SAND2020-8515

SANDIA REPORT

Printed

Sandia National Laboratories

GentenMPI: Distributed Memory Sparse Tensor Decomposition

Karen Devine, Grey Ballard

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

GentenMPI is a toolkit of sparse canonical polyadic (CP) tensor decomposition algorithms that is designed to run effectively on distributed-memory high-performance computers. Its use of distributed-memory parallelism enables it to efficiently decompose tensors that are too large for a single compute node's memory. GentenMPI leverages Sandia's decades-long investment in the Trilinos solver framework for much of its parallel-computation capability. Trilinos contains numerical algorithms and linear algebra classes that have been optimized for parallel simulation of complex physical phenomena. This work applies these tools to the data science problem of sparse tensor decomposition. In this report, we describe the use of Trilinos in GentenMPI, extensions needed for sparse tensor decomposition, and implementations of the CP-ALS (CP via alternating least squares [4, 7]) and GCP-SGD (generalized CP via stochastic gradient descent [11, 12, 17]) sparse tensor decomposition algorithms. We show that GentenMPI can decompose sparse tensors of extreme size, e.g., a 12.6-terabyte tensor on 8192 computer cores. We demonstrate that the Trilinos backbone provides good strong and weak scaling of the tensor decomposition algorithms.

ACKNOWLEDGMENT

We thank Tammy Kolda for her extensive motivation and guidance in this work; she is too selfless to accept that she should be a co-author of this technical report, even though it was our intent to include her as such. We also thank Eric Phipps and Shaden Smith for their guidance on running the software packages Genten and SPLATT, respectively. And we thank Eric Phipps, Chris Siefert, Rich Vuduc, and Jeff Young for helpful discussions and suggestions.

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

GentenMPI is a toolkit of sparse tensor decomposition algorithms that is designed to run effectively on distributed-memory high-performance computers. Its use of distributed-memory parallelism enables it to efficiently operate on tensors that are too large for a single compute node's memory. And its use of the Trilinos framework's Tpetra linear algebra classes delivers scalable performance on large number of processors.

Tensor decomposition is a valuable tool in data analysis and unsupervised machine learning. Kolda and Bader [16] provide a complete mathematical description of tensor operations and survey of tensor decomposition methods. Here, we present only the details needed to describe GentenMPI.

For simplicity, we present algorithms using a three-way tensor $\mathcal{X}$ with dimensions $I \times J \times K$. GentenMPI, however, handles tensors of arbitrary order and, indeed, many of the results presented are for tensors with order greater than three. $\mathcal{X}$ is assumed to be sparse; that is, most tensor entries x_{ijk} are zero.

Tensor decomposition can be see as an extension of matrix decomposition to higher order. The commonly used Canonical Polyadic (CP) decomposition [4, 7] is a tensor decomposition in which tensor $\mathcal{X}$ is approximated by $\mathcal{M}$, the sum of R rank-one tensors. For a three-way tensor, CP decomposition can be written as

$$x_{ijk} \approx m_{ijk} = \sum_{r=1}^{R} \lambda_r a_{ir} b_{jr} c_{kr} \tag{1.1}$$

where a_{ir}, b_{jr} and c_{kr} are entries of factor matrices $A \in \mathbb{R}^{I \times R}$, $B \in \mathbb{R}^{J \times R}$, and $C \in \mathbb{R}^{K \times R}$, respectively, and $\lambda \in \mathbb{R}^R$ is a weighting vector. We refer to $\mathcal{M}$ as the *model* and represent it by the *Kruskal* tensor $[\lambda; A, B, C]$. The goal is to minimize the difference between $\mathcal{X}$ and $\mathcal{M}$ with respect to some loss function $f(x,m)$

$$\text{minimize } F(\mathcal{X}, \mathcal{M}) \equiv \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} f(x_{ijk}, m_{ijk}) \quad \text{subject to} \quad \text{rank}(\mathcal{M}) \leq R. \tag{1.2}$$

The CP-ALS (canonical polyadic decomposition via alternating least squares) method [4, 7] uses an L^2 loss function $f(x,m) \equiv (x-m)^2$ in Equation 1.2 and an alternating least squares approach to solve the optimization; details are in Chapter 5. In their generalized CP (GCP) tensor decomposition, Hong, Kolda and Duersch [11, 12] support general loss functions. By providing appropriate loss functions, users can better represent tensors with special form, such as tensors with binary-valued or non-negative-valued data. A method for solving GCP's optimization via

stochastic gradient descent (SGD) was proposed by Kolda and Hong [17]; this algorithm is described in Chapter 6.

The Matlab toolkit Tensor Toolbox [1, 2] provides implementations of both CP-ALS and GCP-SGD tensor decomposition. The C++ toolkit GenTen [18, 19] builds on the Kokkos [5, 6] performance portability library to provide CP-ALS and GCP-SGD implementations that can run on multicore CPUs and GPUs. In distributed memory systems, the SPLATT [21, 22] library performs CP-ALS using OpenMP for on-node multithreading and MPI for interprocessor communication.

Our new GentenMPI implementation provides CP-ALS and GCP-SGD for distributed memory systems. It leverages Sandia's decades-long investment in the Trilinos solver framework [9, 8] for much of its parallel-computation capability. Trilinos contains numerical algorithms and linear algebra classes that have been optimized for parallel simulation of complex physical phenomena. Its Tpetra linear algebra package [3, 10] contains classes for distributed maps, vectors, multivectors, and sparse matrices; these building blocks are used as key kernels of GentenMPI's tensor decomposition.

In this report, we describe the implementation and performance of GentenMPI. We provide a brief introduction to Trilinos' linear algebra package Tpetra [3, 10]. We then detail GentenMPI's main classes, with their use of Tpetra and extensions needed for sparse tensor decomposition. We describe the implementation of a key kernel of CP-ALS and GCP-SGD: the MTTKRP (Matricized Tensor Times Khatri-Rao Product). We then present implementations and results of CP-ALS and GCP-SGD using GentenMPI. We show that GentenMPI can decompose sparse tensors of extreme size, e.g., a 12.6-terabyte tensor on 8192 computer cores. We demonstrate that the Trilinos backbone provides good strong and weak scaling of the tensor decomposition algorithms.

2. USING TRILINOS AND TPETRA

The Trilinos [8] framework is designed to provide linear, nonlinear, and eigen solvers, as well as discretization and load balancing tools, to parallel applications. The tools are designed to run on distributed memory parallel with multicore or GPU nodes. These components can be combined with physics descriptions to rapidly construct applications with minimal computer science effort needed by application developers.

Tpetra [3] is the key linear algebra package in Trilinos. Tpetra contains classes for vectors, multivectors and matrices, with operations performed in CPUs or GPUs. Tpetra exploits the Kokkos performance portability layer to enable computation across a variety of platforms. GentenMPI uses Tpetra to provide factor matrices and parallel distribution maps in tensor decomposition.

2.1. PARALLEL DISTRIBUTION AND TPETRA MAPS

Tpetra uses the `Map` class to describe the distribution of vectors and matrices to processors. Each entity (vector entry, matrix row, matrix column, etc.) has a unique global identifier (ID). The `Map` class describes the assignment of these IDs to processors. It also assigns a local identifier to each global ID on a processor; this local identifier can be used as an array index in local vector data.

The default distribution of IDs in a `Map` among processors is a linear partition of the IDs; for IDs $\{1,2,\ldots,J\}$ on P processors, the default `Map` would assign IDs $\{1,2,\ldots,J/P\}$ to processor 0, IDs $\{J/P+1,\ldots,2J/P\}$ to processor 1, and so on. Users can obtain other distributions, however, by specifying the number of IDs to give to each processor or by providing a list of specific IDs to assign to each processor.

A Tpetra sparse matrix (e.g., `Tpetra::CrsMatrix`) has four maps: a row map, a column map, a domain map, and a range map. The row map of a matrix contains the global indices of each row for which the processor stores at least one nonzero. The column map contains the global indices of each column for which the processor stores at least one nonzero. Neither row maps nor column maps need to be "one-to-one"; that is, many processor may store a given global ID in their row maps. The domain and range maps describe the distribution of the input vectors and output vectors, respectively, to be used in sparse matrix-vector multiplication. These maps are "one-to-one"; each entry is stored in only one processor's `Map`.

Figure 2-1 shows examples of the maps associated with a matrix A, input vector x and output vector y. In the left example, A is distributed in a row-wise manner, so that all nonzero entries (marked with x) in a row are assigned to a single processor. The blue processor has nonzeros in

rows 4 and 5; thus, entries 4 and 5 are in its row map. The nearly dense row 5 causes the blue processor's column map to have nearly all column indices. The distributions of x and y to processors are identical; thus, the range and domain maps are identical. The right example the same distributions of the input and output vectors, but uses a nonzero-based distribution of the tensor. The blue processor has nonzero entries in rows 4, 5, and 8; thus, these entries are in its row map. Note that rows 4 and 5 are also in the red processor's row map. The column map for the blue processor is smaller in the right figure, as the blue processor has nonzeros only in rows 4, 5, 6, and 7.

