SIAM J. SCI. COMPUT.
Vol. 44, No. 1, pp. C25–C53

PARALLEL ALGORITHMS FOR TENSOR TRAIN ARITHMETIC*

HUSSAM AL DAAS†, GREY BALLARD‡, AND PETER BENNER†

Abstract. We present efficient and scalable parallel algorithms for performing mathematical operations for low-rank tensors represented in the tensor train (TT) format. We consider algorithms for addition, elementwise multiplication, computing norms and inner products, orthonormalization, and rounding (rank truncation). These are the kernel operations for applications such as iterative Krylov solvers that exploit the TT structure. The parallel algorithms are designed for distributed-memory computation, and we propose a data distribution and strategy that parallelizes computations for individual cores within the TT format. We analyze the computation and communication costs of the proposed algorithms to show their scalability, and we present numerical experiments that demonstrate their efficiency on both shared-memory and distributed-memory parallel systems. For example, we observe better single-core performance than the existing MATLAB TT-Toolbox in rounding a 2GB TT tensor, and our implementation achieves a 34× speedup using all 40 cores of a single node. We also show nearly linear parallel scaling on larger TT tensors up to over 10,000 cores for all mathematical operations.

Key words. low-rank tensor format, tensor train, parallel algorithms, QR, SVD

AMS subject classifications. 15A69, 15A23, 65Y05, 65Y20

DOI. 10.1137/20M1387158

1. Introduction.

Multidimensional data, or tensors, appear in a variety of applications where numerical values represent multiway relationships. The tensor train (TT) format is a low-rank representation of a tensor that has been applied to solving problems in areas such as parameter-dependent PDEs, stochastic PDEs, molecular simulations, uncertainty quantification, data completion, and classification [9, 10, 19, 22, 32, 34, 41, 46]. As the number of dimensions or modes of a tensor becomes large, the total number of data elements grows exponentially fast, which is known as the curse of dimensionality [22]. Fortunately, it can be shown in many cases that the tensors exhibit low-rank structure and can be represented or approximated by significantly fewer parameters. Low-rank tensor approximations allow for storing the data implicitly and performing arithmetic operations in feasible time and space complexity, avoiding the curse of dimensionality.

In contrast to the matrix case where the singular value decomposition (SVD) provides optimal low-rank representations, there are more diverse possibilities for low-rank representations of tensors [30]. Various representations have been proposed, such as CANDECOMP/PARAFAC (CP) [15, 23], Tucker [52], quantized tensor train [29], and hierarchical Tucker [22], in addition to TT [41], and each has been demonstrated to be most effective in certain applications. The TT format, which is also known as the matrix product state (MPS) in the computational physics and chemistry communities, consists of a sequence of TT cores, one for each tensor dimension, and each core is a 3-way tensor except for the first and last cores, which are matrices. The primary

*Submitted to the journal's Software and High-Performance Computing section December 21, 2020; accepted for publication (in revised form) October 12, 2021; published electronically February 3, 2022.

https://doi.org/10.1137/20M1387158

†Department of Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, 39106, Germany (aldaas@mpi-magdeburg.mpg.de, benner@mpi-magdeburg.mpg.de).

‡Computer Science Department, Wake Forest University, Winston Salem, NC, 27106 USA (ballard@wfu.edu).

advantages of TT are that (1) the number of parameters in the representation is linear, rather than exponential, in the number of modes and (2) the representation can be computed to satisfy a specified approximation error threshold in a numerically stable way.

As these low-rank tensor techniques have been applied to larger and larger data sets, efficient sequential and parallel implementations of algorithms for computing and manipulating these formats have also been developed. Toolboxes and libraries in productivity-oriented languages such as MATLAB and Python [4, 31, 39, 54] are available for moderately sized data, and parallel algorithms implemented in performance-oriented languages exist for computation of decompositions such as CP [20, 48, 36] and Tucker [3, 8, 28, 47] and operations such as tensor contraction [49], allowing for scalability to much larger data and numbers of processors. While efficient computation of TT approximations of explicit tensors has attracted recent attention [13, 21, 37, 45, 55], no such high-performance parallel implementations exist for approximating tensors already in TT format. In condensed matter computations, several advances have been made in parallelizing the density matrix renormalization group (DMRG) algorithm, which computes the ground-state eigenvector in MPS/TT format [27, 35, 50]. The modes' dimensions in these applications are very small and the TT ranks can be very large. In contrast, applications from parameter-dependent PDEs, stochastic PDEs, uncertainty quantification, and molecular simulations [10, 11, 32] yield computations with TT tensors having certain modes with very large dimensions and relatively small TT ranks. The goal of this work is to establish efficient and scalable algorithms for implementing the key mathematical operations on TT tensors for applications where at least one mode has a very large dimension and the TT ranks are relatively small to allow researchers to scale their models beyond the time and memory constraints when using current MATLAB and Python implementations.

We consider mathematical operations such as addition, Hadamard (elementwise) multiplication, computing norms and inner products, left- and right-orthonormalization, as well as rounding (rank truncation). These are the operations required to, for example, solve a structured linear system whose solution can be approximated well by a tensor in TT format using a Krylov method [34]. As we will see in section 2, mathematical operations can increase the ranks of the TT representation of the result tensor, which can then be recompressed, or rounded back to smaller ranks, in order to maintain feasible time and space complexity with some controllable loss of accuracy. As a result, the rounding procedure (and the orthonormalization it requires) is of prime importance in developing efficient and scalable TT algorithms. We will assume throughout that full tensors are never formed explicitly.

In order to develop scalable parallel algorithms, we propose a data distribution and parallelization techniques that maintain computational load balance and attempt to minimize interprocessor communication, which is the most expensive operation on parallel machines in terms of both time and energy consumption. As discussed in section 3, we distribute the slices of each TT core across all processors, where slices are matrices (or vectors) whose dimensions are determined by the low ranks of the TT representation. This distribution allows for full parallelization of each corewise computation and avoids the need for communication within slicewise computations. The orthonormalization and rounding algorithms depend on parallel QR decompositions, and our approach enables the use of the Tall-Skinny QR algorithm, which is communication optimal for the matrix dimensions in this application [18]. We analyze the parallel computation and communication costs of each TT algorithm, demonstrating that the bulk of the computation is load balanced perfectly across processors.

The communication costs are independent of the original tensor dimensions, so their relative costs diminish with small ranks.

We verify the theoretical analysis and benchmark our C/MPI implementation on up to 256 nodes (10,240 cores) of a distributed-memory parallel platform in section 4. Our experiments are performed on synthetic data using tensor dimensions and ranks that arise in a variety of scientific and data analysis applications. On a shared-memory system (one node of the system), we compare our TT-rounding implementation against the TT-Toolbox [39] in MATLAB and show that our implementation is 70% more efficient using a single core and achieves up to a 34× parallel speedup using all 40 cores on the node. We also present strong scaling performance experiments for computing inner products, norms, orthonormalization, and rounding using up to over 10K MPI processes. The experimental results show that the time remains dominated by local computation even at that scale, allowing for nearly linear scaling for multiple operations, achieving for example a 97× speedup of TT-rounding when scaling from 1 node to 128 nodes on a TT tensor with a 28 GB memory footprint. We conclude in section 5 and discuss limitations of our approaches and perspectives for future improvements.

2. Notation and background. In this section, we review the TT format and present a brief overview of the notation and computational kernels associated with it. Tensors are denoted by boldface Euler script letters (e.g., $\mathcal{X}$), and matrices are denoted by boldface block letters (e.g., $\mathbf{A}$). The number I_n for $1 \leq n \leq N$ is referred to as the mode size or mode dimension, and we use i_n to index that dimension. The order of a tensor is its number of modes, e.g., the order of $\mathcal{X}$ is N. The nth TT core (described below) of a tensor $\mathcal{X}$ is denoted by $\mathcal{T}_{\mathcal{X},n}$. We use MATLAB-style notation to obtain elements or subtensors, where a solitary colon (:) refers to the entire range of a dimension. For example $\mathcal{X}(i,j,k)$ is a tensor entry, $\mathcal{X}(i,:,:)$ is a tensor slice (a matrix in this case), and $\mathcal{X}(:,j,k)$ is a tensor fiber (a vector).

The mode-n "modal" unfolding (or matricization or flattening) of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is the matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times \frac{I}{I_n}}$, where $I = I_1 I_2 I_3$. In this case, the columns of the modal unfolding are fibers in that mode. The mode-n product or tensor-times-matrix operation is denoted by $\times_n$ and is defined so that the mode-n unfolding of $\mathcal{X} \times_n \mathbf{A}$ is $\mathbf{A}\mathbf{X}_{(n)}$. We refer to [30, 44] for more details.

The norm of a tensor is defined so that $\|\mathcal{X}\|^2 = \sum_{i_1,\ldots,i_N} \mathcal{X}(i_1,\ldots,i_N)^2$, which generalizes the vector 2-norm and matrix Frobenius norm.

2.1. TT tensors. A tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is in the TT format if there exist strictly positive integers $R_0, \ldots, R_N$ with $R_0 = R_N = 1$ and N order-3 tensors $\mathcal{T}_{\mathcal{X},1}, \ldots, \mathcal{T}_{\mathcal{X},N}$, called TT cores, with $\mathcal{T}_{\mathcal{X},n} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$, such that

$$\mathcal{X}(i_1,\ldots,i_N) = \mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{X},n}(:,i_n,:)\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N).$$

We note that because $R_0 = R_N = 1$, the first and last TT cores are (order-2) matrices so $\mathcal{T}_{\mathcal{X},1}(i_1,:) \in \mathbb{R}^{R_1}$ and $\mathcal{T}_{\mathcal{X},N}(:,i_N) \in \mathbb{R}^{R_{N-1}}$. The $R_{n-1} \times R_n$ matrix $\mathcal{T}_{\mathcal{X},n}(:,i_n,:)$ is referred to as the i_nth slice of the nth TT core of $\mathcal{X}$, where $1 \leq i_n \leq I_n$. Figure 2.1 shows an illustration of an order-5 TT tensor.

Due to the multiplicative formulation of the TT format, the cores of a TT tensor are not unique. For example, let $\mathcal{X}$ be a TT tensor, and let $\mathbf{M} \in \mathbb{R}^{R_n \times R_n}$ be an invertible matrix. Then, the TT tensor $\mathcal{Y}$ defined such that

$$\mathcal{Y}(i_1,\ldots,i_N) = \mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots(\mathcal{T}_{\mathcal{X},n}(:,i_n,:)\mathbf{M})\cdot(\mathbf{M}^{-1}\mathcal{T}_{\mathcal{X},n+1}(:,i_{n+1},:))\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N)$$

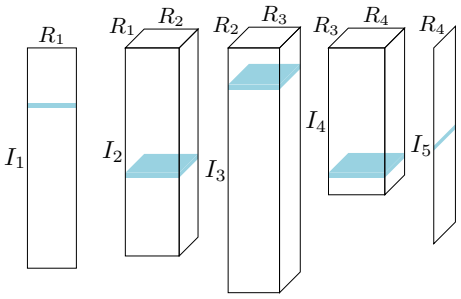


FIG. 2.1. *Order-5 TT tensor with a particular slice from each TT core highlighted. The chain product of these slices produces a scalar element of the full tensor with indices corresponding to the slices.*

is equal to $\mathcal{X}$. Another important remark is the following:

$$\begin{aligned}(2.1)\quad &\mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots(\mathcal{T}_{\mathcal{X},n}(:,i_n,:)\mathbf{M})\cdot\mathcal{T}_{\mathcal{X},n+1}(:,i_{n+1},:)\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N)\\ &\qquad=\mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{X},n}(:,i_n,:)\cdot(\mathbf{M}\mathcal{T}_{\mathcal{X},n+1}(:,i_{n+1},:))\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N)\end{aligned}$$

where $\mathbf{M}$ in this case need not be invertible. Thus, we can "pass" a matrix between adjacent cores without changing the tensor. This property is used to orthonormalize TT cores as we will see in subsection 2.3.

2.2. Unfolding TT cores. In order to express the arithmetic operations on TT cores using linear algebra, we will often use two specific matrix unfoldings of the three-dimensional (3D) tensors. The *horizontal unfolding* of TT core $\mathcal{T}_{\mathcal{X},n}$ corresponds to the concatenation of the slices $\mathcal{T}_{\mathcal{X},n}(:,i_n,:)$ for $i_n = 1,\ldots,I_n$ horizontally. We denote the corresponding operator by $\mathcal{H}$, so that $\mathcal{H}(\mathcal{T}_{\mathcal{X},n})$ is an $R_{n-1}\times R_n I_n$ matrix. The *vertical unfolding* corresponds to the concatenation of the slices $\mathcal{T}_{\mathcal{X},n}(:,i_n,:)$ for $i_n = 1,\ldots,I_n$ vertically. We denote the corresponding operator by $\mathcal{V}$, so that $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ is an $R_{n-1}I_n\times R_n$ matrix. These unfoldings are illustrated in Figure 2.2.

Note that the horizontal unfolding is equivalent to the modal unfolding with respect to the 1st mode, often denoted with subscript (1) to denote the mode that corresponds to rows [30]. Similarly, the vertical unfolding is the transpose of the modal unfolding with respect to the 3rd mode, which also corresponds to the more general unfolding that maps the first two modes to rows and the third mode to columns, denoted with subscript (1:2) to denote the modes that correspond to rows [42]. These connections are important for the linearization of tensor entries in memory and our efficient use of BLAS and LAPACK, discussed in subsection 3.1.

2.3. TT orthonormalization. Different types of orthonormalization can be defined for TT tensors. We focus in this paper on left and right orthonormalizations which are required in the rounding procedure. We use the terms column and row orthonormal to refer to matrices that have orthonormal columns and orthonormal rows, respectively, so that a matrix $\mathbf{Q}$ is column orthonormal if $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$ and row orthonormal if $\mathbf{Q}\mathbf{Q}^\top = \mathbf{I}$.

