Parallel Tucker Decomposition with Numerically Accurate SVD

Zitong Li
Wake Forest University
Winston-Salem, NC, USA
liz20@wfu.edu

Qiming Fang
Wake Forest University
Winston-Salem, NC, USA
fangq18@wfu.edu

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

ABSTRACT

Tucker decomposition is a low-rank tensor approximation that generalizes a truncated matrix singular value decomposition (SVD). Existing parallel software has shown that Tucker decomposition is particularly effective at compressing terabyte-sized multidimensional scientific simulation datasets, computing reduced representations that satisfy a specified approximation error. The general approach is to get a low-rank approximation of the input data by performing a sequence of matrix SVDs of tensor unfoldings, which tend to be short-fat matrices. In the existing approach, the SVD is performed by computing the eigendecomposition of the Gram matrix of the unfolding. This method sacrifices some numerical stability in exchange for lower computation costs and easier parallelization. We propose using a more numerically stable though more computationally expensive way to compute the SVD by preprocessing with a QR decomposition step and computing an SVD of only the small triangular factor. The more numerically stable approach allows us to achieve the same accuracy with half the working precision (for example, single rather than double precision). We demonstrate that our method scales as well as the existing approach, and the use of lower precision leads to an overall reduction in running time of up to a factor of 2 when using 10s to 1000s of processors. Using the same working precision, we are also able to compute Tucker decompositions with much smaller approximation error.

ACM Reference Format:
Zitong Li, Qiming Fang, and Grey Ballard. 2021. Parallel Tucker Decomposition with Numerically Accurate SVD. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472472

1 INTRODUCTION

Scientific data is being generated from observations, experiments, and simulations at an unprecedented pace. The sheer amount of data is overwhelming conventional methods of storage, transfer, and analysis, which drives a need for efficient and accurate data reduction techniques. Many data sets represent multi-way relationships, such as interactions occurring over time, and are naturally represented as a multidimensional array, or tensor. For such datasets, tensor approximations such as the Tucker decomposition provide low-rank representations consisting of far fewer parameters than the original data. Furthermore, a Tucker decomposition can be computed to satisfy a desired approximation error to ensure the representation maintains sufficient fidelity. Prior work has demonstrated the efficacy of Tucker in compressing datasets arising from a wide range of applications [1, 4, 6, 7, 9, 10, 21, 22, 25].

In order to compress datasets that are too large to store or process on a single machine, we need efficient and scalable parallel algorithms for computing Tucker decompositions that do not sacrifice approximation error. We propose in this work a new parallel algorithm for computing Tucker decompositions of dense tensors that is more numerically stable than previous approaches. This algorithm allows us to compute better approximations of large data than previously available, and it also enables us to reduce the working precision to cut down on computation and communication costs, achieving the same approximation error in less time.

Our implementation builds on an existing C++/MPI library called TuckerMPI [6]. TuckerMPI implements the Sequentially Truncated Higher-Order SVD (ST-HOSVD) algorithm [28], which requires a series of matrix singular value decompositions (SVDs) of short-fat matrices. The existing approach uses the Gram-SVD algorithm, which we describe in Sec. 2. This approach sacrifices some numerical stability to reduce computational cost. In contrast, our algorithm is numerically stable, and the parallelization strategy is based on techniques used in the Tall-Skinny QR algorithm [13], a communication-efficient approach to computing a QR decomposition of matrix with many more rows than columns.

Sec. 3 describes our core algorithms, including the matrix SVD algorithm called QR-SVD and the application of QR-SVD to the tensor case in both sequential and parallel scenarios. We also provide complexity analysis to demonstrate the computational and communication overheads of the numerically stable approach compared to TuckerMPI. In addition, our implementation generalizes the TuckerMPI library to use either single or double precision. This enables us to use a combination of precision (single/double) and algorithm (Gram-SVD/QR-SVD) that outperforms the original TuckerMPI, which uses Gram-SVD in double precision, in almost every scenario. The details of the performance of our algorithm are found in Sec. 4.

Our main contributions include

- the development of a numerically stable parallel algorithm for computing Tucker decompositions,
- the generalization (via C++ templates) of the TuckerMPI library to enable single-precision computation,
- a demonstration of improved running times (of up to 2× for synthetic data and 60% for application data) for large approximation error thresholds,
- and the capability of accurately computing decompositions with very small approximation error thresholds (below 10^{-8}).

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472472

2 PRELIMINARIES

2.1 Notation

Following [17], we use bold script letters such as $\mathcal{X}$ to denote tensors, bold capital letters such as $\mathbf{U}$ to denote matrices, and lower case letters to denote scalars. We let N be the number of modes of the data tensor and use 0-indexing, so that $I_0 \times I_1 \times \cdots \times I_{N-1}$ specifies the dimensions of a given tensor. In addition, we define the following notation for products of all dimensions and dimensions before and after a mode n:

$$I_n^{\circledast} = \prod_{k=0}^{N-1} I_k, \qquad I_n^{\circledleft} = \prod_{k=0}^{n-1} I_k, \qquad I_n^{\circledright} = \prod_{k=n+1}^{N-1} I_k.$$

We use Matlab-style indexing to specify sub-tensors, such as $\mathbf{U}(:, j)$ for the jth column of $\mathbf{U}$. A tensor fiber is a vector that is a sub-tensor with all but one index fixed, such as $\mathcal{X}(i, :, k)$. We consider N unfoldings, or matricizations, of a tensor in this work, in which a single mode is mapped to the rows of a matrix and all other modes are mapped to the columns. In particular, the nth unfolding, denoted by $\mathbf{X}_{(n)}$, is an $I_n \times I_n^{\circledleft} I_n^{\circledright}$ matrix whose columns corresponding to the n-mode fibers of the tensor. The main tensor contraction relevant to this work is the tensor-times-matrix (TTM) operation, which is denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$ and defined so that $\mathbf{Y}_{(n)} = \mathbf{U}\mathbf{X}_{(n)}$. A tensor norm is denoted $\|\mathcal{X}\|$ and is defined so that $\|\mathcal{X}\|^2 = \sum_{i_0,\ldots,i_{N-1}} \mathcal{X}(i_0, \ldots, i_{N-1})^2$.

We perform complexity analysis using the Message Passing Interface (MPI) model. In this model, processors communicate via messages that have latency and bandwidth costs. We assume a message of w words costs $\alpha + \beta \cdot w$, where α is the per-message latency cost and β is the per-word bandwidth cost. In analyzing the computation costs, we use γ to denote the per-floating-point-operation cost. We note that the γ and β costs typically vary depending on the working precision, which determines the number of bits per word.

2.2 Tucker and ST-HOSVD

The Tucker decomposition [17, 19, 27], or approximation, of a tensor $\mathcal{X}$ can be viewed as a higher-dimensional generalization of the matrix singular value decomposition. It consists of a core tensor $\mathcal{G}$ and a set of factor matrices $\{\mathbf{U}_n\}$ so that

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

Here $\mathcal{G}$ has as many modes as $\mathcal{X}$ but when used as a low-rank approximation we expect its dimensions $R_0 \times \cdots \times R_{N-1}$, which correspond to the multilinear rank of $\hat{\mathcal{X}}$, to be much smaller that those of $\mathcal{X}$. Because the factor matrices have only two dimensions, the compression ratio provided by a Tucker approximation is close to $I_0 \cdots I_{N-1}/(R_0 \cdots R_{N-1})$. We measure the relative approximation error in a normwise sense: $\|\mathcal{X} - \hat{\mathcal{X}}\|/\|\mathcal{X}\|$.

The core algorithm we employ for computing Tucker decompositions is the Sequentially Truncated Higher-Order Singular Value Decomposition (ST-HOSVD) [28], given in Alg. 1. This algorithm performs a truncated matrix SVD in order to determine a factor matrix for each mode. An important feature of the algorithm is that each computed factor matrix is used to truncate the tensor in that mode before performing the SVD for the next mode in the sequence. This means that both the data and computation tend to decrease as the algorithm progresses through the modes. Another feature of the algorithm is that the algorithm is guaranteed (in exact arithmetic) to produce an approximation that satisfies a prescribed error tolerance, as the truncation ranks can be determined from the matrix SVDs. While the algorithm does not necessarily compute the optimal Tucker approximation with the given ranks, it is quasi-optimal, with an approximation error within a factor of $\sqrt{N}$ of the optimal error. Alg. 1 progresses through the modes in increasing order. However, it can use any mode ordering, which can be tuned to minimize computation or other metrics if the reduced ranks are known a priori.