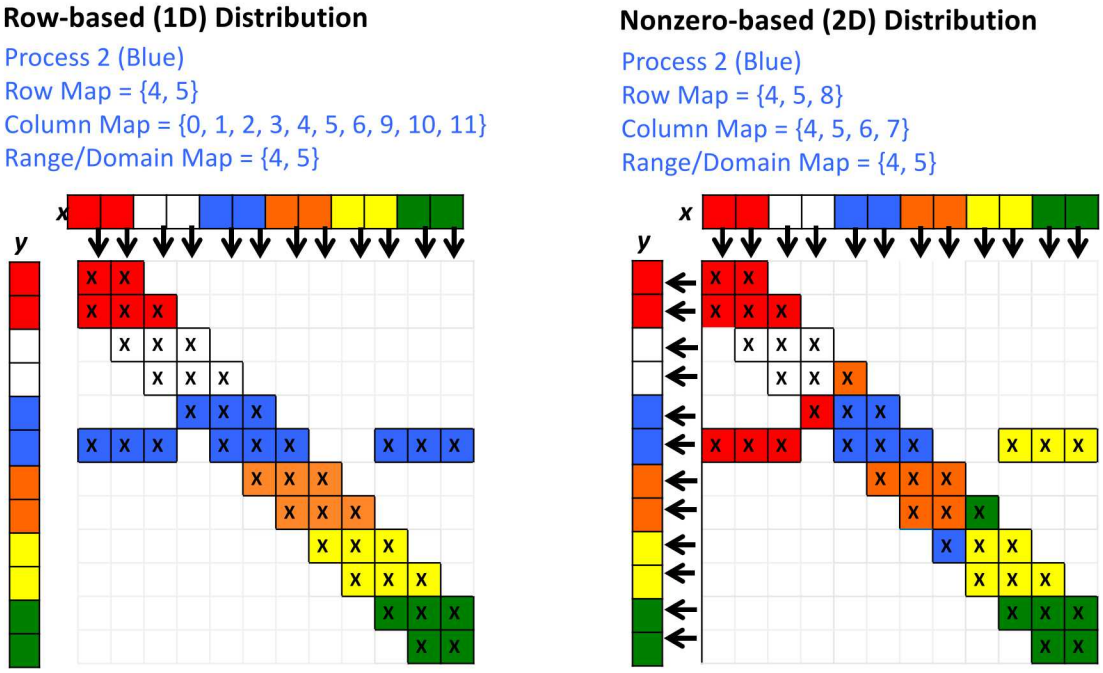

Figure 2-1. Examples of a distributed matrix and vectors in Trilinos. The left figure uses a row-based distribution; the right figure uses a nonzero-based distribution. Colors indicate processor assignment. The `Map` entries for the blue processor are listed.

2.2. TPETRA MULTIVECTORS

Tpetra's `MultiVector` class contains R vectors of length n, distributed to processors according to a Tpetra `Map`. In Figure 2-1, the input and output vectors x and y are `MultiVectors` of length $n = 12$ with $R = 1$ vector. They are distributed to six processors using "one-to-one" maps, so that each entry is uniquely assigned to a processors. `MultiVectors` may also have maps that are "overlapped" (not one-to-one) so that multiple processors have copies of the vectors entries. Such overlapped maps are used in sparse matrix-vector multiplication (SpMV), where several processors may contribute to a single output vector's entry. (See Chapter 4 for details.)

The `MultiVector` class provides operations such as dot products, random vector generation, vector normalization, scaling, and vector norms.

2.3. TRILINOS COMMUNICATION

Trilinos' Teuchos package provides wrappers around MPI Communicators. These wrappers allows Trilinos to be built with MPI for parallel execution, or without it for serial execution. They wrap the fundamental operations of MPI (send, receive, reduce, gather), and are used in all of Tpetra's distributed objects.

Tpetra provides the class `Import` to establish communication patterns between pairs of maps. For example, an `Import` object can be used to redistribute data from a source MultiVector using one map to a target Multivector using a different map. They can reverse the communication pattern as well, sending the data from the target object to the source. Data can be copied or accumulated into the target object; the latter allows data from several sources to be added into a single target entry. An `Import` object uses point-to-point communication in an underlying Tpetra `Distributor` class to perform communication.

3. GENTENMPI CLASSES

GentenMPI's tensor and factor matrix classes rely on the Tpetra classes described in Chapter 2, with additional structures needed to support higher-order tensor data.

3.1. SPARSE TENSOR

File: pt_sptensor.hpp

In GentenMPI's distributed sparse tensor class, each tensor nonzero is assigned and stored on a single processor. Tensor nonzeros are stored in coordinate format; that is, a nonzero is represented by its global tensor indices in each mode and its value. A processor's nonzeros' indices and values are stored in `Kokkos::View` data structures, analogous to 2D and 1D arrays, respectively.

For each tensor mode, the distributed tensor stores a Tpetra `Map`, listing the indices in the mode for which a processor has nonzeros. These maps are analogous to the row and column maps stored for Tpetra matrices (see Section 2.1, and most closely resemble the overlapped maps used for matrix distributions (as in Figure 2-1, right). For example, for a nonzero tensor entry x_{ijk} in $\mathcal{X}$, index i is in the mode-0 map, index j is in the mode-1 map, and index k is in the mode-2 map.

Sparse tensors may also have bounding box information specified by a distributed sparse tensor bounding box class (**File: pt_sptensor_boundingbox.hpp**). In the case the tensor is distributed using a "medium-grain" distribution [21], for example, all of the processor's nonzero entries fall within a Cartesian product of mode index ranges. These ranges can be stored as a bounding box, which is required for sampling strategies (see Section 3.6) within the GCP-SGD algorithm.

3.2. FACTOR MATRICES

File: pt_factormatrix.hpp

A factor matrix $A \in \mathbb{R}^{I \times R}$ is a Tpetra `MultiVector` of length I with R vectors (see Section 2.2). It exploits the MultiVector's methods for normalization, randomization, and norm calculations, as well as the MultiVector's map for its distribution. Like MultiVectors, factor matrices may be distributed uniquely (with one-to-one maps) or with copies (with overlapped maps).

By default, Tpetra stores the MultiVector data in column-major order (Kokkos::LayoutLeft). For many factor matrix operations, however, data is more efficiently accessed row-wise — that is, accessing all R entries for a given index i. Thus, GentenMPI modifies the Tpetra MultiVector to

use row-major storage (Kokkos::LayoutRight). Results showing the benefit of using row-major storage are in Chapter 4.

3.3. SQUARE LOCAL MATRICES

File: pt_squarelocalmatrix.hpp

A square local matrix $G \in \mathbb{R}^{R \times R}$ is stored redundantly on every processor as a `Kokkos::View` 2D data structure. This class is used for Gram matrices of factor matrices and the temporary matrices computed from them. Operations defined for square local matrices include Hadamard (elementwise) products with other square local matrices and with the outer product of a vector with itself and the sum of entries in the matrix (without absolute value). These operations are useful in forming linear systems within CP-ALS iterations (see Section 3.5) and in computing the norm of a Kruskal tensor (see Section 3.4), which itself is used in computing the 2-norm of the residual of a system (see Section 3.5). Square local matrices are nearly always symmetric, but this symmetry is not exploited in the implementation (computations involving these small matrices are rarely a bottleneck).

3.4. KRUSKAL TENSOR

File: pt_ktensor.hpp

The distributed Kruskal tensor (ktensor) class contains a factor matrix for each mode of the model $\mathcal{M}$, and an array λ of length R [1]. Factor matrices stored in the ktensor use one-to-one maps; each factor matrix entry is stored on only one processor. The λ array is stored redundantly on every processor.

3.5. SYSTEM

File: pt_system.hpp

Many operations in tensor decomposition require both a sparse tensor and a Kruskal tensor. GentenMPI's `distSystem` class couples a sparse tensor with a ktensor.

A distSystem's sparse tensor provides Tpetra maps analogous to the row and column maps of a Tpetra matrix. Its ktensor provides Tpetra maps analogous to the domain and range maps of a Tpetra matrix. The distSystem contains additional internal factor matrices for each mode, distributed according to the *sparse tensor's* maps. These factor matrices hold factor matrix entries corresponding to the stored nonzeros of the sparse tensor and typically have overlapped maps. The internal factor matrix entries are used, for example, to evaluate the model $\mathcal{M}$ at the indices of the sparse tensor. To update the internal factor matrices, the distSystem uses a Tpetra `Import`

object for each mode; the object contains the communication pattern necessary to transfer factor matrix entries from the ktensor's distribution to these internal factor matrices, and vice versa.

Operations requiring a sparse tensor and ktensor are also in the distSystem class. These operations include CP-ALS, GCP-SGD, MTTKRP, residual norm computation, loss function evaluation, and evaluation of the model $\mathcal{M}$.

3.6. SAMPLING STRATEGIES

File: pt_samplingstrategies.hpp

The GCP-SGD algorithm [17] can involve multiple sampling strategies of a sparse tensor (see section 6.1). SamplingStrategy is a base class with a derived class for each of three different strategies: stratified, semi-stratified, and full. All of the sampling strategies involve only local data, even for the distributed implementation. For all of the cases, the sampled entries are stored as a sparse tensor (see section 3.1) with both nonzero and (sampled) zero values stored explicitly. (Kolda [15] uses the term "scarce tensors" to refer to sparse tensors storing both nonzeros and zeros as we do here.)

The stratified strategy samples nonzeros and zeros separately. Nonzeros are sampled uniformly from the nonzeros in the original tensor, and zeros are sampled uniformly from within the full range of indices (in the distributed case, this range is determined by the bounding box of the sparse tensor, as described in section 3.1). In order to ensure that sampled zeros do not correspond to nonzero entries, each zero sample must be checked against the nonzero entries of the original tensor. This is implemented using a hash: all original nonzero entries are hashed with a `Kokkos::UnorderedMap`[1], and sampled zero indices are checked against the hash before being accepted as samples.

The semi-stratified strategy also samples nonzeros and zeros separately. Again, nonzeros are sampled uniformly from the nonzeros in the original tensor, and zeros are sampled uniformly from within the bounding box. In this case, zero samples are accepted whether or not they correspond to an original nonzero value; this possible inconsistency is accounted for within the GCP-SGD algorithm.