A TT tensor is said to be right orthonormal if $\mathcal{H}(\mathcal{T}_{\mathcal{X},n})$ is row orthonormal for $n = 2,\ldots,N$ (all but the first core). On the other hand, a tensor is said to be left orthonormal if $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ is column orthonormal for $n = 1,\ldots,N-1$ (all but the last core). More generally, we define a tensor to be n-right orthonormal if the horizontal unfoldings of cores $n+1,\ldots,N$ are all row orthonormal, and a tensor

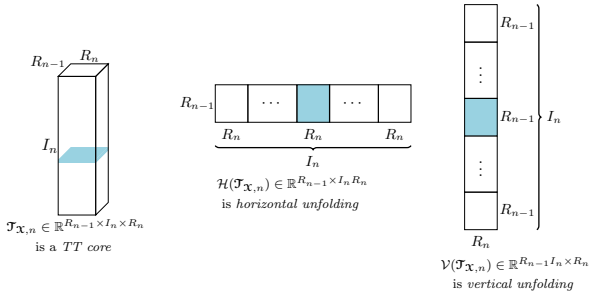


FIG. 2.2. *Horizontal and vertical unfoldings of a TT core.*

is n-left orthonormal if the vertical unfoldings of cores $1, \ldots, n-1$ are all column orthonormal.

These definitions correspond to the fact that the tensor that represents the contraction of these sets of TT cores inherits their orthonormality. For example, let $\mathcal{X}$ be a right-orthonormal TT tensor, then we can write $\mathbf{X}_{(1)} = \mathcal{T}_{\mathcal{X},1}\mathbf{Z}_{(1)}$, where $\mathcal{Z}$ is a $R_1 \times I_2 \times \cdots \times I_N$ tensor whose entries are given by

$$\mathcal{Z}(r_1, i_2, \ldots, i_N) = \mathcal{T}_{\mathcal{X},2}(r_1, i_2, :) \cdot \mathcal{T}_{\mathcal{X},3}(:, i_3, :) \cdots \mathcal{T}_{\mathcal{X},n}(:, i_n, :) \cdots \mathcal{T}_{\mathcal{X},N}(:, i_N).$$

The 1st modal unfolding of $\mathcal{Z}$ is row orthonormal, as shown below [41, Lemma 3.1]:

$$\begin{aligned}
\mathbf{Z}_{(1)}\mathbf{Z}_{(1)}^\top &= \sum_{i_2,\ldots,i_N} \mathcal{Z}(:, i_2, \ldots, i_N)\mathcal{Z}(:, i_2, \ldots, i_N)^\top \\
&= \sum_{i_2,\ldots,i_N} \underbrace{\mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N}(:, i_N)}_{\mathcal{Z}(:, i_2, \ldots, i_N)} \underbrace{\mathcal{T}_{\mathcal{X},N}(:, i_N)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top}_{\mathcal{Z}(:, i_2, \ldots, i_N)^\top} \\
&= \sum_{i_2,\ldots,i_{N-1}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :) \left(\sum_{i_N} \mathcal{T}_{\mathcal{X},N}(:, i_N)\mathcal{T}_{\mathcal{X},N}(:, i_N)^\top \right) \\
&\qquad \cdot \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \sum_{i_2,\ldots,i_{N-1}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :) \underbrace{\mathcal{H}(\mathcal{T}_{\mathcal{X},N})\mathcal{H}(\mathcal{T}_{\mathcal{X},N})^\top}_{I_{R_{N-1}}} \\
&\qquad \cdot \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \sum_{i_2,\ldots,i_{N-1}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :)\mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \sum_{i_2,\ldots,i_{N-2}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :) \Bigg(\sum_{i_{N-1}} \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :) \\
&\qquad \cdot \mathcal{T}_{\mathcal{X},N-1}(:, i_{N-1}, :)^\top \Bigg) \mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \sum_{i_2,\ldots,i_{N-2}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :) \underbrace{\mathcal{H}(\mathcal{T}_{\mathcal{X},N-1})\mathcal{H}(\mathcal{T}_{\mathcal{X},N-1})^\top}_{I_{R_{N-2}}} \\
&\qquad \cdot \mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \sum_{i_2,\ldots,i_{N-2}} \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :)\mathcal{T}_{\mathcal{X},N-2}(:, i_{N-2}, :)^\top \cdots \mathcal{T}_{\mathcal{X},2}(:, i_2, :)^\top \\
&= \cdots = I_{R_1}.
\end{aligned}$$

Similar arguments show that the 1st modal unfolding of the tensor representing the last $N-n$ cores of an n-right orthonormal TT tensor is row orthonormal and that the last modal unfolding of the tensor representing the first $n-1$ cores of an n-left orthonormal TT tensor is row orthonormal.

Given a TT tensor, we can orthonormalize it by exploiting the nonuniqueness of TT tensors expressed in 2.1. That is, we can right- or left-orthonormalize a TT core using a QR decomposition of one of its unfoldings and pass its triangular factor to its neighbor core without changing the represented tensor. By starting from one end and repeating this process on each core in order, we can obtain a left or right orthonormal TT tensor, as shown in Algorithm 2.1 (for right orthonormalization).

Algorithm 2.1 TT-right-orthonormalization

Require: A TT tensor $\mathcal{X}$
Ensure: A right orthonormal TT tensor $\mathcal{Y}$ equivalent to $\mathcal{X}$
1: **function** $\mathcal{Y} =$ RIGHT-ORTHONORMALIZATION($\mathcal{X}$)
2: Set $\mathcal{T}_{\mathcal{Y},N} = \mathcal{T}_{\mathcal{X},N}$
3: **for** $n = N$ down to 2 **do**
4: $[\mathcal{H}(\mathcal{T}_{\mathcal{Y},n})^\top, \mathbf{R}] = \mathrm{QR}(\mathcal{H}(\mathcal{T}_{\mathcal{Y},n})^\top)$ ▷ QR factorization
5: $\mathcal{V}(\mathcal{T}_{\mathcal{Y},n-1}) = \mathcal{V}(\mathcal{T}_{\mathcal{X},n-1})\mathbf{R}^\top$ ▷ $\mathcal{T}_{\mathcal{Y},n-1} = \mathcal{T}_{\mathcal{X},n-1} \times_3 \mathbf{R}^\top$
6: **end for**
7: **end function**

We note that the norm of a right- or left-orthonormal TT tensor can be cheaply computed, based on the idea that postmultiplication by a matrix with orthonormal rows or premultiplication by a matrix with orthonormal columns does not affect the Frobenius norm of a matrix. Thus, we have that $\|\mathcal{X}\| = \|\mathcal{T}_{\mathcal{X},1}\|_F$ provided that $\mathbf{Z}_{(1)}$ has orthonormal rows, and $\|\mathcal{X}\| = \|\mathcal{T}_{\mathcal{X},N}\|_F$ if $\mathcal{X}$ is left orthonormal.

2.4. TT rounding. Orthonormalization plays an essential role in compressing the TT format of a tensor (decreasing the TT ranks R_n) [41]. This compression is known as TT rounding and is given in Algorithm 2.2.

The intuition for rounding can be expressed in matrix notation as follows. Suppose we have a matrix represented by a product

$$\mathbf{A} = \mathbf{QBCZ}, \tag{2.2}$$

where $\mathbf{Q}$ and $\mathbf{Z}$ are column and row orthonormal, respectively. Then the truncated SVD of $\mathbf{A}$ can be readily expressed in terms of the truncated SVD of $\mathbf{BC}$. In our case, $\mathbf{B}$ is tall and skinny and $\mathbf{C}$ is short and wide, so the rank is bounded by their shared dimension. To truncate the rank, one can row-orthonormalize $\mathbf{C}$ and then perform a truncated SVD of $\mathbf{B}$ (or vice-versa). That is, if we compute $\mathbf{R}_C\mathbf{Q}_C = \mathbf{C}$ and $\mathbf{U}_B\mathbf{\Sigma}_B\mathbf{V}_B^\top = \mathbf{BR}_C$, then to round $\mathbf{A}$ we can replace $\mathbf{B}$ with $\hat{\mathbf{U}}_B$ and $\mathbf{C}$ with $\hat{\mathbf{\Sigma}}_B\hat{\mathbf{V}}_B^\top\mathbf{Q}_C$, where $\hat{\mathbf{U}}_B\hat{\mathbf{\Sigma}}_B\hat{\mathbf{V}}_B^\top$ is the SVD truncated to the desired tolerance.

In order to truncate a particular rank R_n by considering only the nth TT core using this idea, the TT format should be both n-left and n-right orthonormal. The unfolding of $\mathcal{X}$ that maps the first n tensor dimensions to rows can be expressed as a product of four matrices:

$$\mathbf{X}_{(1:n)} = (\mathbf{I}_{I_n} \otimes \mathbf{Q}_{(1:n-1)}) \cdot \mathcal{V}(\mathcal{T}_{\mathcal{X},n}) \cdot \mathcal{H}(\mathcal{T}_{\mathcal{X},n+1}) \cdot (\mathbf{I}_{I_{n+1}} \otimes \mathbf{Z}_{(1)}), \tag{2.3}$$

where $\mathbf{Q}$ is $I_1 \times \cdots \times I_{n-1} \times R_{n-1}$ with

$$\mathbf{Q}(i_1, \ldots, i_{n-1}, r_{n-1}) = \mathcal{T}_{\mathcal{X},1}(i_1, :) \cdot \mathcal{T}_{\mathcal{X},2}(:, i_2, :) \cdots \mathcal{T}_{\mathcal{X},n-1}(:, i_{n-1}, r_{n-1}),$$

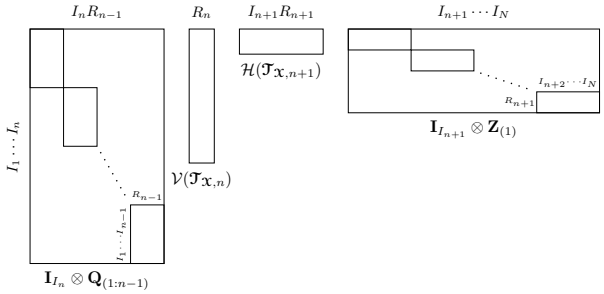


FIG. 2.3. *Visualization of identity* (2.3) *for* $\mathbf{X}_{(1:n)}$.

and $\mathfrak{Z}$ is $R_{n+1} \times I_{n+2} \times \cdots \times I_N$ with

$$\mathfrak{Z}(r_{n+1}, i_{n+2}, \ldots, i_N) = \mathcal{T}_{\mathcal{X},n+2}(r_{n+1}, i_{n+2}, :) \cdot \mathcal{T}_{\mathcal{X},n+3}(:, i_{n+3}, :) \cdots \mathcal{T}_{\mathcal{X},N}(:, i_N).$$

See Figure 2.3 for a visualization and section SM1 for a full derivation of (2.3). If $\mathcal{X}$ is n-left and n-right orthonormal, then $\mathbf{Q}_{(1:n-1)}$ and $\mathbf{Z}_{(1)}$ are column and row orthonormal (and so are their Kronecker products with an identity matrix), respectively, and $\mathcal{H}(\mathcal{T}_{\mathcal{X},n+1})$ is also row orthonormal.

In order to truncate R_n, we view (2.3) as an instance of (2.2) where $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ plays the role of $\mathbf{B}$ and $\mathcal{H}(\mathcal{T}_{\mathcal{X},n+1})$ plays the role of $\mathbf{C}$ (though $\mathcal{H}(\mathcal{T}_{\mathcal{X},n+1})$ is already orthonormalized). We compute the truncated SVD $\mathcal{V}(\mathcal{T}_{\mathcal{X},n}) \approx \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^\top$, replace $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ with $\hat{\mathbf{U}}$, and apply $\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^\top$ to $\mathcal{H}(\mathcal{T}_{\mathcal{X},n+1})$. In this way, R_n is truncated, $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ becomes column orthonormal, and because $\mathfrak{Q}$ and $\mathfrak{Z}$ are not modified, $\mathcal{X}$ becomes $(n+1)$-left and $(n+1)$-right orthonormal and ready for the truncation of R_{n+1}.

The rounding procedure consists of two sweeps along the modes. During the first, the tensor is left or right orthonormalized. On the second, sweeping in the opposite direction, the TT ranks are reduced sequentially via SVD truncation of the matricized cores. The rounding accuracy ε_0 can be defined a priori such that the rounded TT tensor is ε_0-close to the original TT tensor. We note that this method is quasi-optimal in finding the closest TT tensor with prescribed TT ranks to a given TT tensor [40].

Algorithm 2.2 TT-rounding

Require: A tensor $\mathcal{Y}$ in TT format, a threshold ε_0
Ensure: A tensor $\mathcal{X}$ in TT format with reduced ranks such that $\|\mathcal{X} - \mathcal{Y}\| \leq \varepsilon_0 \|\mathcal{Y}\|$

function $\mathcal{X}$ = ROUNDING($\mathcal{Y}, \varepsilon_0$)
  $\mathcal{X}$ = RIGHT-ORTHONORMALIZATION($\mathcal{Y}$)
  Compute $\|\mathcal{Y}\| = \|\mathcal{T}_{\mathcal{X},1}\|_F$ and the truncation threshold $\varepsilon = \frac{\|\mathcal{Y}\|}{\sqrt{N-1}}\varepsilon_0$
  for $n = 1$ to $N - 1$ **do**
    $[\mathcal{V}(\mathcal{T}_{\mathcal{X},n}), \mathbf{\Sigma}, \mathbf{V}] = \text{SVD}(\mathcal{V}(\mathcal{T}_{\mathcal{X},n}), \varepsilon)$ ▷ ε-truncated SVD factorization
    $\mathcal{H}(\mathcal{T}_{\mathcal{X},n+1}) = \mathbf{\Sigma}\mathbf{V}^\top \mathcal{H}(\mathcal{T}_{\mathcal{X},n+1})$ ▷ $\mathcal{T}_{\mathcal{X},n+1} = \mathcal{T}_{\mathcal{X},n+1} \times_1 (\mathbf{\Sigma}\mathbf{V}^\top)$
  end for
end function

2.5. Parallel cost model. To analyze our parallel algorithms, we use the MPI-based model that tracks floating point operations (flops) as well as the amount of data and number of messages communicated along the critical path [6, 16, 51]. In

this model, communication is performed via point-to-point messages, and the time is estimated as the sum of time spent in computation and communication along the critical path. In this way, processors can perform independent computations simultaneously and disjoint pairs of processors can communicate messages simultaneously. Each flop is assumed to cost γ units of time, and message of n words is assumed to cost $\alpha + \beta n$ units, where α is referred to as the per-message latency cost and β is the per-word bandwidth cost. Accumulating costs along the critical path ensures that computation and communication that depend on one another occur in sequence. The parallel time cost is thus estimated as $\gamma \cdot \#$ flops $+ \beta \cdot \#$ words $+ \alpha \cdot \#$ messages. Overlapping computation and communication is a useful optimization (and our implementation does so when possible), but the model ignores this possibility as it affects the overall running time by at most a constant. Algorithms for collective communications among groups of processors, such as AllReduce, have been optimized for this model (and within MPI implementations), and we use the previously established costs of collectives [16, 51] in our analysis.