Algorithm 1 ST-HOSVD [28]

1: **function** ST-HOSVD($\mathcal{X}, \epsilon$)
2: $\quad\mathcal{Y} = \mathcal{X}$
3: $\quad$**for** $n = 0$ to $N-1$ **do**
4: $\quad\quad[\mathbf{U}, \Sigma, \sim] = \text{SVD}(\mathbf{Y}_{(n)})$ ▷ right singular vectors not computed
5: $\quad\quad R_n = \min\left\{R \mid \sum_{i=R+1}^{I_n} \sigma_i^2 \le \epsilon^2 \|\mathcal{X}\|^2/N\right\}$ ▷ determine rank
6: $\quad\quad\mathbf{U}_n = \mathbf{U}(:, 1:R_n)$ ▷ nth factor matrix
7: $\quad\quad\mathcal{Y} = \mathcal{Y} \times_n \mathbf{U}_n^T$ ▷ TTM truncation
8: $\quad$**end for**
9: $\quad\mathcal{G} = \mathcal{Y}$ ▷ core tensor
10: $\quad$**return** $(\mathcal{G}, \mathbf{U}_0, \ldots, \mathbf{U}_{N-1})$
11: **end function**

2.3 TuckerMPI and Gram-SVD

TuckerMPI [6] is an open-source C++/MPI software library that implements the ST-HOSVD algorithm in a distributed-memory parallel environment. Our implementation builds upon the software, and we compare our proposed approach with that of TuckerMPI in this paper. TuckerMPI implements Alg. 1 by parallelizing the SVD in line 4 and the TTM truncation in line 7.

The algorithm it uses for computing the SVD is known as Gram-SVD (see [26, Lecture 31], for example). Given an $m \times n$ matrix $\mathbf{A}$, the Gram-SVD algorithm exploits the observation that the SVD of $\mathbf{A}$ is related to the (symmetric) eigenvalue decomposition of its Gram matrix $\mathbf{A}\mathbf{A}^\mathsf{T}$. If $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\mathsf{T}$ is the SVD of $\mathbf{A}$, then $\mathbf{A}\mathbf{A}^\mathsf{T} = \mathbf{U}\Sigma^2\mathbf{U}^\mathsf{T}$ is the eigendecomposition of the Gram matrix, which has dimension $m \times m$. Note that only the left singular vectors and singular values are required in line 4 of Alg. 1. Thus, the computational costs of Gram-SVD are $nm^2 + O(m^3)$, and if $m \ll n$, then nearly all the computation is performed in the (symmetric) matrix multiplication that forms the Gram matrix. While this is an efficient way to compute all singular values and left singular vectors, it has numerical consequences as we will discuss further in Sec. 3.2.

2.4 Tall-Skinny QR

As we describe in Sec. 3, the main computational bottleneck in our approach to implementing ST-HOSVD is similar to computing the QR decomposition of a tall-skinny matrix. This is a well-studied matrix computations problem, and the approach we use is based on the Tall-Skinny QR (TSQR) algorithm [13], which we briefly summarize here.

The standard algorithm for QR decomposition is based on Householder transformations: for each column of the matrix, a Householder vector is computed and its transformation is applied to

the trailing matrix. Because the Householder transformation application consists of matrix-vector operations, it suffers from low computational intensity and typically poor performance. Consider an $m \times n$ matrix with $m \gg n$. In the sequential case, if m is so large that not even a single column fits into cache, the Householder-based algorithm must stream the trailing matrix from memory n times. In the parallel case, the algorithm requires processors to synchronize n times.

The benefit of the TSQR algorithm is that it streams through the matrix only once in the sequential case and requires only one global synchronization across processors in the parallel case. While the computational costs are nearly the same between the two algorithms, the reduction in communication costs leads to overall performance improvements, as has been demonstrated on various platforms [3, 11, 23]. The idea of the algorithm is to reorder the annihilation of matrix elements below the upper triangle. While the typical Householder algorithm annihilates entries column by column, the sequential TSQR algorithm annihilates row block by row block (where each row block fits entirely in cache), and the parallel algorithm allows processors to annihilate as many local entries as possible before communicating with other processors. The algorithm must be careful not to fill in any previously annihilated entries and respect dependences among annihilations. The order of annihilation can be organized into a reduction tree, where the sequential approach is a flat tree and the parallel approach is a binomial tree. We use both techniques in this work. The local computational kernels for implementing TSQR-based algorithms are available as subroutines in current versions of LAPACK.

3 ALGORITHMS

We present our proposed algorithms in this section, starting with an explanation of the QR-SVD algorithm in Sec. 3.1 and a numerical comparison with Gram-SVD in Sec. 3.2. Then we explain how to incorporate QR-SVD in the sequential ST-HOSVD algorithm in Sec. 3.3 and in the parallel algorithm in Sec. 3.4. Finally, we compare the complexity analysis of ST-HOSVD via QR-SVD with TuckerMPI's approach in Sec. 3.5.

3.1 QR-SVD

While TuckerMPI employs Gram-SVD, we propose to use the QR-SVD algorithm. This method is also known as R-bidiagonalization [15, Section 5.4.9]. The idea is that pre-multiplying a matrix by an orthogonal matrix does not change the singular values or the right singular vectors. Thus, computing a QR decomposition of a tall-skinny matrix reduces the SVD problem to a small, upper triangular matrix.

Likewise, in our case, an LQ decomposition of short-fat matrix reduces the problem to a small lower triangular matrix. That is, for $m \times n$ matrix $\mathbf{A}$ with LQ decomposition $\mathbf{A} = \mathbf{LQ}$, the SVD of $\mathbf{L} = \mathbf{U}\Sigma\mathbf{V}_L^\mathsf{T}$ yields the SVD of $\mathbf{A} = \mathbf{U}\Sigma(\mathbf{Q}^\mathsf{T}\mathbf{V}_L)^\mathsf{T}$. When only the singular values and left singular vectors are desired, neither $\mathbf{Q}$ nor $\mathbf{V}_L$ need be computed. The computational cost of QR-SVD in this case is $2mn^2 + O(m^3)$, and when $m \ll n$ the cost is dominated by the LQ decomposition. Note that this is approximately twice the cost of Gram-SVD.

3.2 Numerical Accuracy of SVD Algorithms

In this section we detail the differences in error bounds between QR-SVD and Gram-SVD. We use ε to denote machine precision. Considering an $m \times n$ matrix $\mathbf{A}$, let $\sigma_i = \sigma_i(\mathbf{A})$ denote the ith singular value of $\mathbf{A}$, and let $\tilde{\sigma}_i$ be its computed approximation. In the context of ST-HOSVD, we focus on the accuracy of the left singular vector matrix $\mathbf{U}$ of $\mathbf{A}$ (also the eigenvector matrix of $\mathbf{AA}^\mathsf{T}$) consisting of vectors $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$ with computed approximations $\{\tilde{\mathbf{u}}_1, \ldots, \tilde{\mathbf{u}}_m\}$. We define the $m \times k$ matrix $\mathbf{U}_k$ to have columns $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$, which is approximated by the computed $\tilde{\mathbf{U}}_k$ defined similarly. The angular difference between subspaces $\mathcal{U}$ and $\mathcal{V}$, denoted $\theta(\mathcal{U}, \mathcal{V})$, is the maximum principal (canonical) angle between the subspaces [15, Section 6.4.3]. In this section we use $\|\cdot\|$ notation to denote either the Frobenius or spectral norm of a matrix.

The following result applies to the QR-SVD approach because the QR preprocessing step is backward stable.

Theorem 1 ([12, 15, 26]). *Using a backward stable algorithm for SVD, the absolute error of each computed singular value satisfies*

$$|\tilde{\sigma}_i - \sigma_i| = O(\varepsilon\|\mathbf{A}\|). \tag{1}$$

The angle between each computed and exact left singular vector satisfies

$$\cos^{-1}(\mathbf{u}_i^\mathsf{T}\tilde{\mathbf{u}}_i) = \theta_i \approx \frac{1}{2}\sin 2\theta_i = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\min\{\sigma_{i-1}-\sigma_i, \sigma_i-\sigma_{i+1}\}}\right), \tag{2}$$

where $\theta \approx \frac{1}{2}\sin 2\theta$ when $\theta \ll 1$. Furthermore, the angular difference of the subspace spanned by the leading k computed singular vectors and the true space satisfies

$$\theta\left(range(\mathbf{U}_k), range(\tilde{\mathbf{U}}_k)\right) = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\sigma_k - \sigma_{k+1}}\right). \tag{3}$$

The low-rank approximation relative error satisfies

$$\frac{\left\|(\mathbf{I} - \tilde{\mathbf{U}}_k\tilde{\mathbf{U}}_k^\mathsf{T})\mathbf{A}\right\|}{\|\mathbf{A}\|} = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\sigma_k - \sigma_{k+1}}\right) + \frac{\left\|(\mathbf{I} - \mathbf{U}_k\mathbf{U}_k^\mathsf{T})\mathbf{A}\right\|}{\|\mathbf{A}\|}. \tag{4}$$

For Gram-SVD, the error bounds for the ith singular value and left singular vector are larger by a factor of $\|\mathbf{A}\|/\sigma_i$, and the error bounds for the leading k computed singular vectors are larger by a factor of $\|\mathbf{A}\|/\sigma_k$.