The full sampling strategy is used only for testing and debugging. It samples all nonzero values and all zero values of the original tensor and stores them in sparse format.

3.7. LOSS FUNCTIONS

File: pt_lossfns.hpp

[1] The TensorHash class (**File: pt_tensorhash.hpp**) wraps the `Kokkos::UnorderedMap` in order to use a variable number of indices (up to 6).

The Generalized CP (GCP) decomposition is defined for general loss functions. The loss function can be specified by a derived class of the base lossFunction class. The base class has three key operations: function evaluation, partial derivative evaluation, and model lower bound. For example, for the L^2 loss function (for Gaussian data), the loss function evaluation returns $f(x,m) = (x-m)^2$, the partial derivative evaluation returns $\frac{\partial f}{\partial m}(x,m) = 2(x-m)$, and the model lower bound is $-\infty$ (implemented as the lowest floating point number). The distributions with loss functions implemented are Gaussian, Poisson (-log), Bernoulli (odds and logit), Rayleigh, and Gamma.

4. MTTKRP

The Matricized Tensor Times Khatri-Rao Product (MTTKRP) operation is a key kernel of many tensor decomposition methods. In CP-ALS, for example, MTTKRP updates one factor matrix A of a Kruskal tensor using values from the other factor matrices B and C as follows:

$$a_{ir} = \sum_{jk \in \mathcal{X}} x_{ijk} b_{jr} c_{kr},\ i = 1,\ldots,I,\ r = 1,\ldots,R \tag{4.1}$$

In GCP-SGD, MTTKRP updates factor matrices of a gradient ktensor $\mathcal{G}$ in a similar manner.

4.1. PARALLEL MTTKRP VS. PARALLEL SPMV

The analogy between parallel MTTKRP and parallel sparse matrix-vector multiplication (SpMV) is strong. In the SpMV $(\vec{a}) = X(\vec{b})$, vector entries of the vector $(\vec{a})$ are updated using the values of $(\vec{b})$:

$$a_i = \sum_{j \in \mathcal{X}} x_{ij} b_j. \tag{4.2}$$

In parallel SpMV, input vector entries b_j must be communicated to processors having nonzeros in column j of the matrix; this communication is called an "expand" communication. The received values are multiplied by the processor's x_{ij}. The resulting products are then summed across matrix rows. All processors with nonzeros in row i must accumulate their partial sums into output vector entry a_i; this communication operation is called a "fold" communication. These expand and fold operations are illustrated in Figure 4-1.

Parallel MTTKRP requires the same expand and fold communications, as illustrated in Figure 4-2. In the expand communication, entries from factor matrices B and C are communicated to processors with corresponding tensor entries. Partial sums are then computed within the processor. The fold operation then communicates and accumulates the partial sums into the result factor matrix A.

4.2. IMPLEMENTATION

The MTTKRP implementation in GentenMPI works much like the implementation of SpMV in Tpetra. Here, we'll assume factor matrix A will receive the result of the MTTKRP computed from

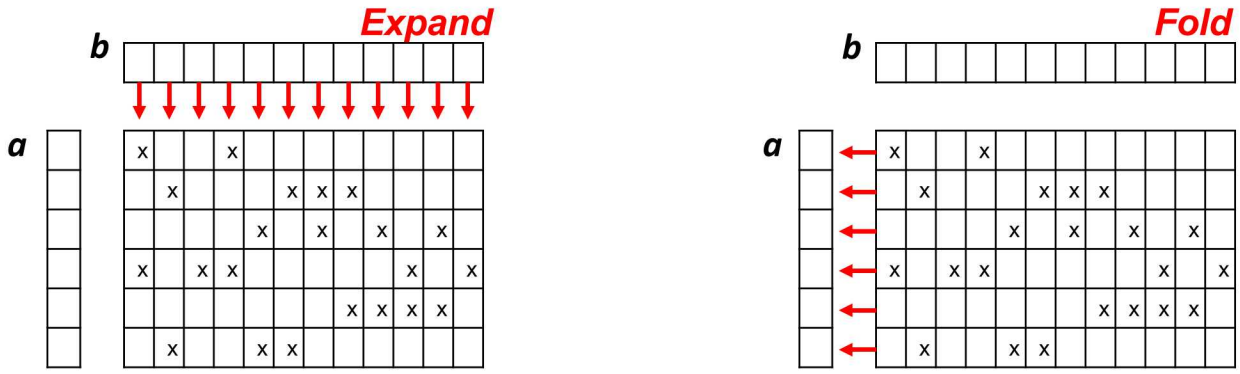

Figure 4-1. The expand and fold communication involved in SpMV. In the expand communication, the input vector entries b_j are communicated to processors with nonzeros in column j. Local products $x_{ij}b_j$ are computed. Then the fold communication accumulates partial sums across processors sharing matrix rows i into output vector entry a_i.

three-way tensor $\mathcal{X}$ and factor matrices B and C; this scenario is relevant to CP-ALS (Chapter 5).

1. MTTKRP requires a `distSystem` object D constructed from the tensor $\mathcal{X}$ and a Kruskal tensor with the factor matrices A, B, and C. During construction of the D, its internal factor matrices $\hat{A}$, $\hat{B}$, and $\hat{C}$ are updated via communication using the `Import` objects in each mode. This update constitutes the "expand" communication, and brings to each processor the factor matrix entries in each mode corresponding to the indices of the processor's x_{ijk} entries.

2. Local products are computed within each processor. The processor loops over its owned tensor entries x_{ijk} and multiplies them by the associated entries $\hat{b}_{jr}$ and $\hat{c}_{kr}$, accumulating the results in $\hat{a}_{ir}$. This operation is a triply nested loop, first over the N_p tensor entries on a processor, then over tensor modes, and then over the rank R.

3. After completion of the local computation, D's `Import` object associated with A communicates values from $\hat{A}$ back to A, with contributions from multiple processors summed into A. This operation is the "fold" communication.

4. Once A is updated by all processors, the accrued values of A can be communicated back to $\hat{A}$ to keep the internal factor matrices up-to-date with actual factor matrix values.

For GCP-SGD (Chapter 6), the result of the MTTKRP with B and C is not stored in A, but rather, is stored in an additional factor matrix G. In this case, additional storage $\hat{G}$ is used to receive the result of the MTTKRP in step 2, and $\hat{G}$ is communicated to G in step 3. Step 4 is not needed since neither $\hat{A}$ nor A is changed in the MTTKRP.

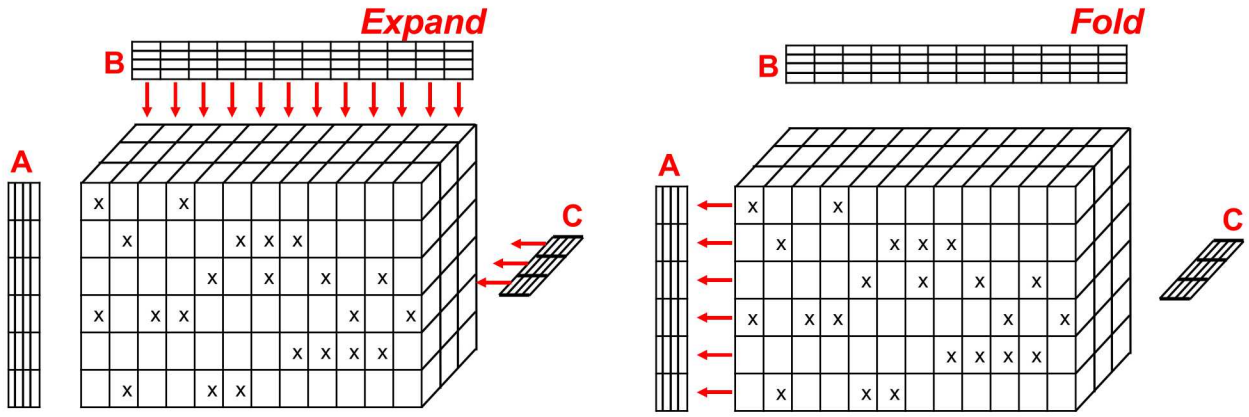

Figure 4-2. The expand and fold communication involved in MTTKRP. In the expand communication, the entries input factor matrices B and C are communicated to processors with corresponding entries in the tensor $\mathcal{X}$. Local products $x_{ijk}b_{jr}c_{kr}$ are computed. Then the fold communication accumulates partial sums across processors into output factor matrix A.

4.3. COLUMN-MAJOR VS ROW-MAJOR

The default layout of data in Tpetra's `MultiVector` class is column-major (`Kokkos::LayoutLeft`); that is, for an $I \times R$ factor matrix, the first vector of length I is stored, followed by the second vector of length I, and so on. This layout is convenient for some linear solvers that need to access a single vector or a subset of vectors of a given multivector. However, in MTTKRP, all R factor matrix entries for a given index i are accessed together in step 2 above. The default layout causes strided memory acceses that can lead to poor cache performance.

A better layout for MTTKRP is row-major (`Kokkos::LayoutRight`), in which all R entries for a given index i are stored contiguously in memory. Thus, GentenMPI uses a modified Tpetra `MultiVector` that uses row-major storage; these modifications were trivial and did not interfere with Tpetra's communication of multivector data values.