3. Parallel algorithms for TT. In this section we detail the parallel algorithms for manipulating TT tensors that are distributed over multiple processors' memories. We describe our proposed data distribution of the core tensors in subsection 3.1, which is designed for efficient orthonormalization and truncation of TT tensors. In subsection 3.2 we show how to perform basic operations on TT tensors in this distribution such as addition, elementwise multiplication, and applying certain linear operators. Our proposed parallel orthonormalization and truncation routines are presented in subsections 3.4 and 3.5, respectively. Both of those routines rely on an existing communication-efficient parallel QR decomposition algorithm called Tall-Skinny QR (TSQR) [18], which is given for completeness in subsection 3.3. A summary of the costs of the parallel algorithms is presented in Table 3.1.

Table 3.1

Summary of computation and communication costs of parallel TT operations using P processors, assuming inputs are N-way tensors with identical dimensions $I_n = I$ and ranks $R_n = R$. The computation cost of rounding assumes the original ranks are reduced in half; the constant can range from 3 to 13 depending on the reduced ranks.

TT algorithm	Computation	Comm. Data	Comm. Msgs
Summation	—	—	—
Hadamard	$\frac{NIR^4}{P}$	—	—
Inner Product	$4\frac{NIR^3}{P}$	$O(NR^2)$	$O(N \log P)$
Norm	$2\frac{NIR^3}{P}$	$O(NR^2)$	$O(N \log P)$
Orthonormalization	$5\frac{NIR^3}{P} + O(NR^3 \log P)$	$O(NR^2 \log P)$	$O(N \log P)$
Rounding	$7\frac{NIR^3}{P} + O(NR^3 \log P)$	$O(NR^2 \log P)$	$O(N \log P)$

3.1. Data distribution and layout. We are interested in the parallelization of TT operations with a large number of modes and where one or multiple mode sizes are very large compared to the TT ranks. This type of configuration arises in many applications such as parameter dependent PDEs [34], stochastic PDEs [32], and molecular simulations [46]. In case there exist TT cores with relatively small mode sizes, those can be stored redundantly on each processor. We note that our implementation can deal with both cases.

Algorithms for orthonormalization and rounding of TT tensors are sequential with respect to the mode; often computation can occur on only one mode at a time. In order to utilize all processors and maintain load balancing in a parallel environment,

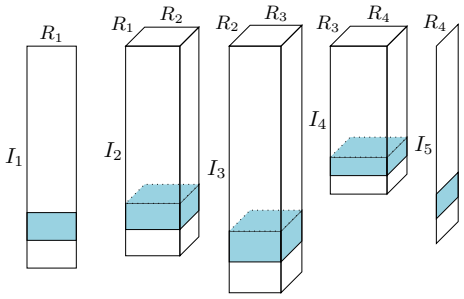


FIG. 3.1. *One-dimensional (1D) distribution of a TT tensor across P processors with data owned by a particular processor highlighted in blue.*

we choose to distribute each TT core over all processors, so that each processor owns a subtensor of each TT core. To ensure the computations on each core can be done in a communication-efficient way, we choose a 1D distribution for each core, where the mode corresponding to the original tensor is divided across processors. This corresponds to a Cartesian distribution of each $R_{n-1} \times I_n \times R_n$ core over a $1 \times P \times 1$ processor grid, or equivalently, a block row distribution of $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ or a block column distribution of $\mathcal{H}(\mathcal{T}_{\mathcal{X},n})$ for $n = 1, \ldots, N$; see Figure 3.1. In this manner, each processor owns N local subtensors with dimensions $\{R_{n-1} \times (I_n/P) \times R_n\}$. The notation $\mathcal{T}^{(p)}_{\mathcal{X},n}$ denotes the local subtensor of the nth core owned by processor p.

This distribution allows performing basic operations, such as addition and elementwise multiplication, on the TT representation locally; see subsection 3.2. Furthermore, the bottleneck computations within orthonormalization and rounding are orthonormalization of vertical and horizontal unfoldings of TT cores. For communication optimality of these operations, the TSQR algorithm (see subsection 3.3) requires that both of these unfoldings are in 1D matrix distribution, which in turn requires that the TT core be distributed over a $1 \times P \times 1$ processor grid. The distribution of a TT core in this way can also be seen as a generalization of the distribution of a vector in parallel iterative linear solvers [1, 26]. Indeed, if $\mathbf{A}$ is an $I_n \times I_n$ sparse matrix distributed across processors as block row panels, the computation of $\mathbf{A}\mathcal{T}_{\mathcal{X},n}(k, :, l)$ can be done by using standard parallel sparse-matrix-vector multiplication routines. We note that a drawback of this distribution is that the available parallelism in each TT core computation is limited to the size of the tensor dimension. If the TT ranks are much larger than the tensor dimension, then alternative distributions, redistributions, and parallelizations should be considered.

Tensor entries are linearized in memory. Each local core tensor $\mathcal{T}^{(p)}_{\mathcal{X},n}$ is $R_{n-1} \times (I_n/P) \times R_n$, and we store it in the "vec-oriented" or "natural descending" order [8, 44] in memory. For 3-way tensors, this means that mode-1 fibers (of length R_{n-1}) are contiguous in memory, as this corresponds to the mode-1 modal unfolding. Additionally, the mode-3 slices (of size $R_{n-1} \times (I_n/P)$) are also contiguous in memory and internally linearized in column-major order, as this corresponds to the more general (1:2) unfolding [42, 44]. In particular, these facts imply that both the vertical and horizontal unfoldings are column major in memory.

BLAS and LAPACK routines require either row- or column-major ordering (unit stride for one dimension and constant stride for the other), but this property of the vertical and horizontal unfoldings means that we can operate on them without any physical permutation of the tensor data. For example, we can perform operations

such as QR factorization of $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})$ and $\mathcal{V}(\mathcal{T}_{\mathcal{X},n})\mathbf{R}$, where $\mathbf{R} \in \mathbb{R}^{R_n \times R_n}$, with a single LAPACK or BLAS call.

This choice of ordering comes at the expense of less convenient access to the mode-2 modal unfolding (of dimension $(I_n/P) \times R_{n-1}R_n$), which is neither row nor column major in memory. This unfolding can be visualized in memory as a concatenation of R_n contiguous submatrices, each of dimension $(I_n/P) \times R_{n-1}$ and each stored in row-major order [8]. In order to perform the mode-2 multiplication (tensor times matrix operation), as is necessary in the application of a spatial operator on the core, we must make a sequence of calls to the matrix-matrix multiplication BLAS subroutine. That is, we make R_n calls for multiplications of the same $I_n \times I_n$ matrix with different $I_n \times R_{n-1}$ matrices.

3.2. Basic operations.

3.2.1. Summation. To sum two tensors $\mathcal{X}$ and $\mathcal{Y}$, we can write [41]

$$\begin{aligned}\mathcal{Z}(i_1,\dots,i_N) &= \mathcal{X}(i_1,\dots,i_N) + \mathcal{Y}(i_1,\dots,i_N)\\ &= \mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N) + \mathcal{T}_{\mathcal{Y},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{Y},N}(:,i_N)\\ &= \begin{pmatrix}\mathcal{T}_{\mathcal{X},1}(i_1,:) & \mathcal{T}_{\mathcal{Y},1}(i_1,:)\end{pmatrix}\begin{pmatrix}\mathcal{T}_{\mathcal{X},2}(:,i_2,:) & \\ & \mathcal{T}_{\mathcal{Y},2}(:,i_2,:)\end{pmatrix}\\ &\quad\cdots\begin{pmatrix}\mathcal{T}_{\mathcal{X},N-1}(:,i_{N-1},:) & \\ & \mathcal{T}_{\mathcal{Y},N-1}(:,i_{N-1},:)\end{pmatrix}\begin{pmatrix}\mathcal{T}_{\mathcal{X},N}(:,i_N)\\ \mathcal{T}_{\mathcal{Y},N}(:,i_N)\end{pmatrix}.\end{aligned}$$

Thus, the TT representation of $\mathcal{Z} = \mathcal{X} + \mathcal{Y}$ is given by the following slicewise formula:

$$\mathcal{T}_{\mathcal{Z},n}(:,i_n,:) = \begin{pmatrix}\mathcal{T}_{\mathcal{X},n}(:,i_n,:) & \\ & \mathcal{T}_{\mathcal{Y},n}(:,i_n,:)\end{pmatrix}$$

for $2 \le n \le N-1$, and $1 \le i_n \le I_n$. We also have $\mathcal{T}_{\mathcal{Z},1} = \begin{pmatrix}\mathcal{T}_{\mathcal{X},1} & \mathcal{T}_{\mathcal{Y},1}\end{pmatrix}$ and

$$\mathcal{T}_{\mathcal{Z},N} = \begin{pmatrix}\mathcal{T}_{\mathcal{X},N}\\ \mathcal{T}_{\mathcal{Y},N}\end{pmatrix}.$$

Note that the TT ranks of this representation of $\mathcal{Z}$ are the sums of the TT ranks of $\mathcal{X}$ and $\mathcal{Y}$.

Given the 1D data distribution of each core described in subsection 3.1, the summation operation can be performed locally with no interprocessor communication. That is, because $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ have identical dimensions, they will have identical distributions, and each slice of a core tensor of $\mathcal{Z}$ will be owned by the processor that owns the corresponding slices of cores of $\mathcal{X}$ and $\mathcal{Y}$.

3.2.2. Hadamard product. To compute the Hadamard (elementwise) product of two tensors $\mathcal{X}$ and $\mathcal{Y}$, we can write [41]

$$\begin{aligned}\mathcal{Z}(i_1,\dots,i_N) &= \mathcal{X}(i_1,\dots,i_N)\cdot\mathcal{Y}(i_1,\dots,i_N)\\ &= (\mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N))\cdot(\mathcal{T}_{\mathcal{Y},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{Y},N}(:,i_N))\\ &= (\mathcal{T}_{\mathcal{X},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{X},N}(:,i_N))\otimes(\mathcal{T}_{\mathcal{Y},1}(i_1,:)\cdots\mathcal{T}_{\mathcal{Y},N}(:,i_N))\\ &= (\mathcal{T}_{\mathcal{X},1}(i_1,:)\otimes\mathcal{T}_{\mathcal{Y},1}(i_1,:))\cdots(\mathcal{T}_{\mathcal{X},N}(:,i_N)\otimes\mathcal{T}_{\mathcal{Y},N}(:,i_N)).\end{aligned}$$

Thus, the TT representation of $\mathcal{Z} = \mathcal{X} * \mathcal{Y}$ is given by the following slicewise formula: $\mathcal{T}_{\mathcal{Z},n}(:,i_n,:) = \mathcal{T}_{\mathcal{X},n}(:,i_n,:)\otimes\mathcal{T}_{\mathcal{Y},n}(:,i_n,:)$ for $1 \le n \le N$ and $1 \le i_n \le I_n$. Here, the TT ranks of the representation of $\mathcal{Z}$ are the products of the TT ranks of $\mathcal{X}$ and $\mathcal{Y}$.

Again, given the 1D data distribution of each core and the fact that each core is computed slicewise, the Hadamard product can be performed locally with no interprocessor communication. We note that because of the extra expense of the Hadamard product (due to computing explicit Kronecker products of slices), it is likely advantageous to maintain Hadamard products in implicit form for later operations such as rounding. While we do not pursue this approach further in this work, the combination of Hadamard products and recompression has been shown to be effective for Tucker tensors, but it requires randomization in the truncation operations [33].

3.2.3. Inner product. To compute the inner product of two tensors $\mathcal{X}$ and $\mathcal{Y}$, using similar identities as for the Hadamard product, we can write [41]

$$\begin{aligned}\langle \mathcal{X}, \mathcal{Y} \rangle &= \sum_{i_1,\ldots,i_N} \mathcal{X}(i_1,\ldots,i_N) \cdot \mathcal{Y}(i_1,\ldots,i_N) \\ &= \sum_{i_1,\ldots,i_N} (\mathcal{T}_{\mathcal{X},1}(i_1,:) \otimes \mathcal{T}_{\mathcal{Y},1}(i_1,:)) \cdots (\mathcal{T}_{\mathcal{X},N}(:,i_N) \otimes \mathcal{T}_{\mathcal{Y},N}(:,i_N)) \\ &= \sum_{i_1} (\mathcal{T}_{\mathcal{X},1}(i_1,:) \otimes \mathcal{T}_{\mathcal{Y},1}(i_1,:)) \sum_{i_2} (\mathcal{T}_{\mathcal{X},2}(:,i_2,:) \otimes \mathcal{T}_{\mathcal{Y},2}(:,i_2,:)) \\ &\quad \cdots \sum_{i_N} (\mathcal{T}_{\mathcal{X},N}(:,i_N) \otimes \mathcal{T}_{\mathcal{Y},N}(:,i_N)) .\end{aligned}$$

This expression can be evaluated efficiently by a sequence of structured matrix-vector products that avoid forming Kronecker products of matrices, and these matrix-vector products are cast as matrix-matrix multiplications.