Theorem 2. *Using the Gram approach, computing the SVD of $\mathbf{A}$ via the eigendecomposition of the Gram matrix $\mathbf{AA}^T$, the absolute error of each computed singular value satisfies*

$$|\tilde{\sigma}_i - \sigma_i| = O\left(\varepsilon\|\mathbf{A}\| \cdot \frac{\|\mathbf{A}\|}{\sigma_i}\right) \tag{5}$$

and the angle between each computed and exact left singular vector satisfies

$$\cos^{-1}(\mathbf{u}_i^\mathsf{T}\tilde{\mathbf{u}}_i) = \theta_i \approx \frac{1}{2}\sin 2\theta_i = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\min\{\sigma_{i-1}-\sigma_i, \sigma_i-\sigma_{i+1}\}} \cdot \frac{\|\mathbf{A}\|}{\sigma_i}\right), \tag{6}$$

where $\theta \approx \frac{1}{2}\sin 2\theta$ when $\theta \ll 1$. Furthermore, the angular difference of the subspace spanned by the leading k computed singular vectors and the true space satisfies

$$\theta\left(range(\mathbf{U}_k), range(\tilde{\mathbf{U}}_k)\right) = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\sigma_k - \sigma_{k+1}} \cdot \frac{\|\mathbf{A}\|}{\sigma_k}\right). \tag{7}$$

The low-rank approximation relative error satisfies

$$\frac{\left\|(\mathbf{I}-\tilde{\mathbf{U}}_k\tilde{\mathbf{U}}_k^{\mathsf{T}})\mathbf{A}\right\|}{\|\mathbf{A}\|} = O\left(\frac{\varepsilon\|\mathbf{A}\|}{\sigma_k-\sigma_{k+1}}\cdot\frac{\|\mathbf{A}\|}{\sigma_k}\right)+\frac{\left\|(\mathbf{I}-\mathbf{U}_k\mathbf{U}_k^{\mathsf{T}})\mathbf{A}\right\|}{\|\mathbf{A}\|}. \quad (8)$$

These bounds can be established by applying standard results (see [12, 15, 26] for example) for the symmetric eigendecomposition of the Gram matrix. Equations (4) and (8) follow from eqs. (3) and (7), respectively, via [14, Theorem 6].

From eq. (1), we see that for QR-SVD the relative error in the singular values $|\tilde{\sigma}_i-\sigma_i|/\sigma_i = O(1)$ when $\|\mathbf{A}\|/\sigma_i \approx 1/\varepsilon$. That is, QR-SVD can compute singular values to the right order of magnitude until the ratio of the largest singular value to the ith singular value is $O(1/\varepsilon)$. For Gram-SVD, eq. (5) implies that the relative singular value error becomes $O(1)$ when $\|\mathbf{A}\|/\sigma_i \approx 1/\sqrt{\varepsilon}$, so Gram-SVD can compute singular values to the right order of magnitude only in a range of $O(\sqrt{\varepsilon})$.

Thus, computed singular values that are smaller than $O(\|\mathbf{A}\|\varepsilon)$ (for QR-SVD) and $O(\|\mathbf{A}\|\sqrt{\varepsilon})$ (for Gram-SVD) should be considered noise due to roundoff error and cannot be trusted. This is particularly important for the rank determination procedure in ST-HOSVD (line 5 in Alg. 1), and it means that ST-HOSVD cannot compute a Tucker decomposition with approximation error smaller than $O(\varepsilon)$ if QR-SVD is used or $O(\sqrt{\varepsilon})$ if Gram-SVD is used. From eqs. (4) and (8), we see that the relative approximation error bounds depend on two summands, the second of which is the error of the exact truncated SVD. Again, the first term of the error for Gram-SVD is amplified by a factor of $\|\mathbf{A}\|/\sigma_k$ compared to that of QR-SVD. These bounds show the maximum attainable approximation accuracy of each method. For example, if $\sigma_{k+1} = 0$, the first term becomes $O(1)$ when $\|\mathbf{A}\|/\sigma_k \approx 1/\varepsilon$ (for QR-SVD) or $\|\mathbf{A}\|/\sigma_k \approx 1/\sqrt{\varepsilon}$ (for Gram-SVD).

To illustrate these properties, we present in Fig. 1 the results of an experiment using QR-SVD and Gram-SVD algorithms to compute the singular values of a synthetic matrix. We create the 80×80 matrix to have geometrically decaying singular values from 10^0 to 10^{-18} and random singular vectors. We use single ($\varepsilon_s = 2^{-23} \approx 10^{-7}$) and double ($\varepsilon_d = 2^{-52} \approx 10^{-16}$) precision for each algorithm. In our experiment, the Gram-SVD algorithm computes some negative eigenvalues of the Gram matrix corresponding to singular values that have lost all relative accuracy. To compute the singular values, we take the square root of the absolute value of the eigenvalues and then sort them in decreasing order.

Observe that the largest singular values are computed accurately (at least to the correct order of magnitude) in all cases until approximately $\sqrt{\varepsilon_s} \approx 10^{-4}$, at which point the Gram-SVD algorithm in single precision loses accuracy. The next algorithm to lose accuracy is QR-SVD in single precision, as it hits $\varepsilon_s \approx 10^{-7}$. Shortly afterward, at $\sqrt{\varepsilon_d} \approx 10^{-8}$, Gram-SVD in double precision loses accuracy. The QR-SVD algorithm in double precision can compute singular values to the correct order of magnitude until they become smaller than $\varepsilon_d \approx 10^{-16}$.

3.3 Sequential ST-HOSVD via QR-SVD

To perform ST-HOSVD on a tensor that exists in the memory of a single processor, we design the computational kernels to operate on the memory layout of the tensor unfoldings. In this way, we avoid costly reorderings of tensor data in memory. These kernels are also used as subroutines in our parallel implementation.

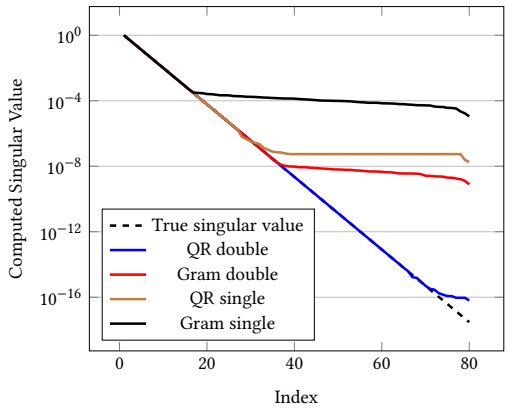


Figure 1: Computed singular values of SVD algorithms on matrix with geometrically decaying singular values

Data Layout. The memory layout of a tensor unfolding can be seen as a series of contiguous row-major column blocks [4, 6, 16, 20, 24]. For the nth unfolding, the number of rows is I_n, the number of columns of each block is $I_n^{\textcircled{<}}$, and the number of column blocks is $I_n^{\textcircled{>}}$. We use the notation $\mathbf{Y}_{(n)}[j]$ to denote the jth column block of the nth tensor unfolding, which is a contiguous row-major matrix. There are special cases for $n = 0$, where $I_n^{\textcircled{<}} = 1$ so that the unfolding is a single column-major matrix, and $n = N-1$, where $I_n^{\textcircled{>}} = 1$ so that the unfolding is a single row-major matrix.

ST-HOSVD. In our implementation of ST-HOSVD using QR-SVD, we build upon the previous work of TuckerMPI [6]. As we see in Alg. 1, the key change is in how the SVD of line 4 is computed. The algorithm for computing the Gram matrix in the Gram-SVD approach is performed using successive calls to the `syrk` BLAS routine [6, Alg. 2]. Our algorithm for computing the LQ factorization in the QR-SVD approach is given by Alg. 2 and is described in detail below. Instead of performing a symmetric eigendecomposition of the Gram matrix, we compute a singular value decomposition of the triangular factor $\mathbf{L}$, discarding the right singular vectors. We use the existing TTM kernel from TuckerMPI [6, Alg. 3] in our implementation of line 7 of Alg. 1.

Algorithm 2 Sequential LQ of Tensor Unfolding

1: **function** $\mathbf{L}$ =TensorLQ($\mathbf{Y}$, n)
2: **if** $n = 0$ **then**
3: $\mathbf{L} = \mathrm{LQ}(\mathbf{Y}_{(n)})$ ▷ Call to `gelq`, $\mathbf{Y}_{(0)}$ is col-major
4: **else if** $n < N-1$ **then**
5: $\mathbf{L} = \mathrm{LQ}(\mathbf{Y}_{(n)}[0])$ ▷ Call to `geqr`
6: **for** $j = 1$ to $I_n^{\textcircled{>}} - 1$ **do**
7: $\mathbf{L} = \mathrm{LQ}(\begin{bmatrix}\mathbf{L} & \mathbf{Y}_{(n)}[j]\end{bmatrix})$ ▷ Call to `tpqrt`
8: **end for**
9: **else**
10: $\mathbf{L} = \mathrm{LQ}(\mathbf{Y}_{(n)})$ ▷ Call to `geqr`, $\mathbf{Y}_{(N-1)}$ is row-major
11: **end if**
12: **end function**

LQ Decomposition. Alg. 2 is the key to the efficiency of our sequential implementation of ST-HOSVD. We consider the first and last modes as special cases. The first unfolding $\mathbf{Y}_{(0)}$ is column major, so we make a direct call to the LQ driver routine `gelq`. The last unfolding $\mathbf{Y}_{(N-1)}$ is row major, so we make a direct call to the QR driver routine `geqr`, which is equivalent to computing an LQ of the transpose and accounts for the internal layout. In the general case, we compute LQ using a TSQR-like algorithm based on a flat tree [13]. After the first block of the unfolding is reduced to a triangle, we successively annihilate each block and update the triangle. At each iteration, we are performing (the transpose of) a QR decomposition of a triangle on top of a rectangle (or more generally a pentagon), which is implemented in the computational subroutine `tpqrt`. Note that the case $N-1$ can be subsumed into the general case; we separate it in the pseudocode to emphasize the single call to `geqr` because the first and last modes are typically computational bottlenecks. We also point out one implementation detail that is not performance critical. If $\mathbf{Y}_{(n)}[0]$ is not short and fat, then the LQ factorization in line 5 will not produce a triangular matrix. In this case, we combine as many blocks as necessary to obtain a submatrix with more columns than rows before performing the first LQ decomposition.