Figure 4-3 shows the difference in performance using column-major vs row-major layouts. This example was run on one processor of Sandia's SkyBridge cluster with 2.6GHz Intel Sandy Bridge processors. It uses the *delicious-4d* tensor from the FROSTT [20] tensor collection, a $532,924 \times 17,262,471 \times 2,480,308 \times 1443$ tensor with 140 million nonzeros. With rank $R = 8$, the difference in MTTKRP time between column- and row-major layouts is visible; column-major layout requires 1.7 times more execution time per MTTKRP. With larger values of R, the difference becomes even more significant, with column-major layout taking 3.4 times longer than row-major layout for $R = 32$.

For all experiments in the remainder of this report, GentenMPI uses row-major (`Kokkos::LayoutRight`) layout for all factor matrices.

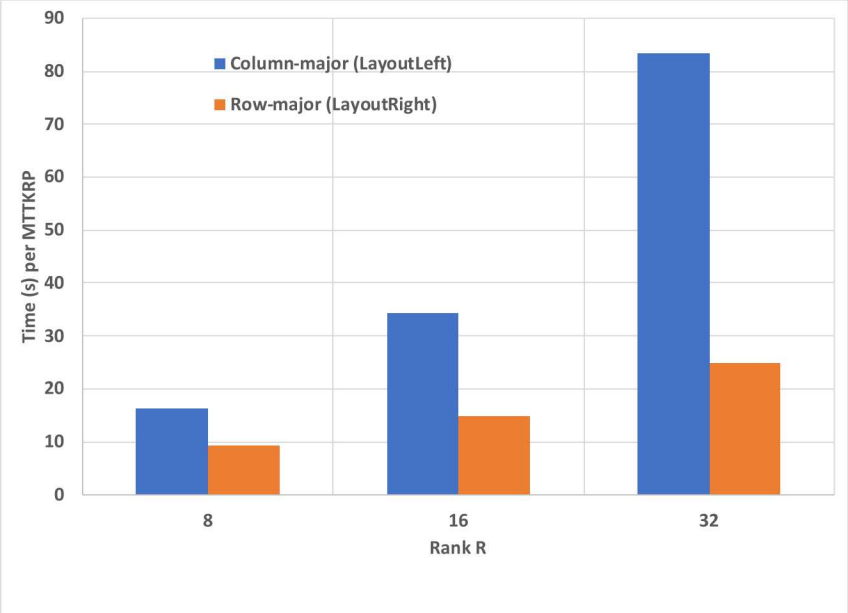

Figure 4-3. MTTKRP times using column-major (LayoutLeft) and row-major (LayoutRight) layout of the factor matrices. (*delicious-4d* tensor, rank $R = 8, 16, 32$)

5. CP-ALS

The Canonical Polyadic decomposition (CPD) [4, 7] is a tensor decomposition that uses an L^2 loss function $f(x,m) \equiv (x-m)^2$ in the optimization in Equation 1.2. CP-ALS — CP solved via alternating least squares optimization — is one approach for performing this tensor decomposition.

5.1. ALGORITHM

CP-ALS uses an alternating least squares approach to perform the optimization. A sketch of the algorithm for three-way tensors is included in Algorithm 1. The method and GentenMPI implementation extend to tensors of any order; for more details, see Kolda and Bader's survey [16]. First, factor matrix A is computed with fixed factor matrices B and C; then B is updated using A and C; and then C is updated using A and B. The algorithm iterates over the updates until a desired convergence tolerance is reached or the specified maximum numer of iterations is exceeded.

Algorithm 1 CP-ALS algorithm for a three-mode tensor $\mathcal{X}$

1: **procedure** CP-ALS($\mathcal{X}, [\lambda; A, B, C]$)
2: **repeat**
3: $A \leftarrow$ MTTKRP($\mathcal{X}, B, C$) $(C^T C * B^T B)^{-1}$
4: Normalize columns of A
5: $B \leftarrow$ MTTKRP($\mathcal{X}, C, A$) $(A^T A * C^T C)^{-1}$
6: Normalize columns of B
7: $C \leftarrow$ MTTKRP($\mathcal{X}, A, B$) $(B^T B * A^T A)^{-1}$
8: Normalize columns of C; store norms as λ
9: **until** converged or max iterations reached
10: **return** $[\lambda; A, B, C]$

5.2. IMPLEMENTATION

In GentenMPI, CP-ALS is implemented using the MTTKRP algorithm described in Chapter 4. The $R \times R$ Gram matrices $(C^T C * B^T B)$ are small enough to replicate on every processor; contributions from each processor are accumulated using an MPI_Allreduce operation. The linear

systems involving these matrices are solved using LAPACK's GESV method. Convergence is checked by computing the L^2-norm of $\mathcal{X} - \mathcal{M}$.

While GentenMPI's CP-ALS can function with any distribution of the tensor and factor matrices, faster performance and better scaling is achieved when the number of tensor entries per processor is balanced and tensor indices are localized to reduce the number of factor matrix entries needed in MTTKRP. The SPLATT tensor code provides a "medium-grain" decomposition in which the tensor is divided into subtensors with roughly uniform number of nonzeros per subtensor [21]. In each mode, then, expand and fold communication is done among processors within a slice of subtensors. A illustration of a medium-grain decomposition is in Figure 5-1. For a three-way tensor, $P = 48$ processors are organized into a three-way $6 \times 4 \times 2$ grid. Cuts in each mode then greedily assign tensor slices to processors in a way that balances the number of nonzero entries.

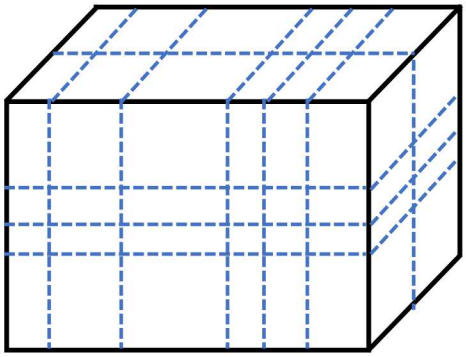

Figure 5-1. Illustration of a SPLATT medium-grain distribution [21] of a three-way tensor to 48 processors

We adopt this medium-grain decomposition for CP-ALS in GentenMPI. Because equal number of nonzers are assigned to each processor, this decomposition provides good load balance in the MTTKRP computation. While it doesn't explicitly attempt to minimize communication during MTTKRP, it provides reasonable alignment between factor matrix distributions and needed tensor entries. And it is inexpensive to compute compared to partitions that do explicitly attempt to reduce communication, such as the hypergraph methods of Kaya and Ucar [13].

We use the default Trilinos `Map` layout (Section 2.1) for factor matrices; that is, on P processors, factor matrix $A \in \mathbb{R}^{I \times R}$ is divided into P chunks of length I/P, with processor 0 receiving chunk $\{1, 2, \ldots, I/P\}$, processor 1 receiving chunk $\{I/P + 1, \ldots, 2I/P\}$, and so on.

5.3. EXPERIMENTAL RESULTS

The main motivation in creating GentenMPI is to enable decomposition of tensors too large to fit into a single node's memory. To demonstrate this capability, we study the weak-scaling of GentenMPI's CP-ALS by generating random sparse tensors and apply CP-ALS to them. The generated tensors are four-way tensors with 64 million nonzeros per processor and the mode lengths adjusted to maintain constant nonzero density of 0.001024. With these characteristics, the

tensor size is 12.6 Terabytes on 8192 processors: 524 billion nonzeros with four integers and one double per nonzero.

Weak scaling results on Sandia's SkyBridge cluster (2.6 GHz Intel Sandy Bridge nodes with Infiniband network) are shown in Figure 5-2. We show both the average time per CP-ALS iteration and MTTKRP within a CP-ALS iteration. Clearly, the MTTKRP kernel dominates the CP-ALS computation. Weak scaling is very good, but degrades slightly due to an increased number of neighboring processors (and, thus, of messages) as the number of processors increases. (Envision more layers of processors being added to the processor distribution in Figure 5-1 as the number of processors increases.)

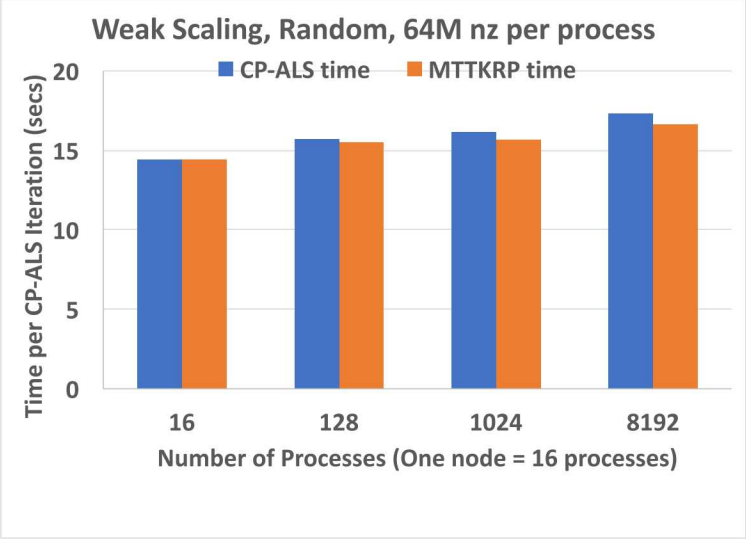

Figure 5-2. Weak scaling of CP-ALS on a four-way random tensor: the time for one CP-ALS iteration is in blue, with the MTTKRP time per iteration in orange. The largest tensor, decomposed on 8192 processors, is 12.6 Terabytes.

We next examine the strong scaling of GentenMPI's CP-ALS implementation, comparing GentenMPI's performance with SPLATT [21] and Genten [19]. SPLATT can run with distributed memory parallelism only ("MPI-only") or with hybrid distributed memory and shared memory threading ("MPI+OpenMP"). We compare with both configurations. For the MPI-only case, we use the same number of MPI ranks for SPLATT and GentenMPI. For MPI+OpenMP, we use 16 threads per MPI rank and one MPI rank in SPLATT for every 16 MPI ranks in GentenMPI. For 16 processor runs, we compare with Genten with 16 threads.