To see how, we assume that the TT ranks of $\mathcal{X}$ and $\mathcal{Y}$ are $\{R_n^{\mathcal{X}}\}$ and $\{R_n^{\mathcal{Y}}\}$, respectively. First, we explicitly construct the row vector

$$\mathbf{w}_1 = \sum_{i_1} \mathcal{T}_{\mathcal{X},1}(i_1,:) \otimes \mathcal{T}_{\mathcal{Y},1}(i_1,:),$$

which has dimension $R_1^{\mathcal{X}} \cdot R_1^{\mathcal{Y}}$. Note that $\mathbf{w}_1$ is the vectorization of the matrix $\mathcal{V}(\mathcal{T}_{\mathcal{Y},1})^\top \mathcal{V}(\mathcal{T}_{\mathcal{X},1})$. Then we distribute $\mathbf{w}_1$ to all terms within the next summation to compute $\mathbf{w}_2$ using

$$\mathbf{w}_2 = \sum_{i_2} \mathbf{w}_1 \left(\mathcal{T}_{\mathcal{X},2}(:,i_2,:) \otimes \mathcal{T}_{\mathcal{Y},2}(:,i_2,:)\right),$$

with each term in the summation evaluated via $\mathtt{vec}\left(\mathcal{T}_{\mathcal{Y},2}(:,i_2,:)^\top \mathbf{W}_1 \mathcal{T}_{\mathcal{X},2}(:,i_2,:)\right)$, where $\mathbf{W}_1$ is a reshaping of the vector $\mathbf{w}_1$ into a $R_1^{\mathcal{Y}} \times R_1^{\mathcal{X}}$ matrix and $\mathtt{vec}$ is a row-wise vectorization operator. We note that $\mathcal{T}_{\mathcal{X},2}(:,i_2,:)$ is $R_1^{\mathcal{X}} \times R_2^{\mathcal{X}}$, and $\mathcal{T}_{\mathcal{Y},2}(:,i_2,:)$ is $R_1^{\mathcal{Y}} \times R_2^{\mathcal{Y}}$, and $\mathbf{w}_2$ therefore has dimension $R_2^{\mathcal{X}} \cdot R_2^{\mathcal{Y}}$. This process is repeated with

$$\mathbf{W}_n = \sum_{i_n} \mathcal{T}_{\mathcal{Y},n}(:,i_n,:)^\top \mathbf{W}_{n-1} \mathcal{T}_{\mathcal{X},n}(:,i_n,:), \tag{3.1}$$

until the last core, when we compute the inner product as $\langle \mathcal{X}, \mathcal{Y} \rangle = \sum_{i_N} \mathcal{T}_{\mathcal{Y},N}(:,i_N)^\top \mathbf{W}_{N-1} \mathcal{T}_{\mathcal{X},N}(:,i_N)$, where $\mathbf{W}_{N-1}$ is an $R_{N-1}^{\mathcal{Y}} \times R_{N-1}^{\mathcal{X}}$ matrix.

If all the tensor dimensions are the same and all TT ranks are the same, i.e., $I = I_1 = \cdots = I_N$ and $R = R_1^{\mathcal{X}} = R_1^{\mathcal{Y}} = \cdots = R_{N-1}^{\mathcal{X}} = R_{N-1}^{\mathcal{Y}}$, the computational complexity is approximately $4NIR^3$.

Evaluating (3.1) directly can exploit the efficiency of dense matrix multiplication, but it requires many calls to the BLAS subroutine. With some extra temporary memory, we can reduce the number of BLAS calls to 2, performing the same overall number of flops. Let $\mathcal{Z}$ be defined such that $\mathcal{H}(\mathcal{T}_{\mathcal{Z},n}) = \mathbf{W}_{n-1}\mathcal{H}(\mathcal{T}_{\mathcal{X},n})$, or the mode-1 multiplication between the core and the matrix, for $n = 1, \ldots, N$ (with $\mathbf{W}_0 = 1$). Then, we have $\mathbf{W}_n$ as a contraction of modes 1 and 2 between cores of $\mathcal{Y}$ and $\mathcal{Z}$, or

$$\mathbf{W}_n = \mathcal{V}(\mathcal{T}_{\mathcal{Y},n})^\top \mathcal{V}(\mathcal{T}_{\mathcal{Z},n}) \quad \text{for } \; n = 1, \ldots, N.$$

Each of these two multiplications requires a single BLAS call because horizontal and vertical unfoldings are column major in memory. We note the final contraction in mode N is a dot product instead of a matrix multiplication.

When the input TT tensors are distributed across processors as described in subsection 3.1, we can compute the inner product using this technique. Each term in the summation of (3.1), which involves corresponding slices of the input tensors, is evaluated by a single processor as long as the matrix $\mathbf{W}_n$ is available on each processor. Thus, the computation can be load balanced across processors as long as the distribution is load balanced, and each processor can apply the optimization to reduce BLAS calls independently. We perform an AllReduce collective operation to compute the summation for each mode. With constant tensor dimensions and TT ranks, the computational cost is approximately $4NIR^3/P$ and the communication cost is $\beta \cdot O(NR^2) + \alpha \cdot O(N \log P)$.

3.2.4. Norms. To compute the norm of a tensor in TT format, we consider two approaches. The first approach is to use the inner product algorithm described in subsection 3.2.3 and the identity $\|\mathcal{X}\|^2 = \langle \mathcal{X}, \mathcal{X} \rangle$. We note that in this case, the matrices $\{\mathbf{W}_n\}$ are symmetric and positive semidefinite (SPSD) (see (3.1)), and the structured matrix-vector products can exploit this property to save roughly half the computation. Since $\mathbf{W}_n$ is SPSD, it admits a triangular factorization given by pivoted Cholesky (or LDL): $\mathbf{W}_n = \mathbf{P}_n \mathbf{L}_n \mathbf{L}_n^\top \mathbf{P}_n^\top$. Thus, the matrix $\mathbf{W}_n$ is computed as $\mathbf{W}_n = \mathcal{V}(\mathcal{T}_{\mathcal{Z},n})^\top \mathcal{V}(\mathcal{T}_{\mathcal{Z},n})$, where $\mathcal{H}(\mathcal{T}_{\mathcal{Z},n}) = \mathbf{L}_{n-1}^\top (\mathbf{P}_{n-1}^\top \mathcal{H}(\mathcal{T}_{\mathcal{X},n}))$. The triangular multiplication to compute the nth core of $\mathcal{Z}$ and the symmetric multiplication to compute $\mathbf{W}_n$ each require half the flops of a normal matrix multiplication, so the overall computational complexity of this approach is $2NIR^3$. It is parallelized similarly to the general inner product.

The second approach is to first right- or left-orthonormalize the tensor using Algorithm 2.1, and then the norm of the tensor is given by $\|\mathcal{T}_{\mathcal{X},1}\|_F$ or $\|\mathcal{T}_{\mathcal{X},N}\|_F$ as shown in subsection 2.3. This approach can be more accurate than the first one when computing small norms, as the first approach can suffer from cancellation error. When the TT tensor is distributed, the orthonormalization procedure is more complicated than computing inner products; we describe the parallel algorithm in subsection 3.4.

3.2.5. Matrix-vector multiplication. In order to build Krylov-like iterative methods to solve linear systems with solutions in TT-format, we must also be able to apply a matrix operator to a vector in TT-format. We will consider a restricted set of matrix operators: sums of Kronecker products of sparse matrices [12, 32, 34, 53].

Each term in the sum can be seen as a generalization of a rank-one tensor to the operator case. We use the notation

$$\mathbf{A} = \mathbf{A}_1 \otimes \cdots \otimes \mathbf{A}_N$$

to denote a single Kronecker product of matrices, where the dimensions of $\mathbf{A}_n$ are $I_n \times I_n$, conforming to the dimensions of $\mathcal{X}$ in TT-format. In this case, we can compute

the matrix-vector multiplication $\text{vec}(\mathcal{Y}) = \mathbf{A} \cdot \text{vec}(\mathcal{X})$, where

$$\begin{aligned}
\mathcal{Y}(i_1, \ldots, i_N) &= \sum_{j_1,\ldots,j_N} \mathbf{A}_1(i_1, j_1) \cdots \mathbf{A}_N(i_N, j_N) \cdot \mathcal{X}(j_1, \ldots, j_N) \\
&= \sum_{j_1,\ldots,j_N} \mathbf{A}_1(i_1, j_1) \cdots \mathbf{A}_N(i_N, j_N) \cdot \mathcal{T}_{\mathcal{X},1}(j_1, :) \cdots \mathcal{T}_{\mathcal{X},N}(:, j_N) \\
&= \sum_{j_1} \mathbf{A}_1(i_1, j_1) \mathcal{T}_{\mathcal{X},1}(j_1, :) \cdots \sum_{j_N} \mathbf{A}_N(i_N, j_N) \mathcal{T}_{\mathcal{X},N}(:, j_N) \\
&= \mathcal{T}_{\mathcal{Y},1}(i_1, :) \cdots \mathcal{T}_{\mathcal{Y},N}(:, i_N)
\end{aligned}$$

with $\mathcal{T}_{\mathcal{Y},1} = \mathbf{A}_1 \mathcal{T}_{\mathcal{X},1}$, $\mathcal{T}_{\mathcal{Y},n} = \mathcal{T}_{\mathcal{X},n} \times_2 \mathbf{A}_n$ for $1 < n < N$, and $\mathcal{T}_{\mathcal{Y},N} = \mathcal{T}_{\mathcal{X},N} \mathbf{A}_N^\top$. Here the notation $\times_2$ refers to the mode-2 tensor-matrix product, defined so that

$$\mathcal{T}_{\mathcal{Y},n}(r_{n-1}, :, r_n) = \mathbf{A}_n \mathcal{T}_{\mathcal{X},n}(r_{n-1}, :, r_n)$$

for $1 < n < N$, $1 \le r_{n-1} \le R_{n-1}$, and $1 \le r_n \le R_n$.

Thus, applying a Kronecker product of matrices to a vector in TT-format maintains the TT-format with the same ranks, and operations on cores can be performed independently. In order to apply an operator that is a sum of multiple Kronecker products of matrices, we can apply each term separately and use the summation procedure described in subsection 3.2.1 along with TT-rounding to control rank growth. We note that it is possible to apply more general forms of tensorized operators to vectors in TT-format [41], but we do not consider them here.

When the vector in TT-format is distributed as described in subsection 3.1, we must perform the mode-2 tensor-matrix product using a parallel algorithm. We can view the mode-2 tensor-matrix product as applying the matrix to the mode-2 unfolding of the tensor core $\mathcal{T}_{\mathcal{X},n}$ (often denoted with subscript (2) [30]), which has dimensions $I_n \times R_{n-1}R_n$. We observe that the parallel distribution of the mode-2 unfolding of $\mathcal{T}_{\mathcal{X},n}$ is 1D row-distributed: each processor owns a subset of the rows of the matrix (corresponding to slices of the core tensor). Thus, the application of $\mathbf{A}_n$ to this unfolding has the same algorithmic structure as the sparse-matrix-times-multiple-vectors operation (SpMM) where all vectors have the same parallel distribution. Assuming the matrix $\mathbf{A}_n$ is sparse and also row-distributed, as is common in libraries such as PETSc [5] and Trilinos [24], the parallel algorithm involves communication of input tensor core slices among processors, where the communication pattern is determined by $\mathbf{A}_n$ and its distribution. We do not explore experimental results for such matrix-vector multiplications in this paper, as the performance depends heavily on the application and sparsity structure of the operator matrices.

3.3. TSQR. As is evident in Algorithms 2.1 and 2.2, the QR factorization of tall-skinny matrices is a key subroutine in TT rounding. To compute the QR factorizations within the TT orthonormalization and TT rounding procedures in parallel, we use the Tall-Skinny QR algorithm [18], which is designed (and communication efficient) for matrices with many more rows than columns. For completeness, we present the TSQR subroutine as Algorithm 3.1, which corresponds to [7, Alg. 7], and the TSQR-Apply-Q subroutine as Algorithm 3.2. While TSQR is strictly a matrix algorithm, it is fundamental to the TT algorithms and analysis of subsections 3.4 and 3.5, so we present it separately in this subsection. The subroutines assume a power-of-two number of processors to simplify the pseudocode; see section SM2 for the generalizations to any number of processors.

For a tall-skinny matrix that is 1D row-distributed over processors (as is the case for the vertical unfolding and the transpose of the horizontal unfolding), the parallel Householder QR algorithm requires synchronizations for each column of the matrix (to compute and apply each Householder vector). Furthermore, the local computation of Householder QR is nearly always memory-bandwidth bound in the form of BLAS-2 subroutines (matrix-vector operations). The idea of the TSQR algorithm is that the entire factorization can be computed using a single reduction across processors, and each local computation becomes a smaller QR factorization. That is, while parallel Householder QR has latency cost of $O(b)$ for a matrix with b columns, TSQR has latency cost $O(\log P)$ (see subsection 3.3.1). The superior performance of TSQR over Householder QR has been demonstrated on both distributed-memory and shared-memory platforms [2, 7, 17, 38].

The price of TSQR is that the implicit representation of the orthonormal factor is more complicated than a single set of Householder vectors, and that the representation depends on the structure of the reduction tree. We can maintain and apply the orthonormal factor in this implicit form as long as the parallel algorithm for applying it uses a consistent tree structure. We note that we employ the "butterfly" variant of TSQR, which corresponds to an AllReduce-like collective operation such that at the end of the algorithm the triangular factor $\mathbf{R}$ is owned by all processors redundantly. At each of the $\log P$ steps, each processor determines a different partner processor with which to exchange data. Another variant uses a binomial tree, corresponding to a reduce-like collective with the triangular factor owned by a single processor. In the context of TT, the key advantage of the butterfly over the binomial variant is the reduction in communication when the implicit orthogonal factor is applied to another matrix, as we describe in subsection 3.3.2. We compare performance of these two variants in subsection 4.3.1.