3.4 Parallel ST-HOSVD via QR-SVD

Performing ST-HOSVD in parallel requires performing a sequence of parallel matrix operations. Unfortunately, the best matrix distributions for this sequence of operations cannot be simultaneously achieved. We use a combination of a carefully chosen data distribution with redistribution of data where necessary to achieve scalable performance.

Data Distribution. Following TuckerMPI [6], we logically organize the processors into a processor grid with as many modes as the data tensor and distribute the tensor across processors in a block fashion. In this way, given a tensor with dimensions $I_0 \times \cdots \times I_{N-1}$ and a processor grid $\mathcal{P}$ with dimensions $P_0 \times \cdots \times P_{N-1}$, each processor owns a contiguous subtensor with dimensions $(I_0/P_0) \times \cdots \times (I_{N-1}/P_{N-1})$, assuming even division. In the case of uneven division, each mode can be handled independently. In mode n we assign $\lceil I_n/P_n \rceil$ indices to the first $I_n \bmod P_n$ processors in each fiber and $\lfloor I_n/P_n \rfloor$ indices to the remaining processors. At the end of the algorithm, the output core tensor has the same distribution as the input tensor (though with reduced tensor dimensions), and the output factor matrices are stored redundantly on all processors.

When the tensor is distributed in block fashion, each tensor unfolding inherits a block matrix distribution [6, Section 4.4]. While they are not necessarily a standard block or block-cyclic matrix distribution, the distributions of the unfoldings do correspond to a 2D block distribution after a column permutation is applied to the unfolding. Because we are interested in left singular vectors of each unfolding, this logical column permutation can be ignored, allowing us to assume a standard 2D block distribution.

ST-HOSVD. As in the sequential case, our contribution is the efficient implementation of the QR-SVD approach to line 4 of Alg. 1. TuckerMPI implements the Gram-SVD approach, where performance is determined by the parallel computation of the Gram matrix of the tensor unfolding [6, Alg. 4]. The efficiency of our approach depends on computing an LQ decomposition of each unfolding in parallel, as presented in Alg. 3. We perform a redundant sequential SVD (using `gesvd`) of the resulting triangular factor, and we re-use the existing parallel routines in TuckerMPI for TTM truncation.

LQ Decomposition. In order to compute the LQ decomposition of each unfolding in parallel, we take the general approach of the Tall-Skinny QR (TSQR) algorithm [13], as described in Sec. 2.4. TSQR is based on a 1D distribution of the matrix and consists of a local QR decomposition phase followed by a reduction-tree phase. The two main challenges of employing this approach for ST-HOSVD is (1) the unfolding may not be in 1D distribution, and (2) the local part of the unfolding may not be in column- or row-major layout. The parallel LQ decomposition algorithm is given as Alg. 3. The pseudocode assumes that the number of processors P is a power of two to simplify the presentation of the reduction-tree phase, but our implementation works for any P.

To address the first challenge, we perform a redistribution of the tensor if the unfolding is not in 1D distribution. Note that if $P_n = 1$, then the nth unfolding is in 1D (column) distribution, so no redistribution is necessary. If $P_n > 1$, then we can obtain a 1D distribution by considering individual processor fibers in mode n. The submatrix of the nth unfolding collectively owned by a processor fiber has dimension $I_n \times \frac{I_n^{\oslash} I_n^{\oslash}}{P_n^{\oslash} P_n^{\oslash}}$ and is distributed row-wise across the P_n processors in the fiber. By redistributing this submatrix to column-wise distribution within the fiber (and doing so within each processor fiber), we obtain a 1D column distribution of the entire unfolding. Note that this is the same redistribution used for the most efficient Gram algorithm [6, Alg. 4], and we re-use existing TuckerMPI code in our implementation. Likewise, other redistribution optimizations that have been performed for the Gram-based approach [10] can also be applied to improve our QR-based approach.

To address the second challenge, we use our sequential TensorLQ algorithm to tailor the local QR computation to the data layout for the particular mode. If $P_n = 1$, then no redistribution occurs, and the local unfolding is in the natural tensor layout (a series of row-major submatrices). If $P_n > 1$, and redistribution occurs, then we tailor the computation to the layout given by the TuckerMPI redistribution routines. For the last mode ($n = N-1$), the output is row major, and the local computation can be performed by a single call to `geqr`. For all other modes, the output is column major, and we use a single call to `gelq`. The variation across modes is the result of techniques to avoid packing or unpacking data for the all-to-all collective where possible.

After the local computation is performed, we perform a butterfly variant [2, 5] of the TSQR reduction tree, which behaves like an all-reduce collective on the triangular factors and ends with the result stored redundantly on every processor. At each of $\log P$ steps, each processor exchanges data with a partner processor and eliminates the lower-ranked triangular factor using an LQ decomposition as a reduction operation. Because each block of the concatenated matrix is a triangle, we can again use the `tpqrt` routine for the QR decomposition of a triangle on top of another triangle (or more generally a pentagon).

Algorithm 3 Parallel LQ of Tensor Unfolding

1: **function** $\mathbf{L}$ =ParTensorLQ($\bar{\mathcal{Y}}, n, \mathcal{P}$)
2: $p = \mathcal{P}(p_0, \ldots, p_{N-1})$ ▷ Get linear index for proc $(p_0, \ldots, p_{N-1})$
3: myProcFiber = $\mathcal{P}(p_0, \ldots, p_{n-1}, :, p_{n+1}, \ldots, p_{N-1})$
4: **if** $P_n = 1$ **then**
5: $\mathbf{L}^{(p)}_{\log P}$ = TensorLQ($\bar{\mathbf{Y}}_{(n)}$) ▷ Call sequential algorithm
6: **else**
7: $\mathbf{Z}$ = Redistribute($\bar{\mathbf{Y}}_{(n)}$, myProcFiber) ▷ MPI_Alltoall
8: $\mathbf{L}^{(p)}_{\log P}$ = LQ($\mathbf{Z}$) ▷ Call to gelq (geqr for $n = N-1$)
9: **end if**
10: **for** $i = \log P - 1$ down to 0 **do**
11: $q = 2^{i+1}\lfloor p/2^{i+1} \rfloor + (p + 2^i) \bmod 2^{i+1}$ ▷ Determine partner
12: Exchange $\mathbf{L}^{(p)}_{i+1}$ and $\mathbf{L}^{(q)}_{i+1}$ with proc q ▷ MPI_Sendrecv
13: **if** $p < q$ **then**
14: $\mathbf{L}^{(p)}_{i} = \text{LQ}(\begin{bmatrix}\mathbf{L}^{(p)}_{i+1} & \mathbf{L}^{(q)}_{i+1}\end{bmatrix})$ ▷ Call to tpqrt
15: **else**
16: $\mathbf{L}^{(p)}_{i} = \text{LQ}(\begin{bmatrix}\mathbf{L}^{(q)}_{i+1} & \mathbf{L}^{(p)}_{i+1}\end{bmatrix})$ ▷ Call to tpqrt
17: **end if**
18: **end for**
19: $\mathbf{L} = \mathbf{L}^{(p)}_0$ ▷ Result computed on all processors
20: **end function**

We point out another implementation detail that is typically not performance critical. If, even after redistribution, the local unfolding is not short and fat, then the LQ decomposition does not produce a triangular matrix, which is required by the tree reduction. This can occur if the number of processors is very large or if the product of all modes but n is very small, which is more likely to occur for later modes during ST-HOSVD when there is significant dimension reduction in earlier modes. In this case, the 1D distribution and TSQR-based approach may not be communication optimal. We choose to proceed with the tree reduction by padding the first block with zeros to make it triangular, which adds only a lower order term to the computational costs. The zeros are quickly filled in after a few levels of the tree.

SVD of $\mathbf{L}$. After the LQ decomposition is computed from the butterfly TSQR, every processor has the same triangular $\mathbf{L}$ factor. To complete the SVD of $\mathbf{Y}_{(n)}$, the SVD of this $\mathbf{L}$ matrix is computed on each processor redundantly. The rest of the algorithm is the same as the original TuckerMPI.