We begin with the *delicious-4d* tensor from the FROSTT [20] tensor collection, a $532,924 \times 17,262,471 \times 2,480,308 \times 1443$ tensor with 140 million nonzeros. We run with 16 to 1024 cores, with rank R ranging from 8 to 128. Times per CP-ALS iteration are shown in Figure 5-3. We see that the strong scaling of GentenMPI is good in all experiments. Runtimes are

generally faster than SPLATT MPI-only; multithreading does make SPLATT's performance with MPI+OpenMP superior to GentenMPI. GentenMPI's runtimes are acceptable when compared to Genten. GentenMPI has the benefit that it can access sufficient memory for the $R = 128$ case, while Genten has the advantage that it can run on both multithreaded CPUs and GPUs.

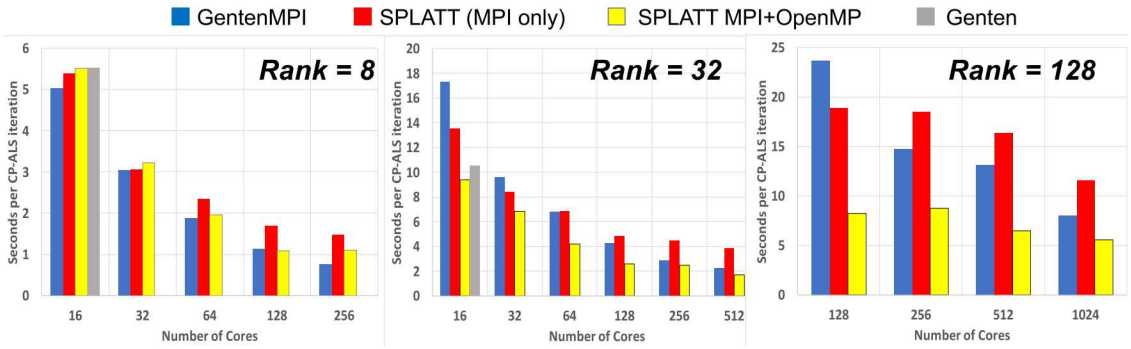

Figure 5-3. Strong scaling of CP-ALS on the *delicious-4d* tensor from the FROSTT [20] collection. GentenMPI times per CP-ALS iteration are compared with those from Genten [19] and SPLATT [21] with MPI-only and MPI+OpenMP.

We demonstrate GentenMPI on the larger *amazon-reviews* tensor from FROSTT, a $4.8M \times 1.8M \times 1.8M$ tensor with 1.7 billion nonzeros. This tensor is too large to fit in a single node of SkyBridge, so comparisons with Genten are not possible. For this tensor, we see in Figure 5-4 that GentenMPI's implementation is faster than both SPLATT MPI-only and SPLATT MPI+OpenMP. Strong scaling is good out to 1024 cores.

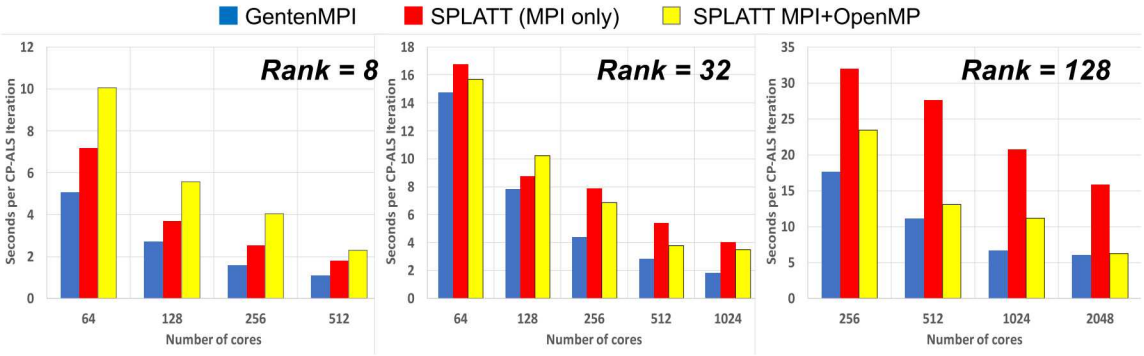

Figure 5-4. Strong scaling of CP-ALS on the *amazon-reviews* tensor from the FROSTT [20] collection. GentenMPI times per CP-ALS iteration are compared with those from SPLATT [21] with MPI-only and MPI+OpenMP.

6. GCP-SGD

Kolda and Hong [17] propose using a stochastic gradient descent method to solve the Generalized Canonical Polyadic (CP) optimization of Equation 1.2. This algorithm relies on samples of the full sparse tensor to inexpensively estimate the loss function F and its gradient with respect to the model.

6.1. DISTRIBUTED-MEMORY SAMPLING

Uniform sampling of a large sparse tensor would likely select mostly zero valued entries of the tensor, since the number of nonzeros in a sparse tensor is much smaller than the number of zeros. To ensure that a sufficient number of nonzeros of the sparse tensor are selected in each sample, Kolda and Hong present two sampling strategies: stratified and semi-stratified. Stratified sampling samples p nonzeros and q zeros of $\mathcal{X}$ separately, allowing sufficient numbers of nonzeros to be selected. Semi-stratified sampling samples p nonzeros and q tensor indices separately; tensor indices include both nonzeros and zeros. A correction to the partial derivative computation accounts for the possibility that a sampled tensor index may actually be a nonzero. All sampling is done "with replacement"; that is, a nonzero or zero may be selected and stored more than once.

Distributed-memory parallelism introduces some challenges to effective sampling. Sampling nonzeros is straightforward; each processor z samples some number p_z of its locally stored nonzeros. Sampling zeros is more challenging because, for sparse tensors, only nonzeros are explicitly stored. In theory, each processor z could simply sample q_z indices from the index space of the entire tensor. In practice, however, this approach has severe parallel performance problems. Stratified sampling of zeros, for example, requires a check for each selected index to confirm that it is actually a zero, not a nonzero. In parallel, this check would require all-to-all communication of all sampled indices to determine whether they are nonzeros on any processor. Even in semi-stratified sampling, which doesn't require a check to confirm that sampled indices are truly zeros, sampling from the entire tensor can cause performance problems, as the resulting set of indices can require factor matrix entries from all processors for model and MTTKRP computation. Restricting the domain from which each processor samples zeros can alleviate both problems.

While the CP-ALS algorithm in GentenMPI can operate with an arbitrary distribution of the sparse tensor, for GCP-SGD, we restrict the distribution so that each processor owns a unique "bounding box" of the tensor's index space. Processors' bounding boxes may not overlap, and they must cover the entire index space of the tensor. Thus, each processor is responsible for both the nonzeros and zeros of a subtensor within the tensor.

Two bounding box options are available in GentenMPI. The first arises from the SPLATT medium-grain distribution [21] from Figure 5-1. In this distribution, every processor has an equal number of nonzeros, but an unequal number of zeros. To sample p nonzeros uniformly across P processors, p_z is chosen to be p/P on every processor z. For load balancing, then, one would want every processor to use the same value of q_z as well, giving an equal number of indices $p_z + q_z$ on each processor. However, because bounding boxes are not uniformly sized with a medium-grain distribution, achieving load balance causes the space of zeros to be sampled nonuniformly, with the density of sampled zeros being higher in small bounding boxes than in large ones. Conversely, scaling q_z to the size of the bounding box results in imbalanced numbers of sampled indices across processors.

An alternative is to use uniformly sized bounding boxes for each processor, as in Figure 6-1. This nearly trivial distribution of the tensor's index space results in processors having unequal numbers of nonzeros and zeros, but equal number of indices overall. Thus, to achieve load balance, each processor z selects $s_z = (p+q)/P$ samples. On processor z with n_z nonzeros out of n nonzeros in the tensor, the number of nonzeros samples p_z is $p(n_z/n)$; the number of zero samples q_z is $s_z - p_z$. In this way, the numbers of indices per processor in the sampled tensor are balanced, and sampling of both zeros and nonzeros is done uniformly across the entire tensor. As a practical consideration, limits are imposed to ensure that p_z is at least $0.1(s_z)$ and no more than $0.9(s_z)$; these limits ensure that both nonzeros and zeros are sampled even when the distribution of nonzeros to boxes is very imbalanced (e.g., when $p(n_z/n) > s_z$).

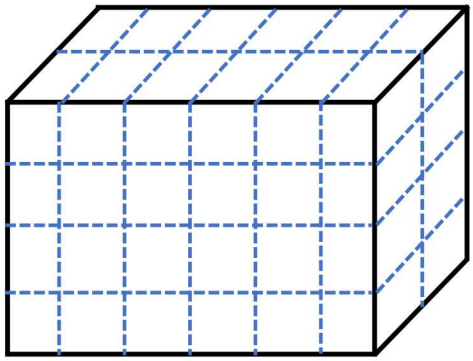

Figure 6-1. Illustration of a uniformly sized box distribution of a three-way tensor to 48 processors

When using a sampled tensor to estimate the loss function, Kolda and Hong [17] scale the contributions of selected zeros and nonzeros by a factor that amplifies the contribution based on sample size. For stratified sampling, they define the estimated loss function $\hat{F}$ (Equation 5.2 of [17]) as

$$\hat{F} \equiv \sum_{x_{ijk} \neq 0} \frac{n}{p} f(x_{ijk}, m_{ijk}) + \sum_{x_{ijk}=0} \frac{(N-n)}{q} f(0, m_{ijk})$$

where the summations are over sampled tensor entries only, N is the total number indices in the tensor (i.e., the product of the tensor dimensions), n is the number of tensor nonzeros, p is the

number of nonzeros samples, and q is the number of zero samples. In distributed memory with local sampling, we locally scale the contributions of each index before summing over all processors:

$$\hat{F} \equiv \sum_{\text{all processors } z} \sum_{x_{ijk}\neq 0} \frac{n_z}{p_z} f(x_{ijk}, m_{ijk}) + \sum_{x_{ijk}=0} \frac{(N_z - n_z)}{q_z} f(0, m_{ijk}) \tag{6.1}$$

where the inner summations are over sampled entries, and, on processor z, N_z is the number of indices in the bounding box (i.e., the product of the bounding box dimensions), n_z is the number of nonzeros, and p_z and q_z are the numbers of nonzero and zero samples, respectively.