3.3.1. Factorization. TSQR (Algorithm 3.1) has two phases: local submatrix orthonormalization (line 3) and parallel reduction of remaining triangular factors (line 4 through line 12). The cost of the TSQR is as follows:

$$\gamma \cdot \left(2\frac{mb^2}{P} + O(b^3 \log P)\right) + \beta \cdot O(b^2 \log P) + \alpha \cdot O(\log P), \tag{3.2}$$

where m is the number of rows and b is the number of columns [18]. The leading order flop cost is the QR of the local $(m/P) \times b$ submatrix (line 3), the leaf of the TSQR tree. The communication costs come from the TSQR tree, which has height $O(\log P)$.

3.3.2. Applying and forming Q. The structure of the TSQR-Apply-Q algorithm (Algorithm 3.2) matches that of TSQR, but in reverse order (because the TSQR algorithm corresponds to applying $Q^\top$). Thus, the root of the tree is applied first and the leaves last. However, by using a butterfly tree the communication cost of the TSQR-Apply-Q algorithm (Algorithm 3.2) is 0 if the number of processors is a power of 2 and $\beta \cdot bc + \alpha$ otherwise (the cost of one message; see section SM2). The cost of TSQR-Apply-Q is then

$$\gamma \cdot \left(4\frac{mbc}{P} + O(b^2 c \log P)\right) + \beta \cdot bc + \alpha, \tag{3.3}$$

where the additional parameter c is the number of columns of C. The leading order flop cost is the application of the local Q matrix at the leaf of the TSQR tree (line 12).

Algorithm 3.1 Parallel Butterfly TSQR

Require: $\mathbf{A}$ is an $m \times b$ matrix 1D-distributed so that proc p owns row block $\mathbf{A}^{(p)}$
Require: Number of procs is power of two; see Algorithm SM2.1 for general case
Ensure: $\mathbf{A} = \mathbf{QR}$ with $\mathbf{R}$ owned by all procs and $\mathbf{Q}$ represented by $\{\mathbf{Y}_\ell^{(p)}\}$ with redundancy $\mathbf{Y}_\ell^{(p)} = \mathbf{Y}_\ell^{(q)}$ for $p \equiv q \mod 2^\ell$ and $\ell < \log P$

1: **function** $[\{\mathbf{Y}_\ell^{(p)}\}, \mathbf{R}] = \text{PAR-TSQR}(\mathbf{A}^{(p)})$
2: $p = \text{MYPROCID}()$
3: $[\mathbf{Y}_{\log P}^{(p)}, \bar{\mathbf{R}}_{\log P}^{(p)}] = \text{Local-QR}(\mathbf{A}^{(p)})$ ▷ Leaf node QR
4: **for** $\ell = \log P - 1$ down to 0 **do**
5: $j = 2^{\ell+1}\lfloor \frac{p}{2^{\ell+1}} \rfloor + \left((p + 2^\ell) \mod 2^{\ell+1}\right)$ ▷ Determine partner
6: Send $\bar{\mathbf{R}}_{\ell+1}^{(p)}$ to and receive $\bar{\mathbf{R}}_{\ell+1}^{(j)}$ from proc j ▷ Communication
7: **if** $p < j$ **then**
8: $[\mathbf{Y}_\ell^{(p)}, \bar{\mathbf{R}}_\ell^{(p)}] = \text{Local-QR}\left(\begin{bmatrix} \bar{\mathbf{R}}_{\ell+1}^{(p)} \\ \bar{\mathbf{R}}_{\ell+1}^{(j)} \end{bmatrix}\right)$ ▷ Tree node QR
9: **else**
10: $[\mathbf{Y}_\ell^{(p)}, \bar{\mathbf{R}}_\ell^{(p)}] = \text{Local-QR}\left(\begin{bmatrix} \bar{\mathbf{R}}_{\ell+1}^{(j)} \\ \bar{\mathbf{R}}_{\ell+1}^{(p)} \end{bmatrix}\right)$ ▷ Partner tree node QR
11: **end if**
12: **end for**
13: $\mathbf{R} = \bar{\mathbf{R}}_0^{(p)}$
14: **end function**

Using a binomial tree TSQR algorithm requires more communication in the application phase (see [7, Algorithm 8], for example). We also note that if the input matrix $\mathbf{C}$ is upper triangular, then the leading constant can be reduced from 4 to 2 by exploiting the sparsity structure in this local application (and within the tree because all $\bar{\mathbf{B}}_\ell^{(p)}$ matrices are upper triangular in this case, throughout the algorithm), which matches the computation cost of the factorization. In particular, when we form $\mathbf{Q}$, we use this algorithm with $\mathbf{C}$ as the identity matrix, which is upper triangular.

3.4. TT orthonormalization. Given the parallel TSQR algorithm of subsection 3.3, we now present a parallel algorithm for TT Orthonormalization. Algorithm 3.3 shows right orthonormalization and is a parallelization of Algorithm 2.1. The approach for left orthonormalization is analogous. The algorithm is performed via a sequential sweep over the cores, where at each iteration, an LQ factorization row-orthonormalizes the horizontal unfolding of a core and the triangular factor is applied to its left neighbor core. The 1D parallel distribution of each core implies that the transpose of the horizontal unfolding is 1D row-distributed, fitting the requirements of the TSQR algorithm. Note that we perform a QR factorization of the transpose of the horizontal unfolding, which corresponds to an LQ factorization of the unfolding itself.

Figure 3.2 depicts the operations within a single iteration of the sweep. At iteration n, TSQR is applied to the nth core in line 3 (Figure 3.2(c)) and then the orthonormal factor is formed explicitly in line 4 (Figure 3.2(b)). The notation $\{\mathbf{Y}_{\ell,n}^{(p)}\}$ signifies the set of triangular matrices owned by processor p in the implicit representation of the QR factorization of the nth core, where ℓ refers to the level of the tree and indexes the set. In the case P is a power of 2, each processor owns $\log P$ matrices in

Algorithm 3.2 Parallel Application of Implicit Q from Butterfly TSQR

Require: $\{\mathbf{Y}_\ell^{(p)}\}$ represents orthonormal matrix $\mathbf{Q}$ computed by Algorithm 3.1
Require: $\mathbf{C}$ is $b \times c$ and redundantly owned by all processors
Require: Number of procs is power of two; see Algorithm SM2.2 for general case
Ensure: $\mathbf{B} = \mathbf{Q}\begin{bmatrix}\mathbf{C}\\ \mathbf{0}\end{bmatrix}$ is $m \times c$ and 1D-distributed so that proc p owns row block $\mathbf{B}^{(p)}$

1: **function** $\mathbf{B}$ = Par-TSQR-Apply-Q($\{\mathbf{Y}_\ell^{(p)}\}, \mathbf{C}$)
2: p = MyProcID()
3: $\bar{\mathbf{B}}_0^{(p)} = \mathbf{C}$
4: **for** $\ell = 0$ to $\log P - 1$ **do**
5: $j = 2^{\ell+1}\lfloor \frac{p}{2^{\ell+1}} \rfloor + \left((p + 2^\ell) \mod 2^{\ell+1}\right)$ ▷ Determine partner
6: **if** $p < j$ **then**
7: $\begin{bmatrix}\bar{\mathbf{B}}_{\ell+1}^{(p)}\\ \bar{\mathbf{B}}_{\ell+1}^{(j)}\end{bmatrix} = \text{Loc-Apply-Q}\left(\begin{bmatrix}\mathbf{I}_b\\ \mathbf{Y}_\ell^{(p)}\end{bmatrix}, \begin{bmatrix}\bar{\mathbf{B}}_\ell^{(p)}\\ \mathbf{0}\end{bmatrix}\right)$ ▷ Tree node apply
8: **else**
9: $\begin{bmatrix}\bar{\mathbf{B}}_{\ell+1}^{(j)}\\ \bar{\mathbf{B}}_{\ell+1}^{(p)}\end{bmatrix} = \text{Loc-Apply-Q}\left(\begin{bmatrix}\mathbf{I}_b\\ \mathbf{Y}_\ell^{(p)}\end{bmatrix}, \begin{bmatrix}\bar{\mathbf{B}}_\ell^{(p)}\\ \mathbf{0}\end{bmatrix}\right)$ ▷ Part. tree node apply
10: **end if**
11: **end for**
12: $\mathbf{B}^{(p)} = \text{Loc-Apply-Q}\left(\mathbf{Y}_{\log P}^{(p)}, \begin{bmatrix}\bar{\mathbf{B}}_{\log P}^{(p)}\\ \mathbf{0}\end{bmatrix}\right)$ ▷ Leaf node apply
13: **end function**

its set. Because the TSQR subroutine ends with all processors owning the triangular factor $\mathbf{R}_n$, each processor can apply it to core $n-1$ in the 3rd mode without further communication via local matrix multiplication in Line 5 (Figure 3.2(d)).

Lines 3 and 4 have the costs, given by (3.2) and (3.3) with $m = I_n R_n$ and $b = c = R_{n-1}$. Since the computation to form the explicit $\mathbf{Q}$ matrix exploits the sparsity structure of the identity matrix the constant 4 in (3.3) is reduced to 2. These two lines together cost

$$\gamma \cdot \left(4\frac{I_n R_n R_{n-1}^2}{P} + O(R_{n-1}^3 \log P)\right) + \beta \cdot O(R_{n-1}^2 \log P) + \alpha \cdot O(\log P).$$

Line 5 is a local triangular matrix multiplication costing $\gamma \cdot I_{k-1} R_{k-2} R_{k-1}^2 / P$. Assuming $I_k = I$ and $R_k = R$ for $1 \le k \le N-1$, the total cost of TT orthonormalization is then

$$\gamma \cdot \left(5\frac{NIR^3}{P} + O(NR^3 \log P)\right) + \beta \cdot O(NR^2 \log P) + \alpha \cdot O(N \log P). \tag{3.4}$$

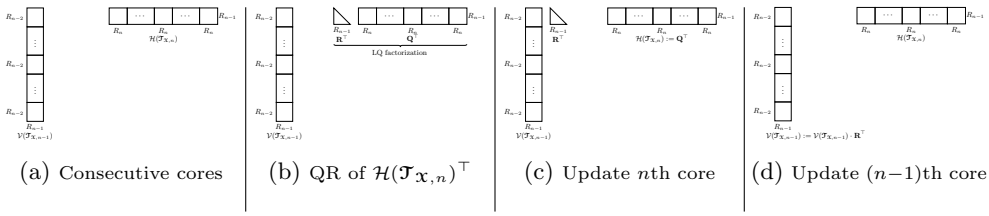
3.5. TT rounding. We present the parallel TT rounding procedure in Algorithm 3.4, which is a parallelization of Algorithm 2.2. The computation consists of two sweeps over the cores, one to orthonormalize and one to truncate. The algorithm shown performs right-orthonormalization and then truncates left to right, and the other ordering works analogously.

Algorithm 3.4 does not call Algorithm 3.3 to perform the orthonormalization sweep. This is because Algorithm 3.3 forms the orthonormalized cores explicitly, and

Algorithm 3.3 Parallel TT-Right-Orthonormalization

Require: $\mathcal{X}$ in TT format with each core 1D-distributed
Ensure: $\mathcal{X}$ is right orthonormal, in TT format with same distribution

1: **function** PAR-TT-RIGHT-ORTHONORMALIZATION($\{\mathcal{T}^{(p)}_{\mathcal{X},n}\}$)
2: **for** $n = N$ down to 2 **do**
3: $[\{\mathbf{Y}^{(p)}_{\ell,n}\}, \mathbf{R}_n] = \text{TSQR}(\mathcal{H}(\mathcal{T}^{(p)}_{\mathcal{X},n})^\top)$ ▷ QR factorization
4: $\mathcal{H}(\mathcal{T}^{(p)}_{\mathcal{X},n})^\top = \text{TSQR-APPLY-Q}(\{\mathbf{Y}^{(p)}_{\ell,n}\}, \mathbf{I}_{R_{n-1}})$ ▷ Form explicit $\mathbf{Q}$
5: $\mathcal{V}(\mathcal{T}^{(p)}_{\mathcal{X},n-1}) = \mathcal{V}(\mathcal{T}^{(p)}_{\mathcal{X},n-1}) \cdot \mathbf{R}_n{}^\top$ ▷ Apply $\mathbf{R}$ to previous core
6: **end for**
7: **end function**



(a) Consecutive cores (b) QR of $\mathcal{H}(\mathcal{T}_{\mathcal{X},n})^\top$ (c) Update nth core (d) Update $(n-1)$th core

FIG. 3.2. *Steps performed in TT right orthonormalization.*

Algorithm 3.4 can leave the orthonormalized cores from the first sweep in implicit form to be applied during the second sweep.