3.5 Complexity Analysis

We now analyze the computation and communication costs of Alg. 3 and provide a comparison with Gram algorithm used by TuckerMPI [6, Algorithm 4]. Here assume a generic processor grid, with $P^{\circledast}$ total processors and P_n in each mode-n processor fiber. We also use generic tensor dimensions $\{J_n\}$, as the algorithm will be called repeatedly within ST-HOSVD with some modes already truncated to dimension R_n and others still the full I_n.

The first computation occurs in lines 5 or 8 and corresponds to an LQ decomposition of the local tensor unfolding of dimension $J_n \times J_n^{\circledleft} J_n^{\circledright}/P^{\circledast}$ (recall that the redistribution ensures that the global unfolding is in a 1D column distribution). In either case, the number of flops is $2J_n^2 J_n^{\circledleft} J_n^{\circledright}/P^{\circledast} - 2/3J_n^3$. The other local computation is performed in lines 14 or 16 and occurs $\log P^{\circledast}$ times. Each operation is the LQ decomposition of a $J_n \times 2J_n$ matrix (with structure) and costs $O(J_n^3)$ flops. Thus, the overall computational cost of Alg. 3 is

$$\gamma \cdot \left(2J_n J^{\circledast}/P^{\circledast} + O(J_n^3 \log P^{\circledast})\right). \tag{9}$$

Communication occurs in lines 7 and 12. Following [6], we will assume the all-to-all redistribution is performed with a point-to-point algorithm using $P_n - 1$ messages per processor, each of the size of the local tensor unfolding divided by P_n. The triangle exchange between partner processors occurs $\log P^{\circledast}$ times and consists of $O(J_n^2)$ data. The overall communication costs are then

$$\beta \cdot \left(J^{\circledast}/P^{\circledast} + J_n^2 \log P^{\circledast}\right) + \alpha \cdot O\left(P_n + \log P^{\circledast}\right). \tag{10}$$

The Gram algorithm used by TuckerMPI [6, Alg. 4] has costs

$$\gamma \cdot J_n J^{\circledast}/P^{\circledast} + \beta \cdot O\left(J^{\circledast}/P^{\circledast} + J_n^2\right) + \alpha \cdot O\left(P_n\right). \tag{11}$$

Comparing eq. (9) and eq. (11), we see that Alg. 3 performs roughly twice the flops of the Gram algorithm and includes another typically lower order term. From eq. (10) and eq. (11), note that Alg. 3 also has higher communication costs, though the increases occur in typically lower order terms.

In terms of the overall ST-HOSVD algorithm, the differences in costs between using Gram-SVD and QR-SVD are tightened, because both approaches share TTM algorithms and redistribution costs. The most significant difference comes from the increase in the flops for local Gram/LQ operations, which dominate the SVD computations and constitute a large fraction of the ST-HOSVD run time. Thus, for small numbers of processors, when the algorithm is compute bound, we expect no more than a 2× slowdown from using the more numerically accurate QR-SVD algorithm. For very large numbers of processors, when the algorithm becomes more communication bound (latency bound in particular), we also expect to see a slight slowdown from QR-SVD. However, we emphasize that the better stability of QR-SVD allows the algorithm to use lower working precision and obtain the same accuracy in most cases. We will see in Sec. 4 that the performance benefits of lower precision more than compensates for the higher flop counts and communication costs.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We conduct our numerical experiments on Andes, a 704-node cluster at Oak Ridge Leadership Computing Facility. Each node has 256 GB of memory, and two 16-core AMD EPYC 7302 processors running at 3 GHz. The peak flop rate of each core is 48 GFLOPS in double precision and 96 GFLOPS in single precision. We link our implementation to Intel MKL v19.0.3 and OpenMPI v4.0.4. Our experimental results report the minimum times over 5 trials for each algorithm and configuration.

To illustrate variability of efficiency of local computations, we also use a local server with an Intel Cascade Lake processor that has 2 sockets each with 8 physical cores. On the Cascade Lake server, we use Intel MKL v2021.1.1 and Intel MPI v2021.1.1.

Time breakdowns are reported according to the breakdown on the slowest processor. We show a breakdown of time across LQ/Gram, SVD/EVD, and TTM computations. The computations of

each mode are ordered from 0 at the bottom to $N-1$ at the top, so they do not necessarily correspond to actual order of computation.

4.2 Performance Tuning

The parallel ST-HOSVD algorithm, independent of the SVD algorithm used, has two main opportunities for performance tuning: mode ordering and processor grid dimensions. In this section we discuss the implications of these tuning configurations on the QR-SVD approach and describe the strategies we use in later experiments. Fig. 2 presents time breakdowns over different configurations of mode ordering and processor grids to demonstrate their effects on the two different platforms.

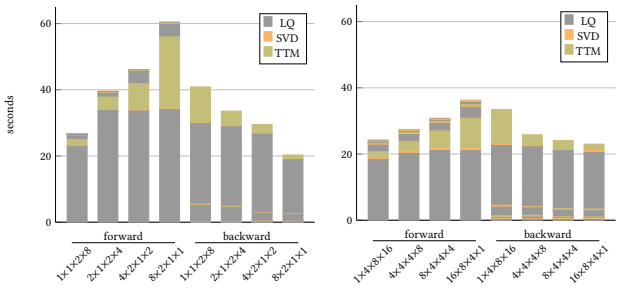
4.2.1 LAPACK Subroutines. An important factor for local computational efficiency is the underlying implementation of LAPACK QR-based subroutines. In order to respect the data layout of the tensor unfoldings (and their submatrices), we use both QR and LQ variants of the orthogonal factorizations. We use the driver routines `geqr` and `gelq` for orthogonal factorizations of any submatrix that is row- or column-major, even if they are very short and fat. Our microbenchmarks indicated that the TSQR-based computational subroutines sometimes provided an improvement over the driver routines, but because those improvements were not consistent across platforms, we chose to use the driver routines to maximize performance portability. For the structured orthogonal factorizations, we use `tpqrt`, which is designed for a triangular block stacked on top of a pentagonal block (which might be rectangular or triangular), and we observe that its use is not performance critical.

We noticed that `geqr` routinely outperformed `gelq` on Cascade Lake; this distinction is equivalent to performing a QR decomposition of a matrix stored in column- vs row-major layout. As indicated in Algs. 2 and 3, `geqr` is applied to the entire unfolding if $n = N-1$ (the last mode) and $P_{N-1} = 1$, and otherwise, `gelq` is used. Thus, we seek to set $P_{N-1} = 1$ and process the last mode early in the ordering in order to benefit from the performance of `geqr`. We suspect, based on the working memory requested, that the underlying implementation of `gelq` for this version of MKL performs an explicit transpose before calling `geqr`. We did not observe the same behavior on Andes; the two routines exhibited similar performance.

4.2.2 Processor Grid. The processor grid configuration can affect both communication and computation efficiency. As noted in prior work [6], the processor grid changes how much data and how many messages are communicated throughout the algorithm. If a processor grid dimension is set to 1, then some communication (including the redistribution in Alg. 3) is avoided altogether. Computation and communication costs tend to decrease through the course of ST-HOSVD as the size of the tensor is reduced. Thus, it is generally beneficial to set processor grid dimensions to 1 (or other small values) in the modes that are processed first.

4.2.3 Mode Ordering. The mode ordering can have a significant impact on the amount of computation performed by ST-HOSVD. For example, significant truncation in one mode will result is less computation required by later modes. If all dimensions and reduced ranks are known at the start of the algorithm, the modes can be ordered to minimize computation or other metrics [6]. However, if the ranks are determined at run time based on the error tolerance, then only heuristics can be used. Mode ordering also has an effect on computational efficiency, as described above, as it can affect how much computation is performed by the most efficient LAPACK subroutines.

In this work, we assume ranks are not known a priori, so that the optimal mode ordering is not known. We consider the data in the mode order used to store it on disk and consider only two orderings: forward and backward. In this way, we can focus on tuning the performance of the first mode truncated, which corresponds to an unfolding that is a contiguous matrix in memory.



(a) Cascade Lake: $300\times300\times300\times300$ **tensor using 16 cores**

(b) Andes: $500\times500\times500\times500$ **tensor using 512 cores**

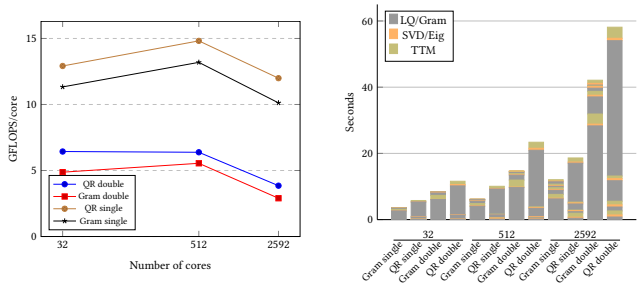
Figure 2: Time breakdown over various configurations

4.2.4 Cross-Platform Results. We consider the results using various configurations on Cascade Lake (Fig. 2a) and Andes (Fig. 2b). In each case, we consider forward and backward mode orderings and processor grids that range from back-loaded to front-loaded.

