy of a HOOI subiteration is less of a priority than high accuracy of the full HOOI iteration.

Subspace Iteration Computational Complexity. Each subspace iteration requires two matrix-matrix multiplications and one QR decomposition. The first matrix-multiplication corresponds to the TTM $\mathcal{G} = \mathcal{Y} \times_j \mathbf{U}_j^\intercal$ (in the notation of Alg. 2) and the second computes the tensor contraction $\mathbf{Y}_{(j)}\mathbf{G}_{(j)}^\intercal$. The total computational cost of performing the TTM and contraction in each HOOI iteration is $4d \cdot nr^d/P$. The cost of the QR decomposition of the matrix $\mathbf{Z} \in \mathbb{R}^{n \times r}$ in each HOOI iteration is $O(dnr^2)$, where we assume a sequential QR decomposition. The total cost of performing subspace iteration in each mode across an entire HOOI iteration is given by

$$4d\frac{nr^d}{P} + O(dnr^2).$$

As shown in Tab. 1, the cost of LLSV using subspace iteration is a factor of $1/4 \cdot n/r$ cheaper than the cost of LLSV via the Gram matrix. When comparing the sequential EVD to the sequential QR decomposition, the cost of the latter is a factor of $O\left(\left(n/r\right)^2\right)$ faster.

Subspace Iteration Communication Complexity. Subspace iteration requires communication in the TTM, tensor contraction, and QR decomposition in each mode. The communication cost of the TTM is given by $r^d/P \cdot (P_j - 1)$, where P_j corresponds to the number of processors in the j^{th} mode. The tensor contraction requires redistribution of both tensors via all-to-all communication steps. However, the all-to-all cost is a lower order term since it is a factor of P_j cheaper than the communication cost associated with the TTM. Once the contraction is performed, a sum reduction followed by a broadcast is required to ensure that all processors can independently compute local QR decompositions. The communication cost of the QR decomposition is given by $2nr$ since $\mathbf{Z} \in \mathbb{R}^{n \times r}$ and must be communicated twice. As shown in Tab. 2, the total communication cost of the LLSV calls within an iteration of HOOI using subspace iteration is given by

$$\frac{r^d}{P}\sum_{j=1}^{d}\left(P_j - 1\right) + 2dnr.$$

4 Results

This section presents a comparison of the running time (strong scaling and running time breakdown) and compression (error vs. time and error vs. compression ratio) performance of the various Tucker algorithms presented in this work. All algorithms were implemented using the TuckerMPI (C++/OpenMPI) library [4].

Computing platform. Our experiments were conducted on NERSC Perlmutter (CPU partition). The system consists of 3072 compute nodes with dual-socket AMD EPYC 7763 64-core CPUs. Each socket has 4 Non-Uniform Memory Access (NUMA) regions for a total of 8 NUMA regions per node. Each NUMA region has 64 GB of DRAM memory, therefore each CPU socket has 256 GB of DRAM for, a total of 512 GB of memory per node.

Experiments. We perform experiments on synthetic tensors that are randomly generated and tensors obtained from real applications. We use 3-way and 4-way tensors for the synthetic experiments, and three real datasets: Miranda [27] (3-way), HCCI [7] (4-way), and SP [16] (5-way). The real datasets are described in more detail in § 4.2.1 and 4.2.2. Experiments performed on synthetic tensors are performed in single precision, while experiments on real datasets are performed in single or double precision depending on their storage precision on disk. Strong scaling experiments are performed on the synthetic tensors. We show running time breakdown of both real and synthetic experiments. For synthetic tensors we show the running time breakdown at small and large scale to highlight how each step in a given algorithm scales. For real tensors we vary the error tolerance and starting ranks to show how performance breakdowns vary. Compression performance experiments are performed only on the real datasets.

Even for a fixed number of processors P, the d-way processor grid has a significant effect on all algorithms. As described in § 3, STHOSVD benefits from processor grids with $P_1 = 1$, and HOOI variants using dimension trees are theoretically more efficient when $P_1 = P_d = 1$. In addition, for modes with small tensor dimension, a large processor dimension in that mode may cause load imbalance due to uneven division. In all experiments, we test all algorithms on a variety of grids, including those we expect to benefit individual algorithms, and we report the fastest observed running times.

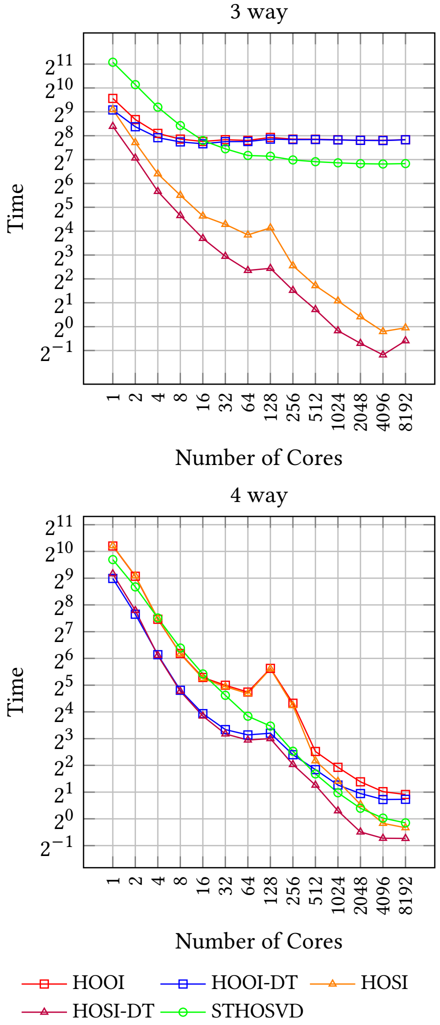

Figure 2: Strong scaling comparison of Tucker algorithms in single precision using a 3-way 3750 × 3750 × 3750 input tensor (top) and a 4-way 560 × 560 × 560 × 560 input tensor (bottom).

4.1 Strong Scaling on Synthetic Tensors

First, we present strong scaling experiments on the 3-way and 4-way synthetic tensors to demonstrate the parallel scaling of HOOI, HOOI-DT, HOSI, HOSI-DT, and STHOSVD. We choose tensor dimensions to maximize the size of the tensor that can fit on a single node (in single precision).