Iteration n of the right-to-left orthonormalization sweep occurs in Lines 3 and 4, which is illustrated in Figure 3.3, which matches Algorithm 3.3 except for the explicit formation of the orthonormal factor. Thus, the cost of the orthonormalization sweep is

$$\gamma \cdot \left(3\frac{NIR^3}{P} + O(NR^3 \log P)\right) + \beta \cdot O(NR^2 \log P) + \alpha \cdot O(N \log P). \tag{3.5}$$

At iteration n of the second loop, Lines 9 to 12 implement the left-to-right truncation procedure for the nth core in parallel (Figure 3.3(b)). Line 9 (Figure 3.3(b)) is a QR factorization and has cost given by 3.2 with $m = I_n L_{n-1}$ and $b = R_n$, as the number of rows of $\mathcal{V}(\mathcal{T}^{(p)}_{\mathcal{Y},n})$ has been reduced from $I_n R_{n-1}$ to $I_n L_{n-1}$ during iteration $n-1$:

$$\gamma \cdot \left(2\frac{I_n L_{n-1} R_n^2}{P} + O(R_n^3 \log P)\right) + \beta \cdot O(R_n^2 \log P) + \alpha \cdot O(\log P).$$

We note that we reuse the notation $\{\mathbf{Y}^{(p)}_{\ell,n}\}$ to store the implicit factorization; while the same variable stored the orthonormal factor of the nth core's horizontal unfolding from the orthonormalization sweep, it can be overwritten by this step of the algorithm (the set of matrices will now have different dimensions). Line 10 (Figure 3.3(c)) requires $O(R_n^3)$ flops, assuming the full SVD is computed before truncating. Line 11 (Figure 3.3(d)) implicitly applies an orthonormal matrix to an $R_n \times L_n$ matrix $\hat{\mathbf{U}}_R$ with cost given by Equation (3.3) with $m = I_n L_{n-1}$, $b = R_n$, and $c = L_n$:

$$\gamma \cdot \left(4\frac{I_n L_{n-1} R_n L_n}{P} + O(R_n^2 L_n \log P)\right) + \beta \cdot R_n L_n + \alpha.$$

Algorithm 3.4 Parallel TT-Rounding

Require: $\mathcal{X}$ in TT format with each core 1D-distributed over $1\times P\times 1$ processor grid
Ensure: $\mathcal{Y}$ in TT format with reduced ranks identically distributed across processors

1: **function** $\{\mathcal{T}_{\mathcal{Y},n}^{(p)}\}$ = PAR-TT-ROUNDING($\{\mathcal{T}_{\mathcal{X},n}^{(p)}\}, \epsilon$)
2: **for** $n = N$ down to 2 **do**
3: $[\{\mathbf{Y}_{\ell,n}^{(p)}\}, \mathbf{R}_n] = \text{TSQR}(\mathcal{H}(\mathcal{T}_{\mathcal{X},n}^{(p)})^\top)$ ▷ QR factorization
4: $\mathcal{V}(\mathcal{T}_{\mathcal{X},n-1}^{(p)}) = \mathcal{V}(\mathcal{T}_{\mathcal{X},n-1}^{(p)}) \cdot \mathbf{R}_n{}^\top$ ▷ Apply $\mathbf{R}$ to previous core
5: **end for**
6: Compute $\|\mathcal{X}\|$
7: $\mathcal{Y} = \mathcal{X}$
8: **for** $n = 1$ to $N-1$ **do**
9: $[\{\mathbf{Y}_{\ell,n}^{(p)}\}, \mathbf{R}_n] = \text{TSQR}(\mathcal{V}(\mathcal{T}_{\mathcal{Y},n}^{(p)}))$ ▷ QR factorization
10: $[\hat{\mathbf{U}}_R, \hat{\Sigma}, \hat{\mathbf{V}}] = \text{TSVD}(\mathbf{R}_n, \frac{\epsilon}{\sqrt{N-1}}\|\mathcal{X}\|)$ ▷ Redundant truncated SVD of $\mathbf{R}$
11: $\mathcal{V}(\mathcal{T}_{\mathcal{Y},n}^{(p)}) = \text{TSQR-APPLY-Q}(\{\mathbf{Y}_{\ell,n}^{(p)}\}, \hat{\mathbf{U}}_R)$ ▷ Form explicit $\hat{\mathbf{U}}$
12: $\mathcal{H}(\mathcal{T}_{\mathcal{Y},n+1}^{(p)})^\top = \text{TSQR-APPLY-Q}(\{\mathbf{Y}_{\ell,n+1}^{(p)}\}, \hat{\mathbf{V}}\hat{\Sigma})$ ▷ Apply $\hat{\mathbf{\Sigma}}\hat{\mathbf{V}}^\top$
13: **end for**
14: **end function**

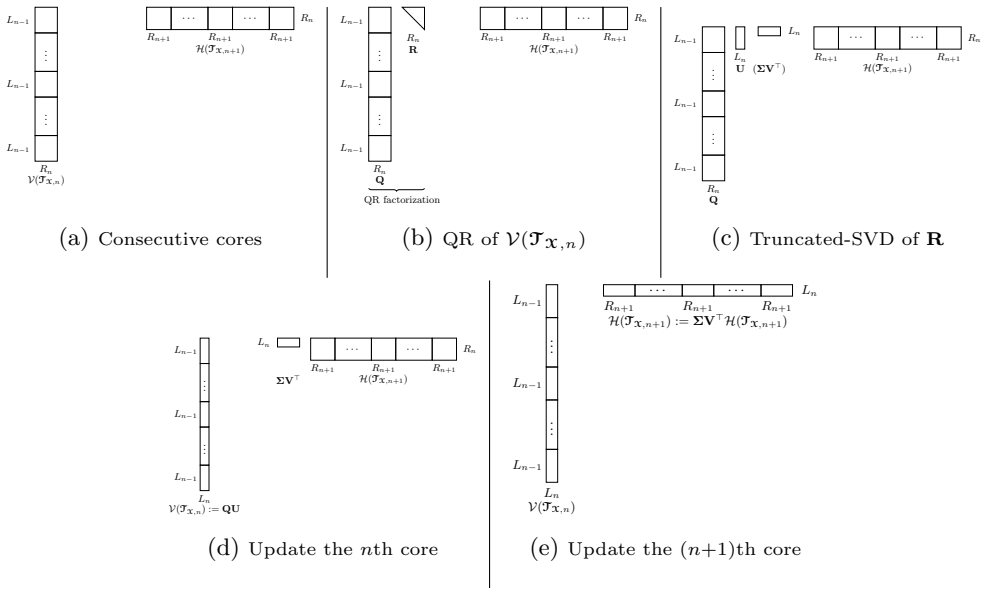


FIG. 3.3. *Steps performed in iteration of the TT left-to-right truncation.*

Line 12 (Figure 3.3(e)) implicitly applies an orthonormal matrix to an $R_n \times L_n$ matrix $\hat{\mathbf{V}}\hat{\mathbf{\Sigma}}$ with cost given by 3.3 with $m = I_{n+1}R_{n+1}$, $b = R_n$, and $c = L_n$:

$$\gamma \cdot \left(4\frac{I_{n+1}R_{n+1}R_nL_n}{P} + O(R_n^2 L_n \log P)\right) + \beta \cdot R_nL_n + \alpha.$$

Assuming $I_k = I$, $R_k = R$, and $L_k = L$ for $1 \le k \le N-1$, the total cost of

Algorithm 3.4 is then

(3.6)
$$\gamma \cdot \left(NIR\frac{3R^2 + 6RL + 4L^2}{P} + O(NR^3 \log P) \right) + \beta \cdot O(NR^2 \log P) + \alpha \cdot O(N \log P).$$

We note that leaving the orthonormal factors in implicit form during the orthonormalization sweep (as opposed to calling Algorithm 3.3) saves up to 40% of the computation, when the reduced ranks L_n are much smaller than the original ranks R_n. As the rank reduction diminishes, so does the advantage of the implicit optimization. For example, when ranks are all halved, the reduction in leading order flop cost is 12.5%.

4. Numerical experiments. In this section we present performance results for TT computations using synthetic tensors with mode and dimension parameters inspired by physics and chemistry applications, as described in subsection 4.2. We first present microbenchmarks in subsection 4.3 to justify key design decisions, and then demonstrate performance efficiency and parallel scaling in subsection 4.4.

All numerical experiments were performed on the Max Planck Society supercomputer COBRA. All computation nodes contain two Intel Xeon Gold 6148 processors (Skylake, 20 cores each at 2.4 GHz) and 192 GB of memory, and the nodes are connected through a 100 Gb/s OmniPath interconnect. We link to Intel Math Kernel Library (MKL) 2020.1 for single-threaded BLAS and LAPACK subroutines.

4.1. Motivating applications. We describe in this section the motivating applications guiding the choice of tensor dimensions and ranks of the synthetic models we use in the experiments.

4.1.1. High-order correlation functions. In the study of stochastic processes, Gaussian random fields are widely used. If f is a Gaussian random field defined on a bounded domain $\Omega \subset \mathbb{R}^d$ ($d = 1, 2, 3$), an N-point correlation function for f is defined on Ω^N. The discretization of the domain determines the N-way tensor dimensions. These N-point correlation functions can often be efficiently approximated in TT format [14, 32]. For typical discretizations, the number of discretization points in the domain Ω can be extremely large leading to tensors with very large dimensions. In order to compute some desired information about the random solution of a stochastic PDE such as its expected value, TT computations including addition and scaling are required. Thus, compressing the resulting TT tensors is required to maintain the tractability of computations. In [14] the authors present a study of single-phase fluid flows in heterogeneous porous media. Due to memory and time constraints, current implementations of TT arithmetic allows one to only perform the aforementioned computations on a moderate size discretizations (10,000) for $d = 1$ or $d = 2$. However, in industrial applications where $\Omega \subset \mathbb{R}^3$, the mode dimension can be of order 10^8.

4.1.2. Molecular simulations. Another important class of applications is molecular simulations. For example, when a spin system can be considered as a weakly branched linear chain, it is typical to represent it as a TT tensor [46]. Each branch is then considered as a spatial coordinate (mode). The number of branches, corresponding to the number of tensor modes, can be arbitrarily large; for example, a simple backbone protein may have hundreds of branches. The TT representation is then inherited from the weak correlation between the branches. However, in the same branch, the correlation cannot be ignored, and thus the exponential growth in the

TABLE 4.1
Synthetic TT models used for performance experiments. In each case the TT ranks are all the same and are cut in half by the TT rounding procedure.

Model	# Modes	Dimensions	Ranks	Memory
1	50	$2K \times \cdots \times 2K$	50	2 GB
2	16	$100M \times 50K \times \cdots \times 50K \times 1M$	30	28 GB
3	30	$2M \times \cdots \times 2M$	30	385 GB

number of states, which corresponds to the dimension of the tensor mode for that branch, cannot be avoided.

4.1.3. Parameter-dependent PDEs. In this application, one or a few modes may be much larger than the rest. This is typically the case in physical applications such as parameter-dependent PDEs, stochastic PDEs, uncertainty quantification, and optimal control systems [9, 10, 11, 19, 25, 34, 43]. In such applications, the spatial discretization leads to a high number of degrees of freedom. This typically results from large domains, refinement procedures, and a large number of parameter samples. Most of other modes correspond to control or uncertainty parameters and can have relatively smaller dimension.

For example, in [10] where the authors study an optimal control problem constrained by random Navier–Stokes equations, certain vectors are represented by 10-mode tensors. The number of degrees of freedom in each mode is as follows: the velocity field has up to 168,240, the time mode has up to 4,096, and the eight modes related to the random variables each has 8. Again, this discretization is limited by memory and time constraints and finer granularity that increases the accuracy of the approximation would lead to dimensions on the order of millions.

4.2. Synthetic TT models. As we are interested in large scale systems, we consider two contexts of applications in which a large number of modes exists. The first context is with each mode of relatively the same (large) dimension, such as the applications described in subsections 4.1.1 and 4.1.2, and the second context is a single or few modes with large dimension as well as many modes of relatively smaller dimension, as arises in parameter-dependent PDEs (subsection 4.1.3). Table 4.1 presents the details of the three models of synthetic tensors we use in the experiments, in order of their memory size. The first and third models correspond to the first context (all modes of the same dimension) and the second model corresponds to the second context (two large modes and many more smaller modes). The first model is chosen to be small enough to be processed by a single core, while the second and third are larger and benefit more from distributed-memory parallelization (the third does not fit in the memory of a single node). The paragraphs below describe the applications that inspire these choices of modes and dimensions.

In all experiments, we generate a random TT tensor $\mathcal{X}$ with a given number of modes N, modes sizes I_n for $n = 1, \ldots, N$, and TT ranks $R_n^{\mathcal{X}}$ for $n = 1, \ldots, N-1$. Then, we form the TT tensor $\mathcal{Y} = 2\mathcal{X} - \mathcal{X}$ whose representation has TT ranks $R_n^{\mathcal{Y}} = 2R_n^{\mathcal{X}}$ for $n = 1, \ldots, N-1$. The algorithms are then applied on the TT tensor $\mathcal{Y}$. Note that the minimal TT ranks of $\mathcal{Y}$ are less than or equal to the TT ranks of $\mathcal{X}$.

4.3. Microbenchmarks. We next present experimental results for microbenchmarks to justify our choices for subroutine algorithms and optimizations. The results presented in subsection 4.4 use the best-performing variants and optimizations demonstrated in this section.

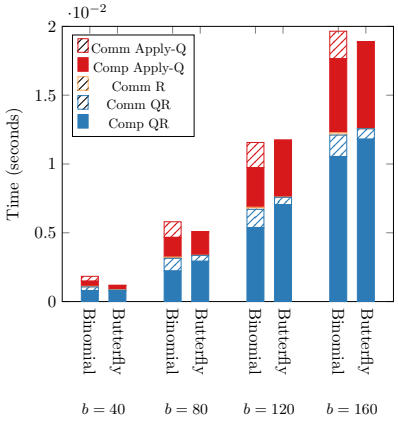


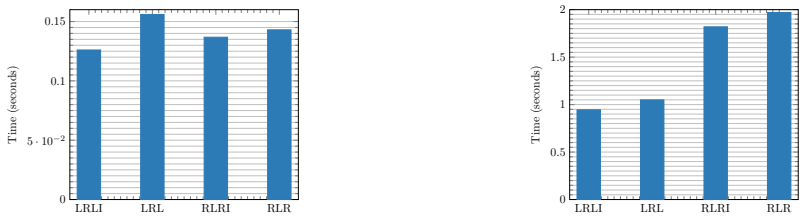
FIG. 4.1. *Time breakdown for TSQR variants for* $1{,}024{,}000 \times b$ *matrix over* $1,024$ *processors, including both factorization and application of the orthonormal factor to a dense* $b \times b$ *matrix.*

4.3.1. TSQR. As discussed in subsection 3.3, the TSQR algorithm depends on a hierarchical tree. Two tree choices are commonly used in practice, the binomial tree and the butterfly tree. In both cases the TSQR computes the QR decomposition sharing the same complexity and communication costs along the critical path, whereas the butterfly requires less communication cost along the critical path of the application of the implicit orthonormal factor. This advantage of the butterfly variant in the application phase is particularly important in the context of TT orthonormalization and rounding because a large percentage of time is spent in the application phase.