On Cascade Lake, we use 16 MPI processes on a $300\times300\times300\times300$ tensor and reduce it to a core of dimension $30\times30\times30\times30$. Note that because all dimensions are consistent across modes, mode ordering does not affect the amount of computation. In all cases, more than half of the overall time is spent in the first LQ computation (mode 0 for forward and mode 3 for backward). In each of the mode orderings, the fastest processor grid configuration is the one with the corresponding mode's processor dimension set to 1. We see that the backward ordering with grid $8\times2\times1\times1$ is faster than the forward ordering with grid $1\times1\times2\times8$, which is because of the superior performance of `geqr` over `gelq` in the LQ.

On Andes, we use 512 MPI processes (16 nodes) on a $500\times500\times500\times500$ tensor and reduce it to a core of dimension $50\times50\times50\times50$. We observe the same pattern of time dominated by the first mode truncated and better performance for grids with dimension 1 in that mode. However, there is little difference between a forward order-processor grid configuration and its backward order mirrored counterpart, due to the similar performance across `geqr` and `gelq` on the AMD nodes.

We note that benchmarks on TuckerMPI's original Gram-SVD approach showed similar behavior across processor grids. There was very little difference between forward and backward orderings for these synthetic cubical tensors.



(a) Performance

(b) Time breakdown

Figure 3: Weak scaling across algorithms and precisions

# of Cores	# of Nodes	QR single/double	Gram single/double
32	1	$4 \times 4 \times 2 \times 1$	$1 \times 1 \times 2 \times 16$
64	2	$8 \times 4 \times 2 \times 1$	$1 \times 1 \times 4 \times 16$
128	4	$8 \times 8 \times 2 \times 1$	$1 \times 1 \times 8 \times 16$
256	8	$16 \times 8 \times 2 \times 1$	$1 \times 1 \times 16 \times 16$
512	16	$16 \times 8 \times 4 \times 1$	$1 \times 2 \times 16 \times 16$
1024	32	$16 \times 16 \times 4 \times 1$	$1 \times 4 \times 16 \times 16$
2048	64	$32 \times 16 \times 4 \times 1$	$1 \times 4 \times 16 \times 32$

Table 1: Strong scaling processor grids

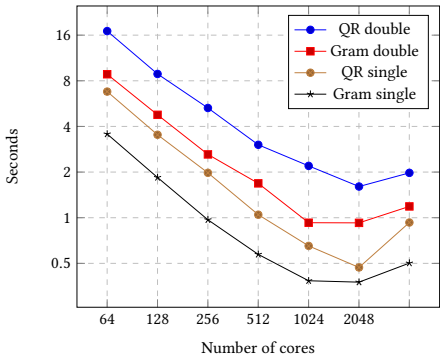


Figure 4: Strong scaling across algorithms and precisions

4.3 Weak Scaling Comparison

To assess the weak scalability of our approach and compare with TuckerMPI, we generate synthetic tensors following the experiment of prior work [6, Section 9.4.3] and report results in Fig. 3. We use a random tensor of dimension $250k \times 250k \times 250k \times 250k$ distributed across k^4 nodes for $k \in \{1, 2, 3\}$, and compress it to core dimensions $25k \times 25k \times 25k \times 25k$. The local tensor data remains fixed at approximately 1 GB throughout the experiment. For Gram-SVD, we use forward ordering with a $1 \times 2k \times 4k \times 4k^2$ processor grid (as suggested [6]) and backward ordering with a $4k^2 \times 4k \times 2k \times 1$ grid for QR-SVD. We use identical tensor dimensions and ranks in each mode to control for computational variations across mode orderings. We are not able to perform the experiment for $k = 4$ because it requires too high a fraction of Andes' nodes.

Note that although the local tensor contains the same amount of data as k increases, the number of columns of each unfolding is increasing. This results in an increase in computation, which leads to an increase in overall runtime. In Fig. 3a, we use GFLOPS/Core performance as a gauge of the weak scaling of the QR-SVD and Gram-SVD algorithms in single and double precision. On a single node, QR-SVD achieves 6.4 GFLOPS in double 13 GFLOPS in single precision (both about 14% of peak). On 81 nodes, QR-SVD maintains 3.8 GFLOPS in double and 12 GFLOPS in single precision (8% and 12.5% of peak, respectively). Recall that QR-SVD performs more flops than Gram-SVD; in this experiment, it performs approximately 83% more flops. While QR-SVD takes more time than Gram-SVD in each precision, we observe that its performance is slightly better and scales similarly.

We present the time breakdown in Fig. 3b. The fastest algorithm is Gram in single precision, QR in single precision is faster than Gram in double precision (TuckerMPI's implementation), and QR in double precision is the slowest. As expected, more than half the time is spent in the first LQ or Gram operation that involves very little communication, which implies that the overall performance is determined in large part by the performance of the local LQ (`geqr`) or Gram (`syrk`) performance. Compared to the similar experiment in [6], we see lower performance for Gram-SVD, which we attribute to suboptimal BLAS/LAPACK implementations available on Andes. We use Intel's MKL on an AMD architecture because it has the best performance over the available libraries in our microbenchmarks; we expect that AMD's ACML would have better performance but is not available. Based on this experiment, we expect that our implementation can scale at least as well as TuckerMPI as long as the LAPACK implementation is as well optimized as the BLAS implementation.

4.4 Strong Scaling Comparison

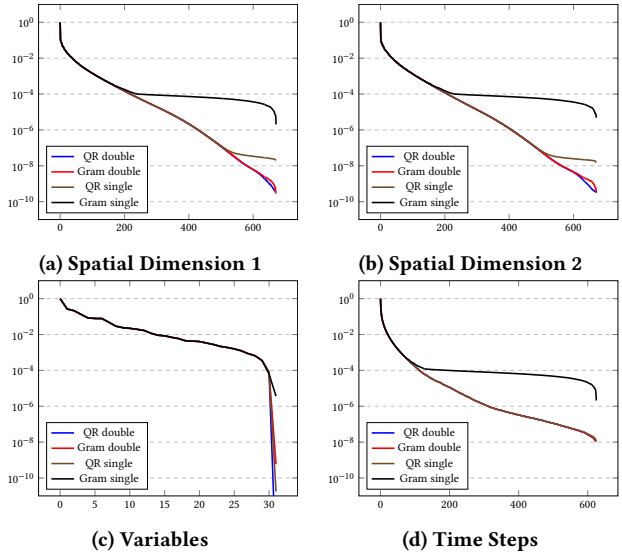
Similar to weak scaling, we use a randomly generated synthetic tensor with dimension $256 \times 256 \times 256 \times 256$ and compress it down to a $32 \times 32 \times 32 \times 32$ core. For the fixed input, we scale the number of nodes from 1 to 64 using the processor grids specified in Tab. 1 and forward and backward ordering for Gram and QR, respectively. The results are given in Fig. 4, and again we see the times decreasing from QR double to Gram double to QR single to Gram single. The scaling is consistent across algorithms and precisions, and all scale to 32 nodes for this problem, which is consistent with prior work [6]. We note that QR in single precision is consistently faster than Gram in double precision, typically about 30% faster and achieves up to a 2× speedup at 32 nodes, and the two algorithms achieve nearly the same accuracy.

4.5 Application Datasets

4.5.1 Dataset Descriptions. In this section we compare the use of Gram-SVD and QR-SVD in single and double precision on application tensor data. We use three datasets in the following experiments:

- *HCCI*: the Homogeneous Charge Compression Ignition (HCCI) dataset is generated from a numerical simulation of combustion [8] and used in prior work [4]. The dimension of the dataset is $627 \times 627 \times 33 \times 627$. Similar to SP, the first two modes corresponds to spatial dimensions, the 3rd mode corresponds to 33 variables, and the last mode 627 time steps.

- *SP*: the Stats-Planar (SP) dataset is the result from a numerical simulation of methane-air combustion [18]. The dimension of the datasets is $500 \times 500 \times 500 \times 11 \times 100$, the first three modes correspond to the three spatial dimensions, the 4th mode corresponds to 11 variables, and the last mode consists of 100 time steps (we use the first 100 of the available 400 time steps). This dataset has also been used in prior work [4, 6].
- *Video*: the video dataset is used in [21] for frame classification. The dimensions are $1080 \times 1920 \times 3 \times 2200$ tensor with the 4 modes corresponding to frame height, frame width, color channel, and frame index.