For synthetic input, we generate tensors by forming a Tucker-format tensor of specified rank and adding a specified level of noise. Thus, these experiments are performed for the rank-specified formulation of the Tucker approximation problem to recover the input. We run for two iterations for each variant of HOOI even though we often have a sufficiently accurate approximation after a single iteration. We include overhead due to core analysis for the error-specified formulation in the experiments on the real datasets. The largest 3-way tensor that fits into single-node memory is a tensor of size $3750 \times 3750 \times 3750$. We generate this tensor to have a rank of 30 in all modes. Similarly, we construct the 4-way tensor of size $560 \times 560 \times 560 \times 560$ with Tucker ranks $(10, 10, 10, 10)$.

Figure 2 shows the strong scaling results of the HOOI variants and STHOSVD on up to 4096 cores for the 3-way and 4-way synthetic datasets. We observe that STHOSVD scales well to 64 cores, attaining a speedup of 15.2× over the single core STHOSVD run. STHOSVD continues to scale up to 2048 cores, but achieves only a modest speedup of 1.3× over the 64 core run. This is due to TuckerMPI's limitation of having a sequential EVD implementation. In contrast, the 4-way STHOSVD strong scaling experiment shows good scaling up to 8192 cores, achieving a speedup of 937× over the single core run. This difference in STHOSVD performance is explained by the tensor dimension: a sequential EVD of a matrix of dimension 560 does not become the bottleneck until P is large.

When comparing the two HOOI variants (which use Gram SVD), we observe that HOOI-DT yields a sequential speedup of 1.4× over HOOI's direct TTM implementation for the 3-way tensor. For the 4-way tensor, HOOI-DT achieves a sequential speedup of 5.4× faster than HOOI. When comparing parallel scaling in the 3-way case, we see that HOOI and HOOI-DT scale to 16 cores with a speedup of 3.5× and 2.8×, respectively, over their single core runs. However, neither variant scales beyond 16 cores for the 3-way tensor because of the sequential EVD bottlenecks. For the 4-way tensor, HOOI and HOOI-DT scale to 8192 cores with a speedup of 629× and 346×, respectively, over their single core runs. The performance of HOOI and HOOI-DT degrades at 128 cores (single node) because both variants are memory-bandwidth bound, and we saturate bandwidth at 64 cores. HOOI and HOOI-DT continue scaling beyond 128 cores (multi-node scaling) because memory bandwidth increases. As can be seen in the 4096 core plots of Fig. 3, HOOI and HOOI-DT suffer from the problem of the sequential EVD, and they are approximately twice as slow as STHOSVD because they do twice as many EVDs over two iterations.

HOSI and HOSI-DT show significantly better scaling on the 3-way tensor when compared to STHOSVD and the HOOI variants because of the difference in LLSV subroutines. HOSI-DT achieves sequential speedups of 6.5× and 1.7× over STHOSVD and HOOI-DT, respectively. The HOSI variants scale to 4096 cores with HOSI-DT achieving significant parallel speedups of 259× and 515× over STHOSVD and HOOI-DT, respectively. HOSI-DT is also the fastest Tucker variant for the 4-way experiment attaining speedups of 1.5× and 2.9× over STHOSVD and HOOI-DT, respectively when comparing the best running times of each algorithm. HOSI and HOSI-DT exhibit similar memory bandwidth scaling behavior as the HOOI variants where performance degrades at 128 cores (single node) and continues to scale beyond 128 cores (multi-node scaling).

4.2 Performance on Simulation Datasets

We turn our focus for the error-specified comparison of our best algorithm, HOSI-DT, and the state-of-the-art, STHOSVD. The data sets are decomposed using three error tolerances; 0.1 ("high compression"), 0.05 ("mid compression"), and 0.01 ("low compression"). Furthermore, we showcase HOSI-DT through three different types of starting ranks for each error tolerance. Perfect starting ranks are the same as the final ranks of STHOSVD given the maximum relative error threshold. We overshoot and undershoot the same starting ranks by 25% above and below to force our algorithm to respectively increase and decrease ranks on the first iteration. We

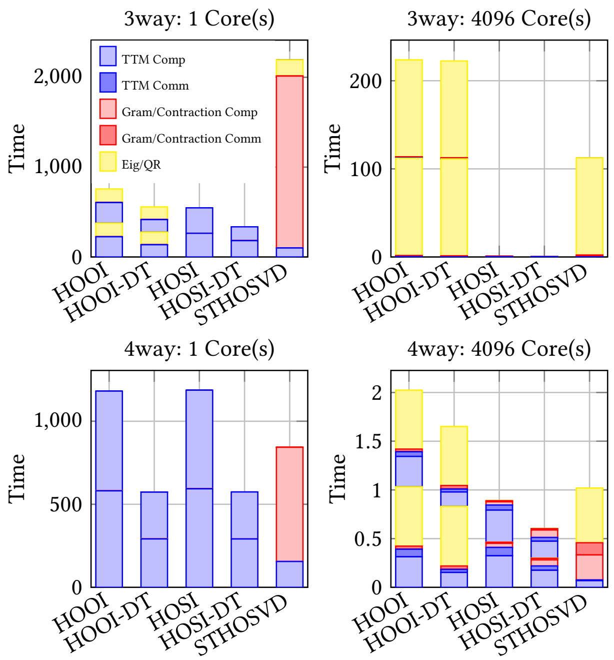

Figure 3: Running time breakdown for the synthetic 3-way (top) and 4-way (bottom) tensors.

cap the number of iterations for HOSI-DT at 3. Though all three iterations are shown in the Error vs Time and Error vs Size plots, the running time breakdown plots show the breakdown only for however many iterations it took for HOSI-DT to reach the desired error threshold. For example, the top right plot of Fig. 7 it can be seen that the HOSI-DT (Over) 0.1 threshold reached the desired at the first iteration, so we don't show the breakdown for the second iteration despite its total time being shown on Fig. 6.

4.2.1 Miranda (3-way). The Miranda dataset is a three-dimensional simulation data of the density ratios of non-reacting flow of viscous fluids [27]. Each of its dimensions is 3072, and it is stored in single precision requiring 115 GB. Our experiments use 1024 cores (8 nodes) for all algorithms.