6.2. ALGORITHM AND PARALLEL IMPLEMENTATION

To solve the optimization problem in Equation 1.2, Kolda and Hong [17] adopt the Adam optimization algorithm [14] outlined in Algorithm 2. Here, we follow [17] and write the model $\mathcal{M} = [\lambda; \{A_k\}]$, where $\{A_k\}$ is the set of factor matrices making up the model. (In our prior three-way examples, $\{A_k\} = \{A, B, C\}$.) Algorithm parameters α, β_1, β_2, ε, τ, and ν are user-specified parameters controlling the learning rate of Adam; we use the default values from TensorToolbox [2]: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e-8$, $\tau = 1000$, and $\nu = 0.1$. Parameter ℓ is a lower bound of reasonable solution values (e.g., 0 for non-negative tensors).

Temporary factor matrices $\{T_k\}$ and $\{U_k\}$ are created using the same parallel distribution (i.e., the same Tpetra `Maps`) as $\{A_k\}$. Thus, the randomization and initialization in lines 2-3 can be done locally on each processor's portion of the factor matrices with no communication. Likewise, the element-wise factor-matrix operations in lines 14-19 can be performed locally; no communication is needed.

The sparse tensor $\hat{\mathcal{X}}$ is used to estimate error during GCP-SGD. It is constructed via stratified sampling using a fixed set of sampled indices. Its creation in line 4 requires communication to create its maps and import objects relative to $\{A_k\}$. The loss function computation in line 5 requires "expand" communication to send the entries of $\{A_k\}$ corresponding to indices of $\hat{\mathcal{X}}$ as described Chapter 4; since $\hat{\mathcal{X}}$ contains zero indices that were not in $\mathcal{X}$, processors need different factor matrix entries than they needed for $\mathcal{X}$. Similarly, expand communication is needed for line 20 as the entries of $\{A_k\}$ were modified by the loop above.

The stochastic gradient computation is by far the most expensive part of the GCP-Adam computation. Algorithm 3 provides a high-level overview of our parallel implementation; mathematical details for stratified and semi-stratified computation are found in [17], Algorithms 4.2 and 4.3, respectively.

In line 2 of Algorithm 3, a sampled tensor $\mathcal{Y}$ with s samples is created via stratified or semi-stratified sampling. The sampling itself is local to each processor, but creation of the sampled tensor requires communication to construct its maps in each mode. In line 3, we build a `distSystem` class to create the communication pattern (import objects) between the $\mathcal{Y}$ and $\{A_k\}$ and expand values from $\{A_k\}$ to processors that need them. Again, $\mathcal{Y}$ may have a different set of zeros from $\mathcal{X}$ and $\hat{\mathcal{X}}$, so different maps and import objects are needed for $\mathcal{Y}$. In line 4, the

Algorithm 2 GCP-Adam

function GCP-ADAM($\mathcal{X}$, $\mathcal{M} = [\lambda; \{A_k\}]$, s, α, β_1, β_2, ε, τ, ν, ℓ)
  Randomly initialize $\{A_k\}$
  $\{T_k\} \leftarrow 0$; $\{U_k\} \leftarrow 0$ ▷ temporary factor matrices for $\{A_k\}$
  $\hat{\mathcal{X}} \leftarrow$ sparse tensor stratified-sampled from $\mathcal{X}$
  $\hat{F} \leftarrow \text{ESTOBJ}(\hat{\mathcal{X}}, \{A_k\})$ ▷ estimate loss with fixed set of samples
  $t \leftarrow 0$ ▷ $t =$ # of Adam iterations
  while max number of bad epochs not exceded **do**
    Save copies of $\{A_k\}$, $\{T_k\}$ and $\{U_k\}$ ▷ save in case of failed epoch
    $\hat{F}_{\text{old}} \leftarrow \hat{F}$ ▷ save to check for failed epoch
    for τ iterations **do** ▷ $\tau =$ # iterations per epoch
      $\{G_k\} \leftarrow \text{STOCGRAD}(\mathcal{X}, \{A_k\}, s)$ ▷ $s =$ # samples per stochastic gradient
      for $k = 1, |\{A_k\}|$ **do**
        $t \leftarrow t + 1$
        $T_k \leftarrow \beta_1 T_k + (1 - \beta_1) G_k$
        $U_k \leftarrow \beta_2 U_k + (1 - \beta_2) G_k^2$
        $\hat{T}_k \leftarrow T_k / (1 - \beta_1^t)$
        $\hat{U}_k \leftarrow U_k / (1 - \beta_2^t)$
        $A_k \leftarrow A_k - \alpha \cdot (\hat{T}_k \oslash (\sqrt{\hat{U}_k} + \varepsilon))$
        $A_k \leftarrow \max\{A_k, \ell\}$ ▷ $\ell =$ lower bound

    // Adam update depends on β_1, β_2, ε; $\alpha =$ learning rate

    $\hat{F} \leftarrow \text{ESTOBJ}(\hat{\mathcal{X}}, \{A_k\})$ ▷ estimate loss with fixed set of samples
    if $\hat{F} > \hat{F}_{\text{old}}$ **then** ▷ check for failure to decrease loss
      Restore saved copied of $\{A_k\}$, $\{T_k\}$, $\{U_k\}$ ▷ revert to last epoch's variables
      $\hat{F} \leftarrow \hat{F}_{\text{old}}$ ▷ revert to prior function value
      $t \leftarrow t - \tau$ ▷ wind back the iteration counter
      $\alpha \leftarrow \alpha \cdot \nu$ ▷ reduce the step length
  return $\{\mathbf{A}_k\}$

sampled values in $\mathcal{Y}$ are overwritten by the element-wise partial gradient tensor such that $y_{ijk} = \frac{\delta f}{\delta m}(x_{ijk}, m_{ijk})$. Then MTTKRP operations (line 6) are used to compute the returned values $\{G_k\}$. Only "fold" communication of $\{G_k\}$ is needed during MTTKRP as the values of $\{A_k\}$ were communicated (expanded) during system construction and do not change in the MTTKRP.

Stochastic gradient implementations in TensorToolbox and Genten fuse sampling with the partial derivative computation. They do not form a sampled tensor but, rather, sample an index and immediately compute the element-wise derivative at the index, storing it in $\mathcal{Y}$. They can do this fusion because they operate in a single memory space and have all factor matrix entries available for use. Since GentenMPI operates in distributed memory, it does not have all factor matrix entries associated with a given sampled index within a processor; those entries must be communicated. We construct the sampled tensor, then, so that the communication can be done in one round rather than for each sampled index.

Algorithm 3 StocGrad

1: **function** STOCGRAD($\mathcal{X}$, $\{A_k\}$, s)
2: Sample s indices and construct sparse tensor $\mathcal{Y}$
3: Build `distSystem` object from $\mathcal{Y}$ and $\{A_k\}$
4: $\mathcal{Y} \leftarrow$ element-wise $\frac{\delta f}{\delta m}$
5: **for** $A_m \in \{A_k\}$ **do**
6: $G_m \leftarrow$ MTTKRP($\mathcal{Y}$, $\{A_k\} \backslash A_m$)
7: return $\{G_k\}$

6.3. EXPERIMENTAL RESULTS

6.3.1. Convergence Results

In this section we present convergence results of GCP-SGD for three different data sets, each using a Poisson loss function and the semi-stratified sampling strategy. Figures 6-2 to 6-4 plot the values of the loss function over time. The first two datasets are small enough to run on a single node, and the first experiment is designed to be compared against an existing result using a MATLAB implementation of the GCP-SGD algorithm [17]. The third data set is too large for GCP-SGD to execute on a single node, and the experiment is performed using 16 nodes.

Figure 6-2 shows the convergence results for GCP-SGD on the *Chicago crime* data set from the FROSTT [20] collection using Poisson loss. The Chicago crime data tensor is $6186 \times 24 \times 77 \times 32$ with 5.3 million nonzeros. This result can be compared against [17, Figure 5.6], where the same GCP-SGD algorithm is used (with similar parameter settings) in a MATLAB environment. In both cases, the Poisson loss converges to approximately `2.05e7`, though the time required is less for the parallel results. The experiment was performed on a single node of Skybridge, with one MPI process for each of the 16 cores. The average time per iteration is 1.24 seconds.