(a) Spatial Dimension 1 (b) Spatial Dimension 2

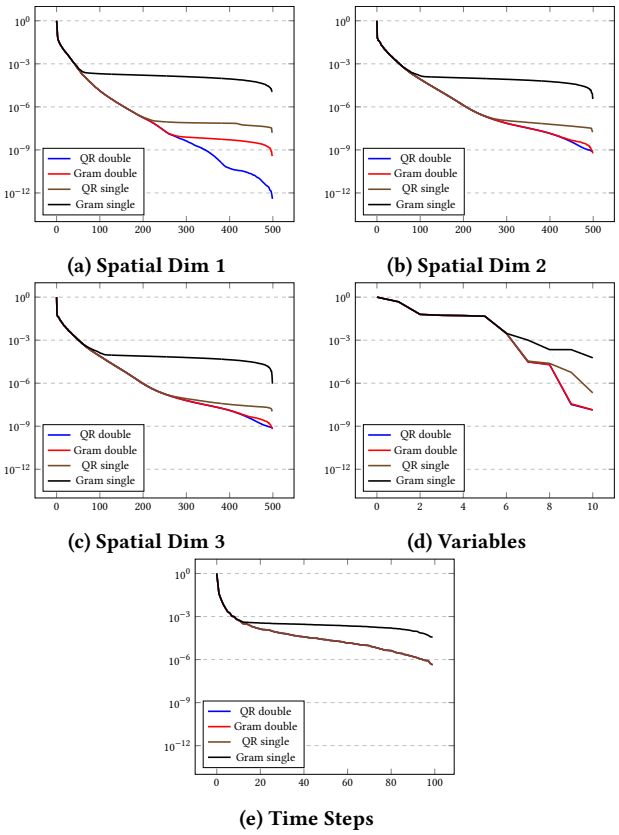
(c) Variables (d) Time Steps

Figure 5: HCCI singular values

4.5.2 Singular Value Accuracy. In order to understand the compressibility of each dataset and compare the accuracy of the various algorithms and precisions, we use the ST-HOSVD algorithm without compression. The per-mode singular values are shown in Figs. 5 to 7. Note that all values are normalized so that the first singular value in each mode is set to 1 to make the relative error estimations easier. We first observe the compressibility of the combustion datasets: the singular values span a wide range, and reasonable error tolerances can yield large compression, as we will see below. The video dataset has rapid decay of about 2 orders of magnitude in 3 of its modes, but then the singular values decay very slowly, offering little compressibility at tight error tolerances. Next, like the matrix experiment considered in Sec. 3.2, we see the loss of accuracy of computed singular values for each method except for QR double. If the error tolerance is tighter than the accuracy of the singular values allows, ST-HOSVD will return unreliable results.

4.5.3 ST-HOSVD Comparison. In the following experiments, we apply the ST-HOSVD algorithm to each data set to compare the time performance and accuracy of the algorithms and precisions for various error tolerances.

Fig. 8 shows the result of compressing the HCCI dataset with tolerances of 10^{-2}, 10^{-4}, 10^{-6}, and 10^{-8}. The values plotted in

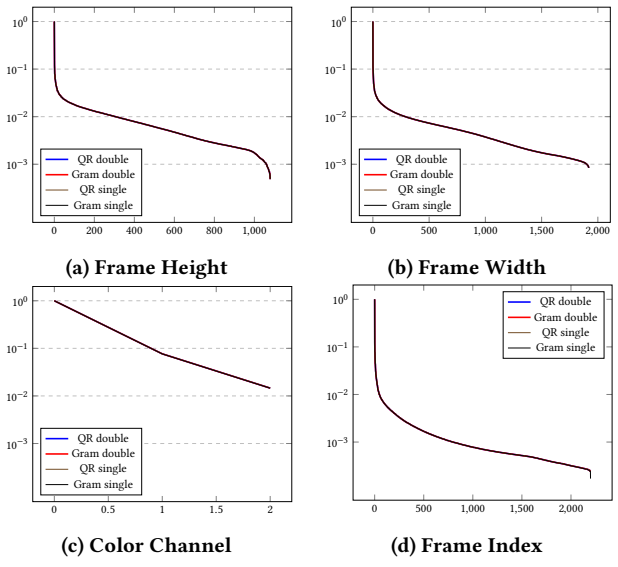


(a) Spatial Dim 1 (b) Spatial Dim 2

(c) Spatial Dim 3 (d) Variables

(e) Time Steps

Figure 6: SP singular values

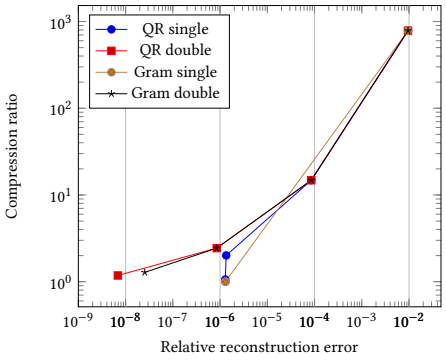


(a) Frame Height (b) Frame Width

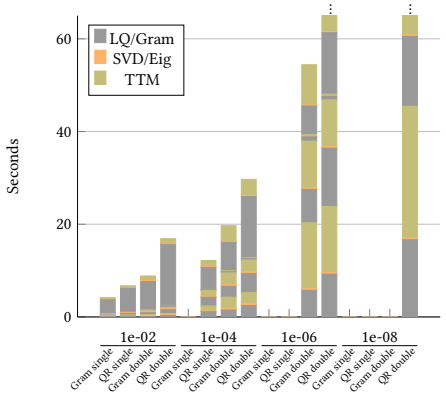
(c) Color Channel (d) Frame Index

Figure 7: Video data singular values

Fig. 8a are also given in Tab. 2. In this experiment we use 4 nodes, backwards mode ordering, and a $16 \times 8 \times 1 \times 1$ processor grid for both QR and Gram. Consider the error tolerance 10^{-2}: from Fig. 8a and Tab. 2 we see that all algorithms attain the same compression ratio and approximation error, which satisfies the desired tolerance. In Fig. 8b we see that Gram single is the fastest algorithm and can be safely used, finishing about twice as fast as Gram double (original TuckerMPI code). However, at error tolerance 10^{-4}, Gram single cannot compute singular values accurately enough and it fails to compresses the data at all. The algorithm computes square factor matrices (corresponding to a full HOSVD [19]) and incurs an error of 1.3e-6 due to multiplications in single precision. Thus, we do not report the time for Gram single at this tolerance, and we see that QR single is the fastest algorithm among the rest, offering a 60% speedup over Gram double. Likewise, the tolerance 10^{-6} represents a range where QR single is not sufficiently accurate, and double Gram is preferred, and at tolerance 10^{-8} and smaller, QR double is the only method accurate enough. However, for this dataset, such a tight tolerance offers practically no compression.



(a) Compression vs error



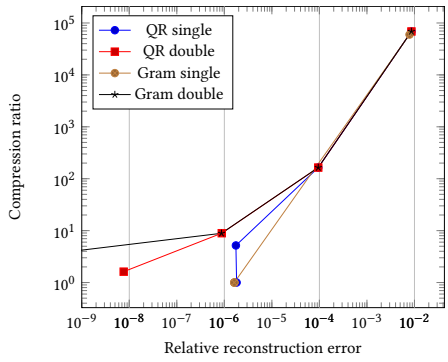
(b) Time breakdown

Figure 8: HCCI dataset time and accuracy comparisons

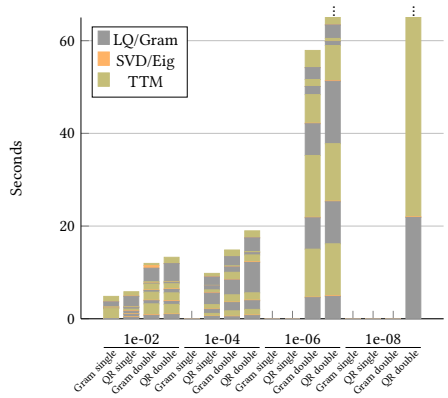
We present the results for the SP dataset in Fig. 9 and Tab. 3. Again, we consider error tolerances of 10^{-2}, 10^{-4}, 10^{-6}, and 10^{-8}, and we observe similar behavior to HCCI, though SP is much larger and more compressible. In order to store the data in memory, we need 50 nodes on Andes; we use a $40 \times 20 \times 2 \times 1 \times 1$ processor grid

Tolerance	QR single		QR double		Gram single		Gram double	
	compression	error	compression	error	compression	error	compression	error
1e-02	7.85e+02	9.54e-03	7.85e+02	9.54e-03	7.85e+02	9.54e-03	7.85e+02	9.54e-03
1e-04	1.48e+01	8.54e-05	1.48e+01	8.54e-05	1.00e+00	1.30e-06	1.48e+01	8.54e-05
1e-06	2.01e+00	1.35e-06	2.44e+00	8.51e-07	1.00e+00	1.30e-06	2.44e+00	8.51e-07
1e-08	1.06e+00	1.29e-06	1.18e+00	6.85e-09	1.00e+00	1.30e-06	1.28e+00	2.54e-08

Table 2: HCCI reconstruction error and compression ratio



(a) Compression vs error



(b) Time breakdown

Figure 9: SP dataset time and accuracy comparisons

Tolerance	QR single		QR double		Gram single		Gram double	
	compression	error	compression	error	compression	error	compression	error
1e-02	6.86e+04	8.63e-03	6.86e+04	8.63e-03	5.98e+04	7.87e-03	6.86e+04	8.63e-03
1e-04	1.64e+02	9.46e-05	1.64e+02	9.46e-05	1.00e+00	1.62e-06	1.64e+02	9.46e-05
1e-06	5.18e+00	1.75e-06	8.92e+00	8.82e-07	1.00e+00	1.62e-06	8.88e+00	8.70e-07
1e-08	1.00e+00	1.81e-06	1.62e+00	7.62e-09	1.00e+00	1.62e-06	1.00e+00	2.18e-15

Table 3: SP reconstruction error and compression ratio

and backwards ordering for all algorithms. At 10^{-2}, Gram single is the method of choice because it is the fastest and all algorithms are sufficiently accurate. For the tighter tolerance of 10^{-4}, QR single is the fastest sufficiently accurate method, outperforming TuckerMPI (Gram double) by 50%, and at 10^{-8}, QR double is the only method that is accurate enough.