Figure 4 demonstrates that for all error tolerances, three iterations of HOSI-DT combined is faster than STHOSVD. But as mentioned earlier, we focus on the least amount of iterations required to reach the desired error threshold. It is in high- and mid-compression where we find the most speedup. Precisely, perfect ranks achieve speedups of 82× for high-compression and 25× for mid-compression, undershooting the ranks achieves speedups of 91× for high-compression and 35× for mid-compression, and overshooting the ranks achieves speedups of 156× for high-compression and 47× for mid-compression. Low-compression is the first scenario where we observe nonnegligible costs of the core analysis subroutine. For high-compression, the best relative compression ratio is 69% which occurs at perfect ranks, mid-compression achieves a 10% improvement using perfect ranks, and low-compression has better compression at 6% when underestimating the ranks.

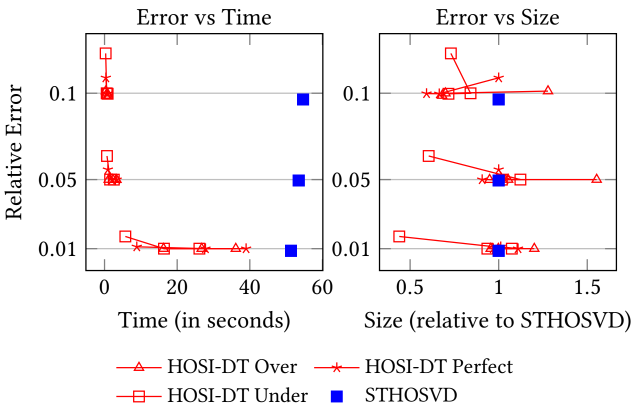

Figure 4: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the Miranda dataset using 1024 cores.

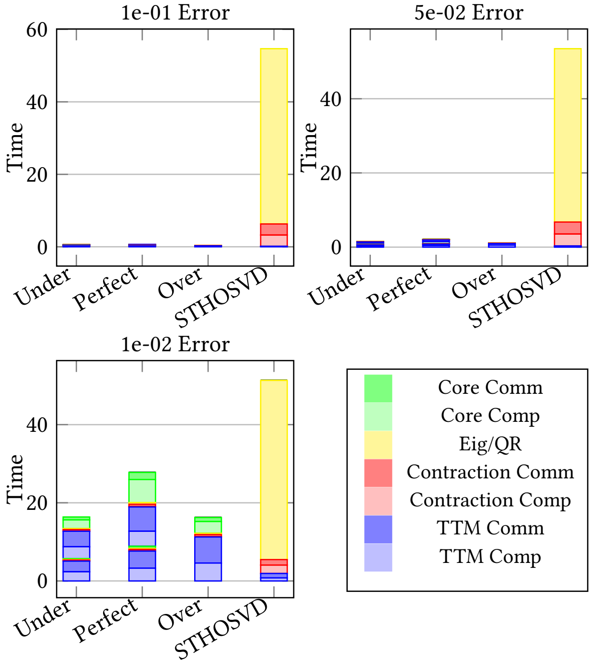

Figure 5: Running time breakdown for the Miranda dataset using 1024 cores under different levels of compression.

4.2.2 HCCI (4-way) and SP (5-way). We combine the discussion of the HCCI and SP datasets results, as the results are qualitatively similar. The Homogeneous Charge Compression Ignition (HHCI) dataset is generated from a numerical simulation of combustion [7]. The dimension of the 4-way dataset is 672 × 672 × 33 × 626 stored in double precision for a total of 75 GB. Thus, we can fit it on a single node and use all 128 cores. The first two modes are spatial dimensions, the third mode corresponds to 33 variable, and the fourth mode corresponds to time steps. The SP dataset is generated from the simulation of a statistically stationary planar methane-air flame [16]. This 5-way dataset has dimensions 500 × 500 × 500 × 11 × 400 stored in double precision and requires 4.4 TB in storage. For these experiments, we use 2048 cores (16 nodes). The first three modes are spatial dimensions, the fourth mode corresponds to 11 variables, and the last mode corresponds to time steps.

In the case where we are dominated by the TTMs, the comparisons between HOSI-DT and STHOSVD are less extreme. Figure 6

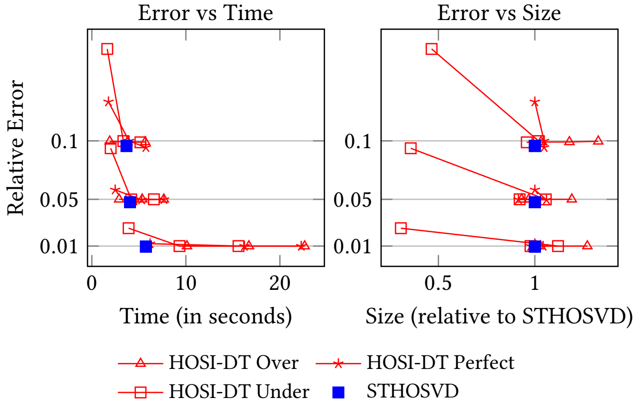

Figure 6: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the HCCI dataset using 128 cores.

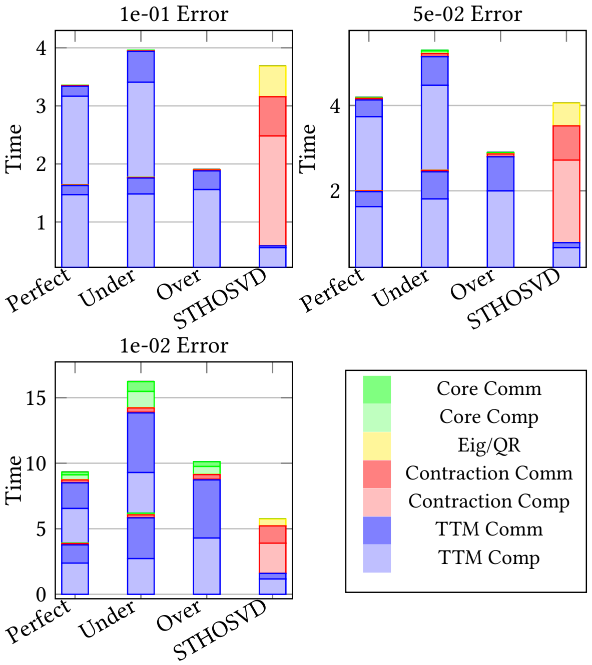

Figure 7: Running time breakdown for the HCCI dataset using 128 cores under different levels of compression.