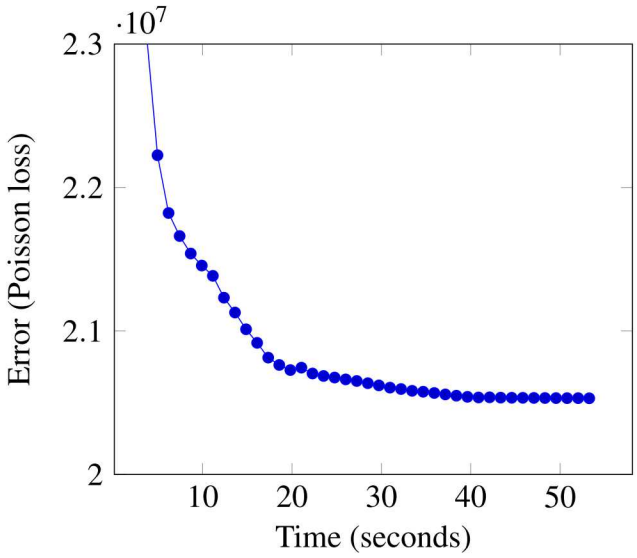

Figure 6-2. GCP-SGD convergence results for Chicago crime data, using Poisson loss, rank $R = 10$, and semi-stratified gradient sampling. Each of the 92 markers corresponds to an epoch, each epoch corresponds to 1000 iterations, and each iteration used a total of 3152 nonzero samples and 3152 index (zero) samples. The error is estimated using stratified objective function sampling with 31,588 nonzero samples and 31,596 zero samples.

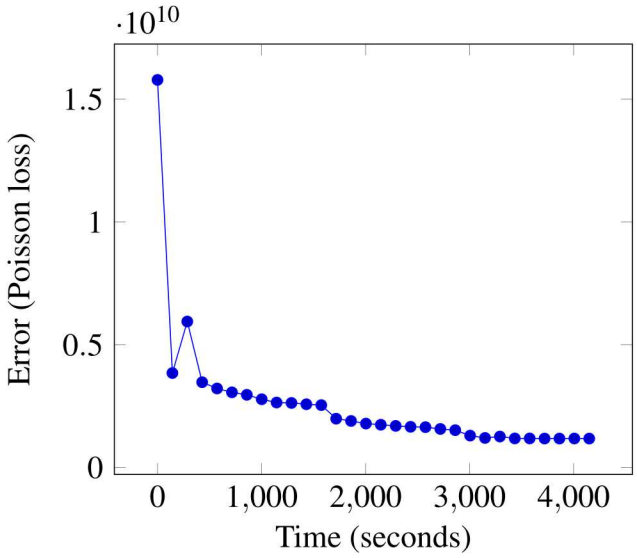

Figure 6-3. GCP-SGD convergence results for LBNL network data, using Poisson loss, rank $R = 10$, and semi-stratified gradient sampling. Each of the 30 markers corresponds to an epoch, each epoch corresponds to 1000 iterations, and each iteration used a total of 439,879 nonzero samples and 439,881 index (zero) samples. The error is estimated using stratified objective function sampling with 4,398,859 nonzero samples and 4,398,869 zero samples.

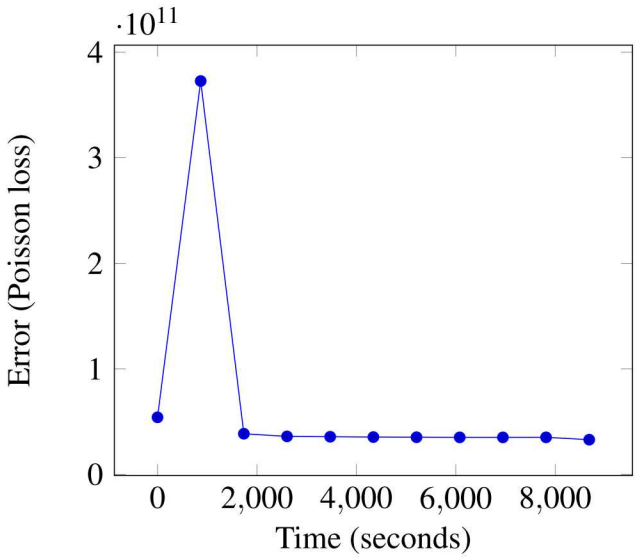

Figure 6-4. GCP-SGD convergence results for *amazon-reviews* data, using Poisson loss, rank $R = 16$, and semi-stratified gradient sampling. Each of the 11 markers corresponds to an epoch, each epoch corresponds to 1000 iterations, and each iteration used a total of 4,200,196 nonzero samples and 4,200,444 index (zero) samples. The error is estimated using stratified objective function sampling with 42,003,186 nonzero samples and 42,003,214 zero samples.

Figure 6-3 shows the convergence results for GCP-SGD on the *LBNL network* data set from the FROSTT [20] collection using Poisson loss. The *LBNL network* tensor is $1605 \times 4198 \times 1631 \times 4209 \times 868131$ with 1.7 million nonzeros. The experiment was performed on a single node of Skybridge, with one MPI process for each of the 16 cores. The average time per iteration is 143 seconds.

Compared to the *Chicago crime* data set, the *LBNL network* data set takes over 100 times longer per iteration. This is because the number of entries in the model is proportional to the sum of the tensor dimensions, and the number of samples used is chosen to be proportional to the number of entries in the model. That is, the number of samples used for *LBNL network* is about 100 times the number used for *Chicago crime*, which helps to explain the increase in time.

Figure 6-4 shows the convergence results for GCP-SGD on the *amazon-reviews* data set from the FROSTT [20] collection using Poisson loss. The experiment was performed on 16 nodes of Skybridge, with one MPI process per core, for a total of 256 MPI processes. The average time per iteration is 869 seconds.

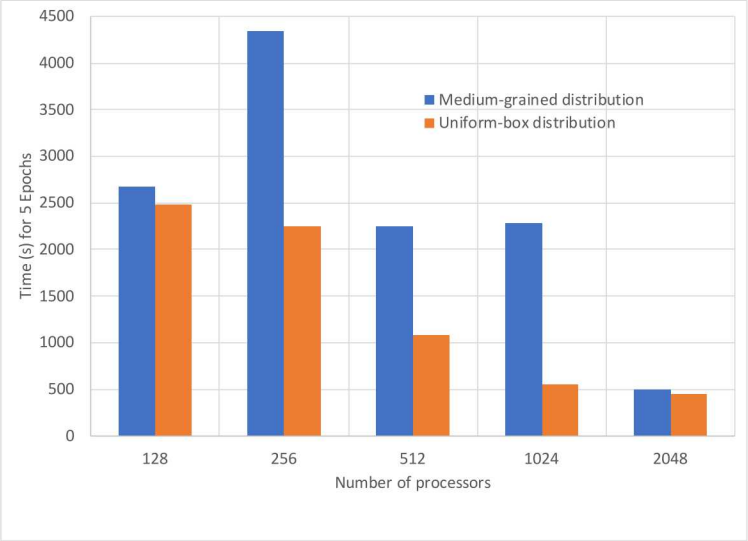

Figure 6-5. Comparison of runtimes using the *amazon-reviews* tensor using L^2 loss, rank $R = 16$, and semi-stratified sampling using medium-grained partitioning versus a uniform box distribution. Each of the five epochs ran 1000 iterations. Each iteration used semi-stratified sampling with approximately 4.2M nonzeros and 4.2M zeros. Error estimation used stratified sampling with approximately 42M nonzeros and 42M zeros.

6.3.2. Parallel Distribution for Sampling

We compare the performance of Algorithm 2 using the medium-grained distribution and uniform-box distributions described in Section 6.1. For our experiments, we use the *amazon-reviews* tensor from the FROSTT [20] collection. The Amazon data tensor is $4821207 \times 1774269 \times 1805187$ with 1.7 billion nonzeros. In Figure 6-5, we show execution times for five epochs with 1000 iterations each on the Skybridge cluster with 128 to 2048 processors. In all cases, the uniform-box distribution resulted in lower execution time and more consistent scaling performance. With the uniform-box distribution, less time was needed for performing MTTKRP and building maps between the sampled tensor and factor matrices.

Given the benefit of uniform-box distribution, we use it in all subsequent parallel experiments.

6.3.3. Strong Scaling and Timing Breakdown

Having confirmed the convergence behavior of GentenMPI, we now consider its parallel performance and strong scaling. Figures 6-6 to 6-8 demonstrate the strong scaling behavior for

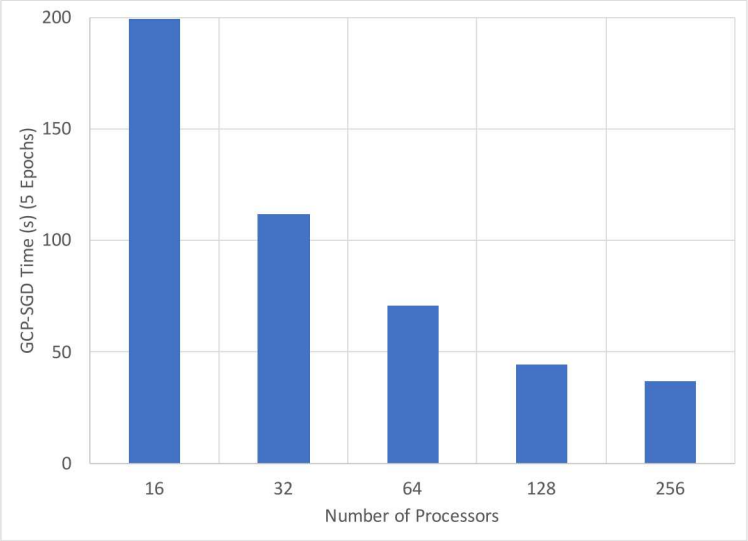

Figure 6-6. Strong scaling for *LBNL network* tensor using L^2 loss, rank $R = 16$, and semi-stratified sampling. Each of the five epochs ran 1000 iterations. Each iteration used semi-stratified sampling with approximately 10K nonzeros and 10K zeros. Error estimation used stratified sampling with approximately 100K nonzeros and 100K zeros.