The video dataset has a much shorter range in singular values, but it is still very compressible when the error tolerance is large enough. In this experiment, we follow prior work [1] and specify the rank to be $200 \times 200 \times 3 \times 200$, resulting in 570× compression. All four variants achieve the same relative error of 0.213, which is also consistent with prior work. Although the relative error is higher, the recovered image still retains recognizable features and is sufficient for the classification task. The time breakdown is shown

in Fig. 10, where we see that Gram single should be used since it is fastest and as accurate as the other variants, offering a 2.2× speedup over the original TuckerMPI code.

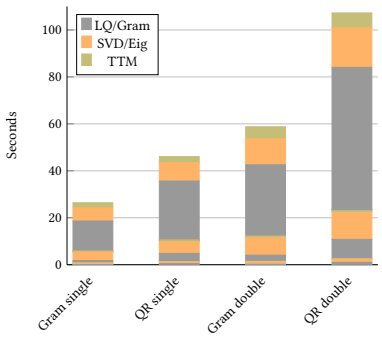


Figure 10: Video dataset time breakdown

5 CONCLUSION

In this work, we generalize the TuckerMPI library to be able to perform ST-HOSVD with the more numerically stable QR-SVD algorithm as well as the computationally cheaper Gram-SVD algorithm. We also enable the use of either single or double precision using C++ templates.

Our numerical results demonstrate that the existing implementation is eclipsed by these generalizations, either in terms of speed or accuracy. For large error tolerances of 10^{-3} or larger, the single precision Gram-SVD is the fastest method, approximately twice as fast as TuckerMPI. For error tolerances between 10^{-3} and 10^{-7}, the single precision QR-SVD is faster by 50-100%. For error tolerances tighter than 10^{-8}, double precision QR-SVD is required to obtain the desired approximation error. Only in the tight range around 10^{-7} to 10^{-8} is the existing implementation the method of choice.

However, we point out that there are still limitations for our implementation. In particular, for tensors with modes that have very large dimension, of 10,000 or more for example, the sequential SVD will be a computational bottleneck and eventually become infeasible. Handling such cases would require parallelizing the SVD of the triangular factor, or perhaps using a different approach to the SVD of the unfolding altogether. For large tolerances where Gram single is the preferred method, alternatives such as randomized and iterative algorithms are likely to be competitive and should be compared against. In future work, we also plan to explore the use of mixed precision within the Gram-SVD algorithm.

ACKNOWLEDGMENTS

The authors would like to thank Hussam Al Daas and Arvind Saibaba for helpful discussions on performance tuning and numerical analysis.

This work is supported by the National Science Foundation under Grant No. CCF-1942892 and the Wake Forest University Undergraduate Research and Creative Activities Center.

REFERENCES

[1] S. Ahmadi-Asl et al. 2021. Randomized Algorithms for Computation of Tucker Decomposition and Higher Order SVD (HOSVD). *IEEE Access* 9 (2021), 28684–28706. https://doi.org/10.1109/ACCESS.2021.3058103

[2] H. Al Daas, G. Ballard, and P. Benner. 2020. *Parallel Algorithms for Tensor Train Arithmetic.* Technical Report. arXiv. https://arxiv.org/abs/2011.06532

[3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. 2011. Communication-Avoiding QR Decomposition for GPUs. In *IPDPS*. IEEE Computer Society, 48–58. https://doi.org/10.1109/IPDPS.2011.15

[4] W. Austin, G. Ballard, and T. G. Kolda. 2016. Parallel Tensor Compression for Large-Scale Scientific Data. In *IPDPS*. 912–922. https://doi.org/10.1109/IPDPS.2016.67

[5] G. Ballard, J. Demmel, L. Grigori, N. Knight, M. Jacquelin, and H. D. Nguyen. 2015. Reconstructing Householder Vectors from Tall-Skinny QR. *J. Parallel and Distrib. Comput.* 85 (August 2015), 3–31. https://doi.org/10.1016/j.jpdc.2015.06.003

[6] G. Ballard, A. Klinvex, and T. 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. https://dl.acm.org/doi/10.1145/3378445

[7] R. Ballester-Ripoll and R. Pajarola. 2015. Lossy volume compression using Tucker truncation and thresholding. *The Visual Computer* 32 (2015), 1433–1446. https://doi.org/10.1007/s00371-015-1130-y

[8] A. Bhagatwala, J. H. Chen, and T. Lu. 2014. Direct numerical simulations of HCCI/SACI with ethanol. *Combustion and Flame* 161, 7 (2014), 1826–1841. https://doi.org/10.1016/j.combustflame.2013.12.027

[9] 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 *IPDPS*. 1038–1047. https://doi.org/10.1109/IPDPS.2017.86

[10] J. Choi, X. Liu, and V. Chakaravarthy. 2018. High-performance Dense Tucker Decomposition on GPU Clusters. In *SC*. IEEE Press, Article 42, 11 pages. https://dl.acm.org/doi/10.1109/SC.2018.00045

[11] P. G. Constantine and D. F. Gleich. 2011. Tall and Skinny QR Factorizations in MapReduce Architectures. In *MapReduce*. ACM, 43–50. https://doi.org/10.1145/1996092.1996103

[12] J. Demmel. 1997. *Applied Numerical Linear Algebra.* SIAM.

[13] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM J. on Sci. Comp.* 34, 1 (2012), A206–A239. https://doi.org/10.1137/080731992

[14] Petros Drineas and Ilse C. F. Ipsen. 2019. Low-Rank Matrix Approximations Do Not Need a Singular Value Gap. *SIAM J. Matrix Anal. Appl.* 40, 1 (2019), 299–319. https://doi.org/10.1137/18M1163658

[15] G.H. Golub and C.F. Van Loan. 2013. *Matrix Computations.* JHU Press.

[16] K. Hayashi, G. Ballard, Y. Jiang, and M. J. Tobia. 2018. Extended Abstract: Shared-memory Parallelization of MTTKRP for Dense Tensors. In *PPoPP*. ACM, 393–394. http://doi.acm.org/10.1145/3178487.3178522

[17] T. G. Kolda and B. W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (September 2009), 455–500. https://doi.org/10.1137/07070111X

[18] H. Kolla, X.-Y. Zhao, J. H. Chen, and N. Swaminathan. 2016. Velocity and Reactive Scalar Dissipation Spectra in Turbulent Premixed Flames. *Comb. Sci. and Tech.* 188, 9 (2016), 1424–1439. https://doi.org/10.1080/00102202.2016.1197211

[19] L. De Lathauwer, B. De Moor, and J. Vandewalle. 2000. A Multilinear Singular Value Decomposition. *SIAM J. Matrix Anal. Appl.* 21, 4 (2000), 1253–1278. https://doi.org/10.1137/S0895479896305696

[20] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. 2015. An Input-Adaptive and In-Place Approach to Dense Tensor-Times-Matrix Multiply. In *SC*. ACM, Article 76, 12 pages. https://doi.org/10.1145/2807591.2807671

[21] O. A. Malik and S. Becker. 2018. Low-Rank Tucker Decomposition of Large Tensors Using TensorSketch. In *NeurIPS*, Vol. 31. https://dl.acm.org/doi/10.5555/3327546.3327674

[22] R. Minster, A. K. Saibaba, and M. E. Kilmer. 2020. Randomized Algorithms for Low-Rank Tensor Decompositions in the Tucker Format. *SIAM J. on Math. of Data Sci.* 2, 1 (2020), 189–215. https://doi.org/10.1137/19M1261043

[23] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. 2009. Minimizing communication in sparse matrix solvers. In *SC*. Article 36, 12 pages. https://doi.org/10.1145/1654059.1654096

[24] A.-H. Phan, P. Tichavsky, and A. Cichocki. 2013. Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations. *IEEE TSP* 61, 19 (Oct 2013), 4834–4846. https://doi.org/10.1109/TSP.2013.2269903

[25] Y. Sun, Y. Guo, C. Luo, J. Tropp, and M. Udell. 2020. Low-Rank Tucker Approximation of a Tensor from Streaming Data. *SIAM J. on Math. of Data Sci.* 2, 4 (2020), 1123–1150. https://doi.org/10.1137/19M1257718

[26] L. N. Trefethen and D. Bau. 1997. *Numerical Linear Algebra.* SIAM.

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

[28] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen. 2012. A New Truncation Strategy for the Higher-Order Singular Value Decomposition. *SIAM J. on Sci. Comp.* 34, 2 (2012), A1027–A1052. https://doi.org/10.1137/110836067