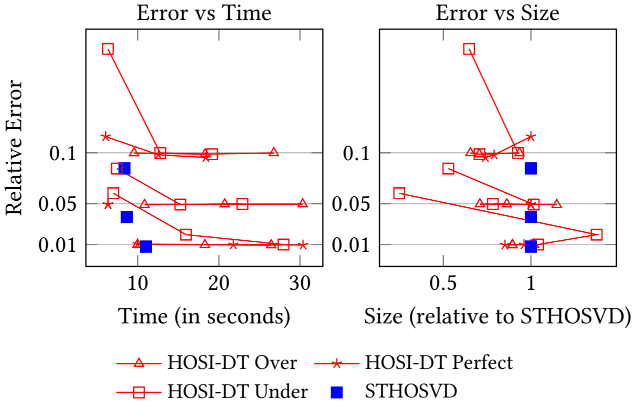

Figure 8: Progression of time, error, and relative size over 3 iterations of rank-adaptive HOSI-DT on the SP dataset using 2048 cores.

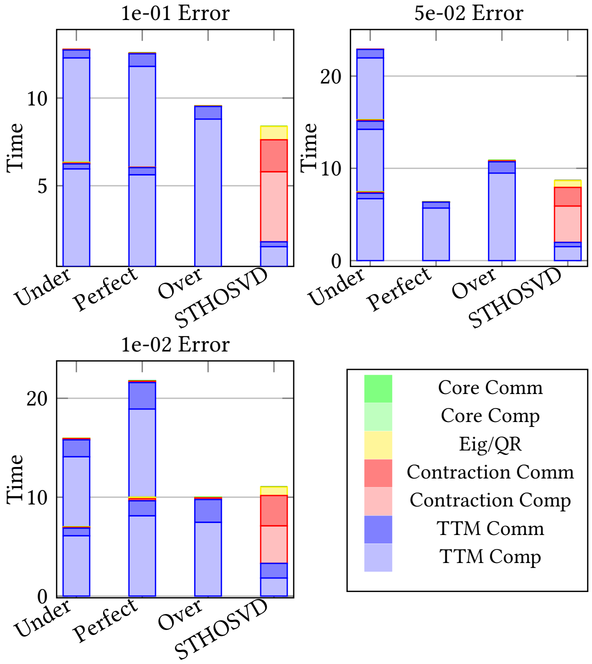

Figure 9: Running time breakdown for the SP dataset using 2048 cores under different levels of compression.

shows that on low-compression, STHOSVD is faster than any of the starting ranks of HOSI-DT to get to the desired threshold. However, for high- and mid-compression HOSI-DT achieves speedups when overshooting the ranks, specifically 1.9× for high-compression and 1.4× for low-compression, neither of which achieved better compression. Figure 7 shows the breakdown times of these speedups. However, HODI-DT achieves better compression with perfect and under ranks for all error tolerances, but always requiring three iterations to do so.

Figure 8 shows that we can typically obtain better compression after three iterations. For example, overestimating the ranks for low compression yields a speedup of 1.1× after 1 iteration, but we do not obtain better compression. Similar to HCCI, three iterations produces a smaller Tucker approximation but takes over twice as long. However, for high compression, starting from perfect and underestimates of the ranks achieve a 27% and 8% improvement on compression over STHOSVD after two iterations, respectively. In another example, Fig. 9 shows that when starting from perfect estimates of the ranks for mid compression, HOSI-DT gets the desired error tolerance and same compression ratio in less time than STHOSVD, with HOSI-DT achieving a 1.4× speedup.

5 Conclusion

Based on the complexity analysis and experimental results, we conclude that our parallel RA-HOSI-DT computes Tucker approximations of comparable error in less time than TuckerMPI's implementation of STHOSVD in two important scenarios: (1) when large individual tensor dimensions create sequential EVD bottlenecks, and (2) when individual ratios between input tensor and core tensor dimensions are large. In the first case, because of the scalability of RA-HOSI-DT, we observe very large speedups with large P. In the second case, our theoretical analysis suggests a speedup roughly proportional to n/r. However, we observe that while the number of flops is reduced compared to STHOSVD, the local matrix computation performance degrades because the smallest matrix dimension in the computation becomes r instead of n. That is, if the ranks are very small, then local matrix computations with RA-HOSI-DT run far below peak processor performance and are instead limited by the memory bandwidth. This memory bandwidth bottleneck is the reason RA-HOSI-DT loses scalability when using all cores on a single node and is the main reason the theoretical computational cost analysis doesn't match empirical performance at scale.

RA-HOSI-DT requires an input estimate of the final core ranks. While priori knowledge is not required, we observe that slight overestimates of the final ranks yield sufficiently accurate solutions often in the first iteration. When ranks are underestimated, HOOI must iterate until an overestimate is discovered, after which a single iteration yields convergence.

Furthermore, in solving the error-specified optimization problem, we highlight that RA-HOSI-DT often identifies Tucker approximations with better compression ratios than STHOSVD. This is due in large part to the flexibility afforded by the RA-HOSI-DT core analysis step to shift ranks across modes to maximize overall compression, as opposed to STHOSVD, which makes greedy decisions at each mode. If compression ratio is more important than time, taking more HOOI iterations can help to improve accuracy and often reduce ranks further.

Acknowledgments

The authors would like to thank John Billos, who contributed significantly to MATLAB implementations of the algorithms in the early stages of this project. This work is supported by the National Science Foundation under grant CCF-1942892. This material is based upon work supported by the US Department of Energy, Office of Science, Advanced Scientific Computing Research program under award DE-SC-0023296.