Here we compare the performance of the TSQR algorithms using the binomial and butterfly trees for both factorization and single application of the orthonormal factor. Since the difference in their costs is solely related to the number of columns, we fix the number of rows in the comparison and vary the number of columns. Figure 4.1 reports the breakdown of time of the variants using 256 nodes with 4 MPI processes per node (2 cores per socket). The local matrix size on each processor is $1{,}000 \times b$ where b varies in $\{40, 80, 120, 160\}$. We observe that the butterfly tree has better performance in terms of communication time in the application phase. Note that the factorization runtime (computation and communication) is relatively the same for both variants. We also time the cost of communicating the triangular factor $\mathbf{R}$, which is required of the binomial variant in the context of TT-rounding, but that cost is negligible in these experiments.

Based on these results (and corroborating experiments with various other parameters), we use the butterfly variant of TSQR for TT computations that require TSQR in all subsequent numerical experiments.

4.3.2. TT rounding. In this section, we consider four variants of TT rounding (Algorithm 3.4), based on the orthonormalization/truncation ordering and the use of the implicit orthonormal factor optimization. As discussed in subsection 2.4, the rounding procedure can perform right- or left- orthonormalization followed by a truncation phase in the opposite direction. We refer to the ordering based on right-orthonormalization and left-truncation as RLR and the ordering based on left-orthonormalization and right-truncation as LRL. The implicit optimization avoids the explicit formation of orthonormal factors during the orthonormalization phase; instead of using Algorithm 3.3 as a black-box subroutine, Algorithm 3.4 leaves or-



(a) Model 2 (b) Model 3

FIG. 4.2. *Performance comparison of TT-Rounding variants for large TT models on* 32 *nodes* (1,280 *cores). LRL refers to left-orthonormalization followed by right-truncation (vice versa for RLR) and I indicates the use of the implicit optimization.*

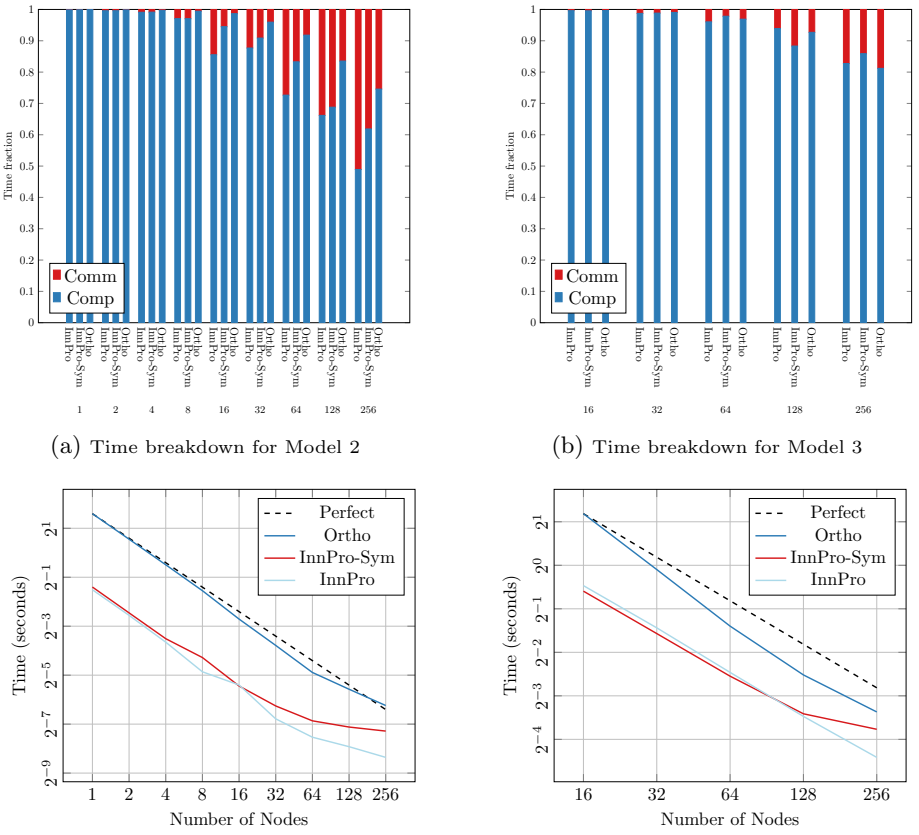
thonormal factors in implicit TSQR form as much as possible, saving a constant factor of computation (and a small amount of communication).

Although the asymptotic complexity of the variants of the rounding procedure are equal, their performance is not the same. This disparity between RLR and LRL orderings is because of the performance difference between the QR and the LQ implementations of the LAPACK subroutines provided by the MKL implementations. Despite the same computation complexity, the QR subroutines has much better performance than the LQ subroutines.

In the LRL ordering, a sequence of calls to the QR subroutine are performed on the vertically unfolded TT cores $\mathcal{T}_{\mathcal{X},n}$ with the increased ranks R_{n-1}, R_n. Along the truncation sweep, the LQ subroutine is called in a sequence to factor the horizontally unfolded TT cores $\mathcal{T}_{\mathcal{X},n}$ with one reduced rank R_{n-1}, L_n. As presented in subsections 3.4 and 3.5, the RLR ordering employs the QR and LQ subroutines in the opposite order. Because the truncation phase involves less computation within local QR/LQ subroutine calls than the orthonormalization phase, the LRL ordering has the advantage that it spends less time in LQ subroutine calls than the RLR ordering.

The effect of the implicit optimization is a reduction in computation (approximately 12.5% in these experiments) and communication, but this advantage is offset in part by the performance of local subroutines. The implicit application of the orthonormal factor involves auxiliary LAPACK routines for applying sets of Householder vectors in various formats. The explicit multiplication of an orthonormal factor to a small square matrix involves a broadcast and a local subroutine call to matrix multiplication, which has much higher performance than the auxiliary routines involving Householder vectors. We use an "I" to indicate the use of the implicit optimization, so that the four variants are LRLI, LRL, RLRI, and RLR.

Figure 4.2 presents the performance results for TT Models 2 and 3 running on 256 nodes. We see that for both models, the LRL ordering with the implicit optimization (LRLI) is the fastest. In the case of Model 2, the implicit optimization makes more of a difference than the ordering. This is because a considerable amount of time is spent in the first mode, where the QR is used (once) in either ordering. In the case of Model 3, the ordering makes a much larger difference in running time, as the internal modes dominate the running time and the QR/LQ difference has a large effect. The implicit optimization still improves performance, but it has less of an effect than the ordering. Based on these results, we use the LRLI variant of TT-rounding in all the experiments presented in subsection 4.4.



(a) Time breakdown for Model 2

(b) Time breakdown for Model 3

(c) Parallel scaling for Model 2

(d) Parallel scaling for Model 3

FIG. 4.3. *Time breakdown and parallel scaling of variants for TT norm computation. "Ortho" refers to orthonormalization (following by computing the norm of a single core), "InnPro" refers to using the inner product algorithm, and "InnPro-Sym" refers to using the inner product algorithm with symmetric optimization.*

4.4. Parallel scaling.

4.4.1. Norms. In this section we compare the performance and parallel scaling of three different algorithms for computing the norm of a TT tensor as discussed in subsection 3.2.4. We focus on this computation because the multiple approaches represent the performance of algorithms for computing inner products and orthonormalization, which are essential on their own in other contexts. We use "Ortho" to denote the approach of first right- or left-orthonormalizing the TT tensor and then (cheaply) computing the norm of the first or last core, respectively. Thus, Ortho performance represents that of Algorithm 3.3. The name "InnPro" refers to the approach of computing the inner product of the TT tensor with itself, and "InnPro-Sym" includes the optimization that exploits the symmetry in the inner product to save up to half the computation. InnPro captures the performance of the algorithm described in subsection 3.2.3 for general TT inner products as well.

We report parallel scaling and a breakdown of computation and communication for all three algorithms and TT Models 2 and 3 in Figure 4.3. Model 2 can be processed

on a single node, but Model 3 requires 16 nodes to achieve sufficient memory; we scale both models up to 256 nodes (10,240 cores). Based on the theoretical analysis (see Table 3.1), when all tensor dimensions are equivalent such as Model 3, Ortho has a leading-order flop constant of 5, InnPro has a constant of 4, and InnPro-Sym has a constant of 2. Ortho also requires more complicated TSQR reductions compared to the All-Reduces performed in InnPro and InnPro-Sym, involving an extra $\log P$ factor in data communicated in theory and slightly less efficient implementations in practice. In addition, the efficiencies of the local computations differ across approaches: Ortho is bottlenecked by local QR, InnPro by local matrix multiplication (GEMM), and InnPro-Sym by local triangular matrix multiplication (TRMM).

Overall, we see that InnPro is typically the best performing approach. The main factor in its superiority is that its computation is cast as GEMM calls, which are more efficient than TRMM and QR subroutines. Although InnPro-Sym performs half the flops of InnPro, the relative inefficiency of those flops translates to a less than $2\times$ speedup over InnPro for Model 3 and a slight slowdown for Model 2. We also note that for high node counts, the cost of the LDLT factorization performed within InnPro-Sym becomes nonneglible and begins to hinder parallel scaling.

Based on the breakdown of computation and communication, we see that all three approaches are able to scale reasonably well because they remain computation bound up to 256 nodes. For Model 2, we see that communication costs are relatively higher, as that tensor is much smaller. Note that Ortho scales better than InnPro-Sym and InnPro, even superlinearly for Model 3, which is due in large part to the higher flop count and relative inefficiency of the local QRs, allowing it to remain more computation bound than the alternatives. Overall, these results confirm that the parallel distribution of TT cores allows for high performance and scalability of the basic TT operations as described in subsection 3.2.

4.4.2. TT rounding.

Single-node performance. We compare in this section our implementation of TT rounding against the MATLAB TT-Toolbox [39] rounding process. Table 4.2 presents a performance comparison on a single node of COBRA, which has 40 cores available. We run the experiment on TT Model 1, which is small enough to be processed by a single core. Because it is written in MATLAB, the TT-Toolbox accesses the available parallelism only through underlying calls to a multithreaded implementation of BLAS and LAPACK. However, the bulk of the computation occurs in MATLAB functions that make direct calls to efficient BLAS and LAPACK subroutines, so it can achieve relatively high sequential performance.

We observe from Table 4.2 that the single-core performance of the two implementations is similar, with a 70% speedup from our implementation. The single-core implementations are employing the same algorithm, and we attribute the speedup to our lower-level interface to LAPACK subroutines and the ability to maintain implicit orthonormal factors to reduce computation. The parallel strong scaling differs more drastically, as expected. The MATLAB implementation, which is not designed for parallelization, achieves less than a $2\times$ speedup when using 20 or 40 cores. Our parallelization, which is designed for distributed-memory systems, also scales very well on this shared-memory machine, achieving over $20\times$ speedup on 20 cores and $34\times$ speedup on 40 cores.

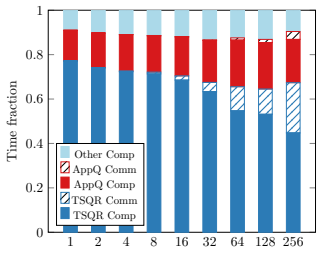
Distributed-memory strong scaling. We now present the parallel performance of TT rounding scaling up to hundreds of nodes (over 10,000 cores). As in the case of subsection 4.4.1, we consider Models 2 and 3. Figure 4.4 presents the relative time

TABLE 4.2
Single-node performance results on TT Model 1 *and comparison with the MATLAB TT-Toolbox.*

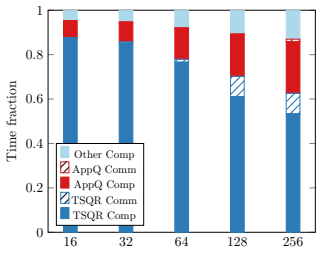
	1 core	20 cores	Par. speedup	40 cores	Par. speedup
TT-Toolbox	15.68	8.34	**1.9**×	8.752	**1.8**×
Our implementation	9.2	0.44	**20.9**×	0.27	**33.9**×
Speedup	**1.7**×	**18.95**×		**32.2**×	

breakdown and raw timing numbers for each model. We use the "LRLI" variant of TT rounding in these experiments per the results of subsection 4.3.2. As in other rounding experiments, the ranks are cut in half for each model.

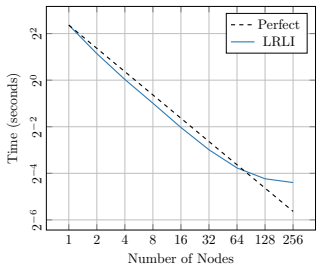
In the time breakdown plots of Figures 4.4(a) and 4.4(b), we distinguish among TSQR factorization (TSQR), application of orthonormal factors (AppQ), and the rest of the computation that includes SVDs and triangular multiplication (Other). We also separate the computation and communication of each category. In the context of Algorithm 3.4, TSQR corresponds to lines 3 and 9, AppQ corresponds to lines 11 and 12, and Other corresponds to lines 4 and 10.