References

[1] 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.* 912–922. doi:10.1109/IPDPS.2016.67

[2] Brett W. Bader, Tamara G. Kolda, et al. 2023. MATLAB Tensor Toolbox Version 3.6. Available online. https://www.tensortoolbox.org

[3] Wouter Baert and Nick Vannieuwenhoven. 2023. Algorithm 1036: ATC, An Advanced Tucker Compression Library for Multidimensional Data. *ACM Trans. Math. Software* 49, 2 (jun 2023), 1–25. doi:10.1145/3585514

[4] 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. Software* 46, 2, Article 13 (June 2020), 31 pages. doi:10.1145/3378445

[5] Grey Ballard and Tamara G. Kolda. 2025. *Tensor Decompositions for Data Science.* Cambridge University Press. https://tensortextbook.com

[6] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2020. TTHRESH: Tensor Compression for Multidimensional Visual Data. *IEEE Transactions on Visualization and Computer Graphics* 26, 9 (2020), 2891–2903. doi:10.1109/TVCG.2019.2904063

[7] Ankit Bhagatwala, Jacqueline H. Chen, and Tianfeng Lu. 2014. Direct numerical simulations of HCCI/SACI with ethanol. *Combustion and Flame* 161, 7 (2014), 1826–1841. doi:10.1016/j.combustflame.2013.12.027

[8] Venkatesan Chakaravarthy, Jee Choi, Douglas Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, and Dheeraj Sreedhar. 2017. On Optimizing Distributed Tucker Decomposition for Dense Tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 1038–1047. doi:10.1109/IPDPS.2017.86

[9] Venkatesan Chakaravarthy, Jee Choi, Douglas Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, and Dheeraj Sreedhar. 2017. On optimizing distributed Tucker decomposition for dense tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 1038–1047. doi:10.1109/IPDPS.2017.86

[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* (Dallas, Texas) *(SC '18).* IEEE Press, Piscataway, NJ, USA, Article 42, 11 pages. http://dl.acm.org/citation.cfm?id=3291656.3291712

[11] Saibal De, Hemanth Kolla, Antoine Meyer, Eric T. Phipps, and Francesco Rizzi. 2024. Hybrid Parallel Tucker Decomposition of Streaming Data. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Zurich, Switzerland) *(PASC '24).* Association for Computing Machinery, New York, NY, USA, Article 20, 12 pages. doi:10.1145/3659914.3659934

[12] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. On the best rank-1 and rank-$(r_1, r_2, ..., r_n)$ approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications* 21, 4 (2000), 1324–1342. doi:10.1137/S0895479898346995

[13] Wolfgang Hackbusch. 2019. *Tensor Spaces and Numerical Tensor Calculus* (2nd ed.). Springer International Publishing. doi:10.1007/978-3-030-35554-8

[14] Arie Kapteyn, Heinz Neudecker, and Tom Wansbeek. 1986. An approach to n-mode components analysis. *Psychometrika* 51 (1986), 269–275. doi:10.1007/BF02293984

[15] Oguz Kaya and Yves Robert. 2019. Computing dense tensor decompositions with optimal dimension trees. *Algorithmica* 81 (2019), 2092–2121. doi:10.1007/s00453-018-0525-3

[16] Hemanth Kolla, Xin-Yu Zhao, Jacqueline H. Chen, and N. Swaminathan. 2016. Velocity and Reactive Scalar Dissipation Spectra in Turbulent Premixed Flames. *Combustion Science and Technology* 188, 9 (2016), 1424–1439. doi:10.1080/00102202.2016.1197211

[17] Pieter M Kroonenberg and Jan De Leeuw. 1980. Principal component analysis of three-mode data by means of alternating least squares algorithms. *Psychometrika* 45 (1980), 69–97. doi:10.1007/BF02293599

[18] Zitong Li, Qiming Fang, and Grey Ballard. 2021. Parallel Tucker Decomposition with Numerically Accurate SVD. In *50th International Conference on Parallel Processing (ICPP '21).* ACM, New York, NY, USA, 11. doi:10.1145/3472456.3472472

[19] Linjian Ma and Edgar Solomonik. 2022. Accelerating alternating least squares for tensor decomposition by pairwise perturbation. *Numerical Linear Algebra with Applications* e2431 (2022), 1–33. doi:10.1002/nla.2431

[20] Rachel Minster, Zitong Li, and Grey Ballard. 2024. Parallel Randomized Tucker Decomposition Algorithms. *SIAM Journal on Scientific Computing* 46, 2 (2024), A1186–A1213. doi:10.1137/22m1540363

[21] Rachel Minster, Arvind K. Saibaba, and Misha E. Kilmer. 2020. Randomized Algorithms for Low-Rank Tensor Decompositions in the Tucker Format. *SIAM Journal on Mathematics of Data Science* 2, 1 (2020), 189–215. doi:10.1137/19M1261043

[22] Anh-Huy Phan, Petr Tichavsky, and Andrzej Cichocki. 2013. Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations. *IEEE Transactions on Signal Processing* 61, 19 (Oct 2013), 4834–4846. doi:10.1109/TSP.2013.2269903

[23] Yuchen Sun and Kejun Huang. 2022. HOQRI: Higher-Order QR Iteration for Scalable Tucker Decomposition. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 3648–3652. doi:10.1109/ICASSP43922.2022.9746726

[24] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. 2012. A New Truncation Strategy for the Higher-Order Singular Value Decomposition. *SIAM Journal on Scientific Computing* 34, 2 (2012), A1027–A1052. doi:10.1137/110836067

[25] Nico Vervliet, Otto Debals, Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. 2016. Tensorlab 3.0. http://www.tensorlab.net

[26] Chuanfu Xiao and Chao Yang. 2024. RA-HOOI: Rank-adaptive higher-order orthogonal iteration for the fixed-accuracy low multilinear-rank approximation of tensors. *Applied Numerical Mathematics* 201, C (July 2024), 290–300. doi:10.1016/j.apnum.2024.03.004

[27] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In *IEEE International Conference on Big Data.* 2716–2724. doi:10.1109/BigData50022.2020.9378449

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

$\mathbf{C_1}$ Parallelization of HOOI using the TuckerMPI library.

$\mathbf{C_2}$ Novel adaptation of HOOI to solve the error-specified formulation of the Tucker approximation problem.

$\mathbf{C_3}$ Efficient memoization of the TTM computations across subiterations of HOOI.

$\mathbf{C_4}$ Novel use of subspace iteration to reduce parallel computational cost of the SVD computations within HOOI.

$\mathbf{C_5}$ Demonstration of faster time-to-solution of HOOI compared to STHOSVD for synthetic and scientific simulation datasets.

A.2 Computational Artifacts

$\mathbf{A_1}$ The TuckerMPI-HOOI code repository: https://doi.org/10.5281/zenodo.16752648 which provides a tar archive of the implementations of the proposed distributed-memory HOOI algorithms and is used to generate all experimental results to support contributions C_1-C_5.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	$C_1 - C_5$	Tables 1-2 Figure 1-5 Algorithms 3-5

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact corresponds to the TuckerMPI-HOOI library that implements distributed-memory parallel variants of HOOI including efficient computational kernels for TTM, SVD, and a novel error-specified HOOI formulation for Tucker decomposition. This artifact also corresponds to the synthetic experiments and Miranda experiments. This artifact supports all contributions of the paper and is used to generate experimental results.

Expected Results

One should expect to successfully compile the TuckerMPI library. To run the HOOI and STHOSVD algorithms, we use a parameter file which allows the user to specify the settings and otpimizations used. An example of this parameter file can be seen in the Artifact Execution section. `Dimension Tree Memoization` is a flag that when set to true enables C_3's contribution. `SVD Method` is a flag that when set to 2 enables C_4's contribution. `HOOI-Adapt Threshold` is a float that enables C_2's contribution when set to a float greater than 0.

Expected Reproduction Time (in Minutes)

Following the steps described in Artifact Setup of this artifact, it takes no more than 2 minutes to clone and compile the libary. One could speed up the compiation process by specifying the number of processes used in the `make` command. The system we used suggests one uses no more than 16 processes when doing so, at which point it takes no more than 30 seconds to clone and compile the library.

Artifact Setup (incl. Inputs)

Hardware. The artifact described in this section was run on NERSC Perlmutter (CPU partition) and is intended for use on distributed-memory parallel systems. However, the library can be compiled on any system which meets the software requirements detailed next. Performance experiments were run on NERSC Perlmutter, so access will be provided for artifact evaluation.

Software. We extend the TuckerMPI library, which is a distributed-memory parallel library for Tucker decomposition, by implementing the HOOI algorithms described in this work. TuckerMPI requires the following software packages:

(1) MPI implementation
(2) BLAS implementation
(3) LAPACK implementation
(4) C++11 or greater

Datasets / Inputs. The STHOSVD and HOOI driver functions in TuckerMPI facilitate synthetic data generation by providing configurations (e.g. tensor dimensions and ranks) via a parameter file passed to the driver.

Installation and Deployment. TuckerMPI uses the CMake build system. First download the tar archive of the repository from Zenodo: 10.5281/zenodo.16752648 and extract it. To build the library, run the following commands:

```
cd tucker-mpi-hooi
mkdir build
cd build
cmake ../src
make
```

Optionally, specify compilers explicitly by passing the `-DCMAKE_CXX_COMPILER=mpicxx` and `-DCMAKE_C_COMPILER=mpicc` flags to CMake. Our cmake configuration uses the FindBLAS and FindLAPACK modules to automatically link the BLAS and LAPACK libraries.

Artifact Execution

Create a parameter file with options for STHOSVD. Small scale example of fixed-rank STHOSVD:

```
Print options = true
Print timings = true
Noise = 0.0001
SV Threshold = 0.0
Perform STHOSVD = true
# 4D grid with 8 processors
Processor grid dims = 1 2 2 2
# decrease Global dims if limited by DRAM
Global dims = 100 100 100 100
Ranks = 10 10 10 10
```

Call the STHOSVD MPI driver function using `mpirun`, `mpiexec` or `srun`, specifying the number of MPI processes (should be the product of the `Processsor grid dims` setting). We assume that "srun" is used and the current working directory is the top-level directory in the tucker-mpi-hooi repository and "STHOSVD.cfg" is the parameter filename stored in the top-level directory. The command to run the small scale STHOSVD experiment is:

```
srun -n 8 ./build/mpi/drivers/bin/sthosvd \
--parameter-file STHOSVD.cfg
```

Create a parameter file with the following options to run a small scale example of HOOI with fixed-rank:

```
Print options = true
Print timings = true
Dimension Tree Memoization = false
HOOI Adapt core tensor gather type = false
Noise = 0.0001
HOOI-Adapt Threshold = 0.0
HOOI max iters = 2
SVD Method = 0
# 4D grid with 4 processors
Processor grid dims = 1 2 2 1
# decrease Global dims if limited by DRAM
Global dims = 100 100 100 100
# True ranks of the tensor
Construction Ranks = 10 10 10 10
# Initial guess of ranks for the core tensor
Decomposition Ranks = 10 10 10 10
```

Call the HOOI MPI driver function:

```
srun -n 4 ./build/mpi/drivers/bin/hooi \
--parameter-file HOOI.cfg
```

The table below shows options required for each HOOI variant.

HOOI Variant	Dimension Tree Memoization	SVD Method
HOOI	false	0
HOOI-DT	true	0
HOSI	false	2
HOSI-DT	true	2

Toggle the `Dimension Tree Memoization` and `SVD Method` options as needed to test different variants.

Artifact Analysis (incl. Outputs)

Running the STHOSVD and HOOI driver functions will produce an output stream that is directed to STDOUT with high-level information on the progress of the Tucker decomposition (e.g. for HOOI the approximation error after each iteration).

The output stream can be inspected to verify that the correct algorithm (e.g. STHOSVD or HOOI variant) is being run. Each driver will begin by printing the parameter file options parsed (or default options that are hardcoded in the driver source file). For HOOI variants, the output stream will clearly identify which SVD method is used and whether the dimension tree memoization is enabled.

B.2 Scaling Experiments on Synthetic Data

Expected Results

This experiment will show that the parallel HOOI variants implemented are faster and scale better than STHOSVD when the input tensor is sufficiently low-rank. These experiments will generate and perform Tucker decomposition of a synthetic 4-way tensor with dimensions 560×560×560×560 whose ranks are 10×10×10×10. Since the ranks are known a priori, the decomposition is performed with fixed-rank STHOSVD and HOOI variants. The results will show that HOSI-DT is the fastest HOOI variant, attaining a speedup of 1.5× over STHOSVD and 2.9× over HOOI-DT. These experiments will also show that HOSI-DT scales to 4096 cores on Perlmutter. Qualitatively, the resulting scaling and running time breakdown plots should look similar to Figures 2 and 3 in the paper.