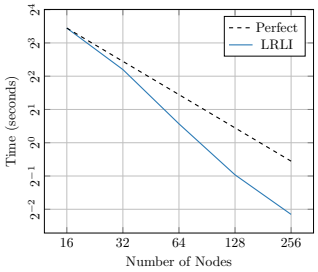
(a) Time breakdown for Model 2



(b) Time breakdown for Model 3



(c) Parallel scaling for Model 2



(d) Parallel scaling for Model 3

FIG. 4.4. *Time breakdown and and parallel scaling of LRLI variant of TT rounding.*

In Figures 4.4(c) and 4.4(d), we observe the strong scaling raw times in log scale compared to perfect scaling (based on time at the fewest number of nodes). We see nearly perfect scaling for Model 2 until 128 nodes; time continues to decrease but is not cut in half when scaling to 256 nodes. The parallel speedup numbers for Model 2 are 97× for 128 nodes and 108× for 256 nodes, compared to performance on 1 node. In the case of Model 3, we see superlinear scaling, even at 256 nodes. We attribute this scaling in part to the baseline comparison of 16 nodes, which already involves parallelization/communication, and in part to local data fitting into higher levels of cache as the number of processors increases, which helps memory-bound local

computations. We observe a 48× speedup for Model 3, scaling from 16 to 256 nodes.

The time breakdown plots also help to explain the scaling performance. We see that for Model 2, over 70% of the time is spent in local computation, while for Model 3, over 90% of the time is computation. Of this computation, the majority is spent in TSQR, which itself is dominated by the initial local leaf QR computations. If the rank is reduced by a smaller factor, then relatively more flops will occur in AppQ. We note that AppQ involves minimal communication because of the use of the Butterfly TSQR variant. The Other category is dominated by the triangular matrix multiplication, which achieves higher performance than the LAPACK subroutines involving orthonormal factors.

5. Conclusions. This work presents the parallel implementation of the basic computational algorithms for tensors represented in low-rank TT format. Because most TT computations involve dependence through the train, we specify a data distribution that distributes each core across all processors and show that the computations and communication costs of our proposed algorithms enable efficiency and scalability for each core computation. The orthonormalization and rounding procedures for TT tensors depend heavily on the TSQR algorithm, which is designed to scale well on architectures with a large number of processors for matrices with highly skewed aspect ratios. Our numerical experiments show that our algorithms are indeed efficient and scalable, outperforming productivity-oriented implementations on a single core and single node and scaling well to hundreds of nodes (thousands of cores). Thus, our approach is useful to applications and users who are restricted to a single workstation as well to those requiring the memory and performance of a supercomputer.

We note that the raw performance of our implementation depends heavily on the local BLAS/LAPACK implementation and the efficiency of the QR decomposition and related subroutines. For example, we observe significant performance differences between MKL's implementations of QR and LQ subroutines, which caused the LRL ordering of TT-rounding to outperform RLR. We also observe performance differences among other subroutines, such as triangular matrix multiplication and general matrix multiplication, again confirming that simple flop counting (even tracking constants closely) does not always accurately predict running times.

There do exist limitations of the parallelization approach proposed in this paper. In particular, modes with small dimensions benefit less from parallelization and can become bottlenecks if there are too many of them. For example, we see the limits of scalability with TT Model 2, which has large first and last modes but smaller internal modes. In fact, the distribution scheme assumes that $P \leq I_n$ for $n = 1, \ldots, N$, and involves idle processors when the assumption is broken. We also note that TSQR may not be the optimal algorithm to factor the unfolding, which can happen if two successive ranks differ greatly and P is large with respect to the original tensor dimensions.

Alternative possibilities to avoid these limitations include cheaper but less accurate methods for the SVD, including via the associated Gram matrices or by using randomization. We plan to pursue such strategies in the future, in addition to considering the case of computing a TT approximation from a tensor in explicit full format. Given these efficient computational building blocks, the next step is to build scalable Krylov and alternating-scheme based solvers that exploit the TT format.

REFERENCES

[1] H. Al Daas, *Solving linear systems arising from reservoirs modeling*, theses, Inria Paris, Sorbonne Université, UPMC University of Paris 6, Laboratoire Jacques-Louis Lions, 2018, https://hal.inria.fr/tel-01984047.

[2] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, *Communication-avoiding QR decomposition for GPUs*, in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, Washington, DC, 2011, IEEE Computer Society, pp. 48–58, https://doi.org/10.1109/IPDPS.2011.15.

[3] W. Austin, G. Ballard, and T. G. Kolda, *Parallel tensor compression for large-scale scientific data*, in Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium, 2016, pp. 912–922, https://www.computer.org/csdl/proceedings/ipdps/2016/2140/00/2140a912-abs.html.

[4] B. W. Bader, T. G. Kolda, et al., *Tensor Toolbox for MATLAB Version* 3.2.1, 2021, https://www.tensortoolbox.org.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page*, 2019, https://www.mcs.anl.gov/petsc.

[6] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, *Communication lower bounds and optimal algorithms for numerical linear algebra*, Acta Numer., 23 (2014), pp. 1–155, https://doi.org/10.1017/S0962492914000038.

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

[8] G. Ballard, A. Klinvex, and T. G. Kolda, *TuckerMPI: A parallel C++/MPI software package for large-scale data compression via the Tucker tensor decomposition*, ACM Trans. Math. Software, 46 (2020), 13.

[9] P. Benner, S. Dolgov, A. Onwunta, and M. Stoll, *Low-rank solvers for unsteady Stokes–Brinkman optimal control problem with random data*, Comput. Methods Appl. Mech. Engrg., 304 (2016), pp. 26–54.

[10] P. Benner, S. Dolgov, A. Onwunta, and M. Stoll, *Low-rank solution of an optimal control problem constrained by random Navier-Stokes equations*, Internat. J. Numer. Methods Fluids, 92 (2020), pp. 1653–1678.

[11] P. Benner, S. Gugercin, and K. Willcox, *A survey of projection-based model reduction methods for parametric dynamical systems*, SIAM Rev., 57 (2015), pp. 483–531, https://doi.org/10.1137/130932715.

[12] G. Beylkin and M. J. Mohlenkamp, *Numerical operator calculus in higher dimensions*, Proc. Natl. Acad. Sci. USA, 99 (2002), pp. 10246–10251, https://doi.org/10.1073/pnas.112329799.

[13] M. Bhattarai, G. Chennupati, E. Skau, R. Vangara, H. Djidjev, and B. Alexandrov, *Distributed non-negative tensor train decomposition*, in Proceedings of the 2020 IEEE High Performance Extreme Computing Conference (HPEC), 2020, pp. 1–10, https://doi.org/10.1109/HPEC43674.2020.9286234.

[14] F. Bonizzoni, F. Nobile, and D. Kressner, *Tensor train approximation of moment equations for elliptic equations with lognormal coefficient*, Comput. Methods Appl. Mech. Engrg., 308 (2016), pp. 349–376.

[15] J. D. Carroll and J.-J. Chang, *Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition*, Psychometrika, 35 (1970), pp. 283–319, https://doi.org/10.1007/BF02310791.

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

[17] P. G. Constantine and D. F. Gleich, *Tall and skinny QR factorizations in mapreduce architectures*, in Proceedings of the Second International Workshop on MapReduce and Its Applications, MapReduce '11, New York, 2011, ACM, pp. 43–50, https://doi.org/10.1145/1996092.1996103.

[18] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. A206–A239, https://doi.org/10.1137/080731992.

[19] S. Dolgov and M. Stoll, *Low-rank solution to an optimization problem constrained by the*

Navier-Stokes equations, SIAM J. Sci. Comput., 39 (2017), pp. A255–A280, https://doi.org/10.1137/15M1040414.

[20] S. Eswar, K. Hayashi, G. Ballard, R. Kannan, M. A. Matheson, and H. Park, *PLANC: Parallel low-rank approximation with nonnegativity constraints*, ACM Trans. Math. Softw., 47 (2021), 20, https://doi.org/10.1145/3432185.

[21] L. Grigori and S. Kumar, *Parallel Tensor Train Through Hierarchical Decomposition*, Tech. Report hal-03081555, INRIA, 2021, https://hal.inria.fr/hal-03081555.

[22] W. Hackbusch and S. Kühn, *A new scheme for the tensor representation*, J. Fourier Anal. Appl., 15 (2009), pp. 706–722, https://doi.org/10.1007/s00041-009-9094-9.

[23] R. A. Harshman, *Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis*, Working Papers in Phonetics, 16 (1970), pp. 1–84, http://www.psychology.uwo.ca/faculty/harshman/wpppfac0.pdf.

[24] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Trans. Math. Software, 31 (2005), pp. 397–423, https://doi.org/10.1145/1089014.1089021.

[25] J. Hesthaven, G. Rozza, and B. Stamm, *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*, SpringerBriefs in Mathematics, Springer, 2015, Cham, https://doi.org/10.1007/978-3-319-22470-1.

[26] P. Jolivet, *Domain Decomposition Methods: Application to High-performance Computing*, Theses, Université de Grenoble, 2014, https://tel.archives-ouvertes.fr/tel-01155718.

[27] A. Kantian, M. Dolfi, M. Troyer, and T. Giamarchi, *Understanding repulsively mediated superconductivity of correlated electrons via massively parallel density matrix renormalization group*, Phys. Rev. B, 100 (2019), 075138, https://doi.org/10.1103/PhysRevB.100.075138.

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

[29] B. N. Khoromskij, *$O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling*, Constr. Approx., 34 (2011), pp. 257–280, https://doi.org/10.1007/s00365-011-9131-1.

[30] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500, https://doi.org/10.1137/07070111X.

[31] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, *TensorLy: Tensor learning in Python*, J. Mach. Learn. Res., 20 (2019), pp. 1–6, http://jmlr.org/papers/v20/18-277.html.

[32] D. Kressner, R. Kumar, F. Nobile, and C. Tobler, *Low-rank tensor approximation for high-order correlation functions of Gaussian random fields*, SIAM/ASA J. Uncertain. Quantif., 3 (2015), pp. 393–416, https://doi.org/10.1137/140968938.

[33] D. Kressner and L. Periša, *Recompression of Hadamard products of tensors in Tucker format*, SIAM J. Sci. Comput., 39 (2017), pp. A1879–A1902, https://doi.org/10.1137/16M1093896.

[34] D. Kressner and C. Tobler, *Krylov subspace methods for linear systems with tensor product structure*, SIAM J. Matrix Anal. Appl., 31 (2009/10), pp. 1688–1714, https://doi.org/10.1137/090756843.

[35] R. Levy, E. Solomonik, and B. K. Clark, *Distributed-memory DMRG via sparse and dense parallel tensor contractions*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, 2020, 24.

[36] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, *Model-driven sparse CP decomposition for higher-order tensors*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, pp. 1048–1057, https://doi.org/10.1109/IPDPS.2017.80.

[37] L. Li, W. Yu, and K. Batselier, *Faster Tensor Train Decomposition for Sparse Data*, preprint, https://arxiv.org/abs/1908.02721, 2020.

[38] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, *Minimizing communication in sparse matrix solvers*, in Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 2009, 36, https://doi.org/10.1145/1654059.1654096.

[39] I. Oseledets et al., *Tensor Train Toolbox Version* 2.2.2. 2020, https://github.com/oseledets/TT-Toolbox.

[40] I. Oseledets and E. Tyrtyshnikov, *TT-cross approximation for multidimensional arrays*, Linear Algebra Appl., 432 (2010), pp. 70–88, https://doi.org/10.1016/j.laa.2009.07.024.

[41] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317, https://doi.org/10.1137/090752286.

[42] A.-H. Phan, P. Tichavsky, and A. Cichocki, *Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations*, IEEE Trans. Signal Process., 61 (2013), pp. 4834–4846, https://doi.org/10.1109/TSP.2013.2269903.

[43] A. Quarteroni, A. Manzoni, and F. Negri, *Reduced Basis Methods for Partial Differential Equations: An Introduction*, UNITEXT, Springer, Cham, 2015, https://doi.org/10.1007/978-3-319-15431-2.

[44] S. Ragnarsson and C. F. Van Loan, *Block tensor unfoldings*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 149–169, https://doi.org/10.1137/110820609.

[45] M. Röhrig-Zöllner, J. Thies, and A. Basermann, *Performance of Low-rank Approximations in Tensor Train Format (TT-SVD) for Large Dense Tensors*, preprint, https://arxiv.org/abs/2102.00104, 2021.

[46] D. V. Savostyanov, S. V. Dolgov, J. M. Werner, and I. Kuprov, *Exact NMR simulation of protein-size spin systems using tensor train formalism*, Phys. Rev. B, 90 (2014), 085139, https://doi.org/10.1103/PhysRevB.90.085139.

[47] S. Smith and G. Karypis, *Accelerating the Tucker decomposition with compressed sparse tensors*, in Euro-Par 2017, Lecture Notes in Comput. Sci. 10417, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, eds., Springer, Cham, 2017, pp. 653–668, https://doi.org/10.1007/978-3-319-64203-1_47.

[48] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, *SPLATT: Efficient and parallel sparse tensor-matrix multiplication*, in Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS '15, Washington, DC, 2015, IEEE Computer Society, pp. 61–70, https://doi.org/10.1109/IPDPS.2015.27.

[49] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, *A massively parallel tensor contraction framework for coupled-cluster computations*, J. Parallel Distrib. Comput., 74 (2014), pp. 3176–3190, https://doi.org/10.1016/j.jpdc.2014.06.002.

[50] E. M. Stoudenmire and S. R. White, *Real-space parallel density matrix renormalization group*, Phys. Rev. B, 87 (2013), 155137, https://doi.org/10.1103/physrevb.87.155137.

[51] R. Thakur, R. Rabenseifner, and W. Gropp, *Optimization of collective communication operations in MPICH*, Int. J. High Perform. Comput. Appl., 19 (2005), pp. 49–66, https://doi.org/10.1177/1094342005051521.

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

[53] E. E. Tyrtyshnikov, *Tensor approximations of matrices generated by asymptotically smooth functions*, Sb. Math., 194 (2003), pp. 941–954, https://doi.org/10.1070/sm2003v194n06abeh000747.

[54] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, *Tensorlab* 3.0, http://www.tensorlab.net, 2016.

[55] X. Wang, L. T. Yang, Y. Wang, L. Ren, and M. J. Deen, *ADTT: A highly efficient distributed tensor-train decomposition method for IIoT big data*, IEEE Trans. Industr. Inform., 17 (2021), pp. 1573–1582, https://doi.org/10.1109/TII.2020.2967768.