Expected Reproduction Time (in Minutes)

Generating the SLURM submission scripts and parsing the outputs takes about 1 minute. Total node time is $O(10)$ hours to generate CSV files for each algorithm, processor count, and processor grid configuration. This estimate does not account for queuing time on NERSC Perlmutter, which may vary depending on the system load. This experiment does not require online monitoring once the SLURM scripts have been submitted to the queuing system. Once all experiments are completed, the post-processing Python scripts `CollectRank` can be run to parse the outputs and generate the plots. This takes about 1 minute.

Setup

Hardware. Our experiments were conducted on NERSC Perlmutter (CPU partition). The system consists of 3072 compute nodes with dual-socket AMD EPYC 7763 64-core CPUs. Each socket has 4 Non-Uniform Memory Access (NUMA) regions for a total of 8 NUMA regions per node. Each NUMA region has 64 GB of DRAM memory, therefore each CPU socket has 256 GB of DRAM for, a total of 512 GB per node.

Software. The software requirements are the same as Appendix B.1, but with the additional requirement of Python to run scripts to generate SLURM submission scripts and parse CSV timing files. The parsed CSV files are then used to plot Figures 2 and 3 using PGFPlots/TikZ, which requires a LaTeX environment (e.g. Overleaf). More details are provided in the A_1 Artifact Evaluation.

Datasets / Inputs. The synthetic datasets are generated by the HOOI and STHOSVD drivers using random number generators. No additional input files are required.

Execution

Step 0 Load python 3.11
- (a) `module load python/3.11`

Step 1 Run python scripts to generate experiment directories:
- (a) `cd python`
- (b) `python ScaleScript.py`. The default experiment this generates is a strong scaling experiment on a 4way synthetic tensor $560 \times 560 \times 560 \times 560$ with a core size of $10 \times 10 \times 10 \times 10$ in single precision. Which can be found in the newly created `experiments` folder.

Step 2 Submit slurm jobs
- (a) `cd ../experiments/4way_560_10_Single/OutSlurms`
- (b) `for script in ../SlurmScripts/N*.slurm; do sbatch "$script"; done`. This submit all slurm jobs in the `SlurmScripts` folder, one slurm script per data point (i.e. number of processor). The default settings in `ScaleScript.py` will generate a single-node scaling experiment with 13 data points ($p = 1$ to $p = 4096$ in increments of powers of two).

Step 3 Once jobs finish, prostprocess data using python scripts
- (a) `cd python`
- (b) `python CollectScaleScript.py`

Step 4 Plot using latex plotting scheme
- (a) `cd latex`
- (b) Compile `synthetic_scaling.tex` and `synthetic_breakdown.tex` to reproduce the 4way experiments in Figure 2 (up to $p = 4096$) and Figure 3. See the Miranda experiments subsection in the Artifact Evaluation for details on how to reproduce the plots as NERSC Perlmutter does not provide a LaTeX environment.

Analysis

After the jobs of step 2 have returned, we will be left with several CSV output files from our experiments. These provide extensive information for our HOOI variants and STHOSVD algorithms. The CSV files can be automatically parsed and preprocessed by using the `CollectScaleScript.py` script and the LaTeX files to reproduce Figures 2 and 3.

B.3 Experiments on Miranda Simulation Data

Relation To Contributions

Experiments in this section will use the Miranda scientific dataset which supports contribution C_5 and paper elements Figure 4 and 5.

Expected Results

The results of these experiments will show that HOSI-DT is faster than STHOSVD at compressing the Miranda dataset while achieving similar approximation error. Experiments will also highlight that the novel core analysis which allows HOSI-DT to adapt ranks from one iteration to the next is cheap and effective.

Expected Reproduction Time (in Minutes)

Downloading the Miranda dataset (107 GB compressed) takes about 40 minutes to download with an average download speed of 50 MB/s on NERSC Perlmutter. Decompressing the gzipped tar file takes about 15 minutes. Postprocessing the Miranda dataset takes about 30 minutes. Once the dataset is downloaded and decompressed, the time to run experiments is $O(10)$ hours. Experiments do not require online monitoring after the SLURM scripts have been submitted to the queuing system. Once all experiments are completed, then the post-processing Python scripts can be run to parse the outputs and generate the plots. This takes about 5 minutes.

Setup

Hardware/Software. These experiments use the same hardware and software as Appendix B.2.

Datasets / Inputs. The Miranda dataset is part of the Scientific Data Reduction Benchmarks repository (https://sdrbench.github.io/). It can be accessed at https://g-8d6b0.fd635.8443.data.globus.org/ds131.2/Data-Reduction-Repo/raw-data/Miranda/SDRBENCH-Miranda-3072x3072x3072.tar.gz.

Execution

Step 0 Load python 3.11
- (a) `module load python/3.11`

Step 1 Run scripts to download Miranda dataset and generate experiment directories:
- (a) `cd python`
- (b) `./download-setup-miranda.sh`. This will download the Miranda dataset, decompress it, and preprocess the data into a TuckerMPI readable format. The script assumes that a SCRATCH environment variable is defined on the system. This variable is defined on NERSC Perlmutter. We recommend running this script in a screen or tmux session as it will take about 1 hour to download and decompress the dataset.
- (c) `python RankScript.py`. Generates all TuckerMPI input files and SLURM scripts for the experiments.

Step 2 Submit slurm jobs
- (a) `cd ../experiments/Miranda/OutSlurms`
- (b) `for script in $(ls ../SlurmScripts/*.slurm | grep -v 'Default'); do sbatch "$script"; done`. This submit all slurm jobs. The experiments will compare HOSI-DT and STHOSVD algorithms and generate outputs that will reproduce Figures 4 and 5.

Step 3 Once the jobs finish, postprocess data using python scripts
- (a) `cd python`
- (b) `python CollectRankScript.py`

Step 4 Plot using LaTeX/TikZ
- (a) `cd latex`
- (b) Compile `miranda_error.tex` to reproduce Figure 4 and `miranda_breakdown.tex` to reproduce Figure 5. See the synthetic experiments subsection in the Artifact Evaluation for additional details on how to reproduce the plots as NERSC Perlmutter does not provide a LaTeX distribution.

Analysis

After the jobs of step 2 have returned, we will be left with several CSV output files from our experiments. These provide extensive information on using the HOSI-DT and STHOSVD algorithms to compress the Miranda dataset using three error thresholds. The CSV files can be automatically parsed and preprocessed by using the `CollectRankScript.py` script and the LaTeX files to reproduce Figures 2 and 3.

Artifact Evaluation (AE)

Prerequisites. Evaluation of Appendices B.2 and B.3 in the Artifact Description (AD) requires access to the NERSC Perlmutter system in order to reproduce experimental results corresponding to paper elements Figures 2-5.

The AD/AE committee should reach out to the authors to obtain access to the NERSC Perlmutter system before proceeding with the evaluation of the synthetic and Miranda experiments. Source code can be build and tested on a different machine/system provided that software prerequisites of the TuckerMPI-HOOI library are met.

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Access NERSC Perlmutter via SSH using a preferred command line interface.

```
ssh <username>@perlmutter.nersc.gov
module load PrgEnv-gnu/8.5.0
cd $SCRATCH
```

Perform steps described in *Installation and Deployment* of A_1 artifact description. This will clone and compile the TuckerMPI-HOOI library in the scratch directory on Perlmutter.

Artifact Execution

Running the small scale STHOSVD and HOOI experiments requires launching an interactive compute job on Perlmutter. To do so, run the following command:

```
salloc -N 1 -n 8 --qos interactive --mem 256G \
-t 00:30:00 --constraint cpu \
--account <NERSC-allocation-number>
```

This command will allocate an interactive job using one node with 8 cores for 30 minutes. Once the job is allocated, perform steps described in *Artifact Execution* of A_1 in the AD to verify that the TuckerMPI-HOOI library is functional.

C.2 Scaling Experiments on Synthetic Data

Setup

Evaluation of Appendix B.2 in the AD describes steps to reproduce single-node scaling experiments (scaling up to $p = 4096$) shown in Figure 2 (and single-core bar plot in Figure 3) for the 4-way synthetic input tensor. We describe necessary modifications to the `ScaleScript.py` and `CollectScaleScript.py` files to generate a truncated set of scaling data points below:

- Overwrite the list `proc_scale` (Line 45 of `ScaleScript.py`) with a list of desired number of processors (e.g. `proc_scale = [1, 256, 1024, 4096]`). Note that $p = 1$ and $p = 4096$ *must be included* in the list to reproduce Figure 3.
- Overwrite the list `proc_scale` (Line 129 of `CollectScaleScript.py`) with the same list of processors as in `proc_scale` from `ScaleScript.py`.
- Modify the `xtick` and `xticklabels` variables (Lines 31–32 of `synthetic_scaling.tex`) file to match the list of processors in `proc_scale`.

Execution

Follow the steps described in *Artifact Execution* of Appendix B.2 in the AD to run the experiments and generate the output CSV files.

Analysis

The output CSV files will be generated in the `experiments/4way_560_10_Single` subfolder of the TuckerMPI-HOOI repository. We recommend proceeding to the Miranda experiments in the AE before attempting to reproduce Figures 2 and 3, as a LaTeX installation is required. We recommend creating a zipped tar archive of the experiments folder and transferring it to a local machine with a LaTeX distribution.

C.3 Experiments on Miranda Simulation Data

Setup

The Artifact Evaluation for the Miranda experiments in the AD describes steps to reproduce experiments associated with Figures 4 and 5. Performing these experiments requires downloading the Miranda dataset and preprocessing it. We provide a bash script to do so in the python subdfolder of the TuckerMPI-HOOI repository. This bash script assumes that a SCRATCH environment variable is defined on the system. This variable is already defined on NERSC Perlmutter. However, modifications to the bash script may be required if running on a different system. Once the dataset is downloaded and preprocessed, a `Miranda_by_slices` folder will be created in the SCRATCH directory. The `RankScript.py` script will assume the above path to the Miranda dataset.

Execution

Follow the steps described in Appendix B.3 of the AD to run the experiments and generate the output CSV files.

Analysis

The output CSV files will be generated in the `experiments/Miranda/` subfolder of the TuckerMPI-HOOI repository. Once CSV files for the synthetic and Miranda experiments are generated, we recommend creating a zipped tar archive of the experiments folder:

- `tar -czvf hooi-experiments.tar.gz ./experiments`

This archive file can be transferred to a local machine with a LaTeX distribution using a file transfer protocol (e.g. SCP, SFTP, Globus) or the File Browser in the NERSC IRIS Utilities. The latter is only possible for small file sizes.

We recommend downloading the TuckerMPI-HOOI (10.5281/zenodo.16752648) on a local machine with a LaTeX distribution and transferring the `hooi-experiments.tar.gz` archive into the repository's top-level directory. Once the archive has been transferred, it can be extracted using the following command:

- `tar -xzvf hooi-experiments.tar.gz`

This will create the `experiments` subfolder in the TuckerMPI-HOOI repository with the required CSV files to reproduce Figures 2-5. Once the CSV files are available, the LaTeX files in the `latex` subfolder can be compiled to reproduce Figures 2-5, using your typical LaTeX compilation commands (`latex`, `pdflatex`, `latexmk`, etc.).

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact’s reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	●	●	○
Badge awarded	yes	yes	no

E Reproduction of Computational Artifacts

E.1 Timeline

The experiments were conducted between September 10, 2025, to September 18, 2025.

E.2 Computational Environment and Resources

In order to evaluate the results in the paper, the time duration as stated exceeded the time allotted for reproducibility.

E.3 Details on Artifact Reproduction

- The TuckerMPI-HOOI code repository was accessible at the Zenodo link. Artifact Setup was exactly as described the the ADAE Report.
- The build and execution of the artifact was attempted initially on a HPE EX40 system at Sandia National Laboratories. The execution of the experiment on the Intel Sapphire Rapids architecture was unsuccessful, due to the differences in node architecture from that which is stated in the ADAE report.
- Upon acquisition of an account on Perlmutter at National Energy Research Scientific Computing Center (NERSC), the named system in the ADAE report, functional evaluation of the artifact was achievable, however, there were several batch submissions that experienced out-of-memory errors, resulting in data gaps.
- The general instructions for software and data acquisition works as described.
- The automation that generates scheduler scripts worked as described to produce usable scripts on the target system.
- The runtime and build time environment was exactly as described, and the artifacts that conduct the scaling, and comparative analyses of the results execute without error.
- Therefore, the reviewer achieved functional evaluation of this experiment’s primary claim, and is awarded the Results Evaluated Badge.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.