LBNL network, random, and *amazon-reviews* data sets. All experiments were performed on Skybridge, and all used L^2 loss and the semi-stratified sampling strategy.

The *LBNL network* data set has order five, with about 1.7 million nonzeros. For the experimental results in Figure 6-6, we use approximately 200,000 samples to estimate the error and 20,000 samples in the stochastic gradient tensor. On 16 processors (1 node of Skybridge), the time for 5 epochs is 192 seconds. For comparison, Genten [19] takes between 101 and 120 seconds, depending on the MTTKRP implementation used. The strong scaling is reasonable up to 128 processors, achieving over a $4\times$ speedup, but there is little reduction from 128 to 256 processors. At 256 processors, the average number of original tensor nonzeros per processor is quite small — less than 10,000 — and the number of samples per processor is less than 100.

Figure 6-7 shows experimental results for a random tensor, which allows for perfect load balance of nonzeros, even in the case of uniform boxes. The random tensor is $1000 \times 1000 \times 500 \times 500$ with 256 million nonzeros. We use approximately 5 million samples to estimate the error and 1.5 million samples in each stochastic gradient. Using 16 processors (1 node), GentenMPI took 373 seconds; Genten [19] with 16 threads took between 306 and 1250 seconds, depending on the MTTKRP implementation used. From one MPI rank to 64 ranks, GentenMPI exhibited a $52.7\times$ speed-up.

From the time breakdown, we see that the dominant kernels in this experiment (using up to 64

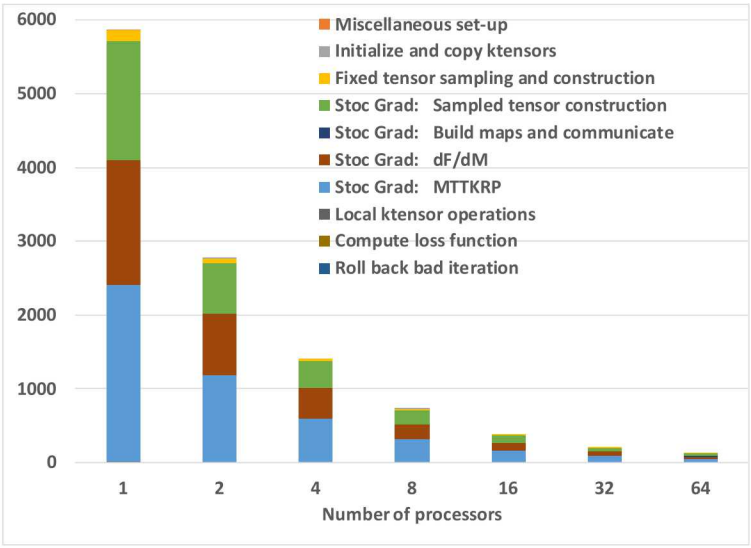

Figure 6-7. Strong scaling for random 4D tensor using L^2 loss, rank $R = 16$, and semi-stratified sampling. Each of the five epochs ran 1000 iterations. Each iteration used semi-stratified sampling with approximately 768K nonzeros and 768K zeros. Error estimation used stratified sampling with approximately 2.6M nonzeros and 2.6M zeros.

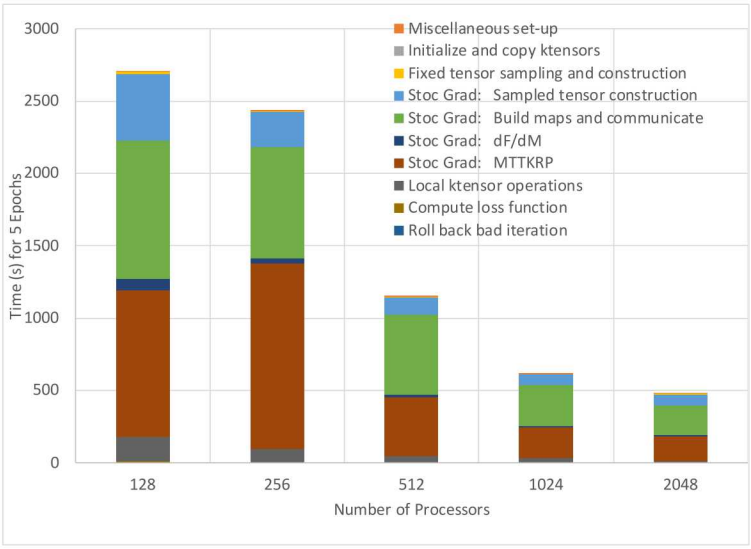

Figure 6-8. Strong scaling for *amazon-reviews* tensor using L^2 loss, rank $R = 16$, and semi-stratified sampling. Each of the five epochs ran 1000 iterations. Each iteration used semi-stratified sampling with approximately 4.2M nonzeros and 4.2M zeros. Error estimation used stratified sampling with approximately 42M nonzeros and 42M zeros.

processors) are within the stochastic gradient computation: evaluating the partial derivative, constructing the sampled tensor, and performing the MTTKRP. No communication is needed to evaluate the partial derivative. Similarly, no communication is needed to sample the tensor, but a small amount of communication occurs in creating the Tpetra maps describing the distribution of the sampled tensor (see Section 3.1). The cost of building maps (the communication pattern) and performing the communication are small in this case, but they grow with the number of processors.

Figure 6-8 presents results for the *amazon-reviews* tensor, using between 128 and 2048 processors of Skybridge (the tensor is too large to run GCP-SGD on fewer processors). We use 84 million samples to estimate the error and 8.4 million samples for each stochastic gradient. Compared to the experiment with the random tensor, we see that communication costs become much more significant in this experiment. The dominant kernels are again in the stochastic gradient computation, but in this case, the communication costs (building maps and communicating) are significant on 128 processors and become a bottleneck on 2048 processors. The scaling is nearly perfect between 256 and 1024 processors, but little speedup is obtained in increasing to 2048 processors.

7. CONCLUSIONS AND FUTURE WORK

We have described GentenMPI, a toolkit for computing low-rank approximations of sparse tensors on distributed memory parallel computers. GentenMPI is built on the Trilinos scientific computing toolkit, which provides data structures and parallel communication classes that can be exploited in tensor decomposition. Using this infrastructure, GentenMPI provides implementations of the classic CP-ALS low-rank decomposition using alternating least squares optimization, and the new GCP-SGD method supporting arbitrary loss functions. We present parallel distribution strategies for sampling tensors in distributed memory environments. And we demonstrate that GentenMPI can achieve good parallel scalability, while enabling decomposition of tensors too large for single-memory computers.

Future work will combine the distributed memory capabilities of GentenMPI with the multicore- and GPU-capabilities of Genten. Both Genten and Trilinos rely on Kokkos for performance-portable multicore and GPU kernels. On-node parallelism related to factor matrices and MPI communication packing/unpacking will be managed by existing Trilinos classes. On-node parallelism in operations such as MTTKRP will exploit methods in Genten. Some modification of GentenMPI's tensor storage will be needed to accommodate use of Genten on the node. In the end, our multicore and GPU version of GentenMPI will exploit the fine-grained parallelism in Trilinos and Genten.

REFERENCES

[1] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007.

[2] Brett W. Bader, Tamara G. Kolda, et al. MATLAB Tensor Toolbox version 3.0-dev. Available online, October 2017. `https://www.tensortoolbox.org`.

[3] C. G. Baker and M. A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, April 2012.

[4] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, Sep 1970.

[5] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, December 2014.

[6] H. Carter Edwards, Christian R. Trott, Daniel Sunderland, et al. Kokkos C++ performance portability programming ecosystem. Available online, April 2015. `https://github.com/kokkos`.

[7] Richard A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis. *Working Papers in Phonetics*, 16(10,085):1 – 84, 1970.

[8] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, September 2005.

[9] Michael A. Heroux, Karen Devine, Roger Pawlowski, Mauro Perego, Siva Rajamanickam, James Willenbring, and many others. Trilinos solver toolkit. Available online, February 1998. `https://github.com/trilinos/Trilinos`.

[10] Mark Hoemmen, Chris Siefert, Jonathan Hu, Brian Kelley, Tim Fuller, Karen Devine, Chris Baker, Mike Heroux, et al. Tpetra linear algebra classes. Available online, April 2012. `https://trilinos.github.io/tpetra.html`.

[11] David Hong, Tamara G Kolda, and Jed A Duersch. Generalized canonical polyadic tensor decomposition. Technical Report 1808.07452, arXiv, 2018.

[12] David Hong, Tamara G. Kolda, and Jed A. Duersch. Generalized canonical polyadic tensor decomposition. *SIAM Review*, 2020. in press.

[13] Oguz Kaya and Bora Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. Technical Report 1412.6980, arXiv, 2015. Published as a conference paper at ICLR 2015.

[15] Tamara G. Kolda. Sparse vs. scarce. Available online, November 2017. `http://www.kolda.net/post/sparse-versus-scarce/`.

[16] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, September 2009.

[17] Tamara G. Kolda and David Hong. Stochastic gradients for large-scale tensor decomposition. Technical Report 1906.01687, arXiv, 2019.

[18] Eric T. Phipps and Tamara G. Kolda. GenTen: Software for generalized canonical polyadic tensor decompositions. Available online, June 2017. `https://gitlab.com/tensors/genten/`.

[19] Eric T. Phipps and Tamara G. Kolda. Software for sparse tensor decomposition on emerging computing architectures. *SIAM Journal on Scientific Computing*, 41(3):C269–C290, 2019.

[20] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017. `http://frostt.io/`.

[21] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *IEEE 30th International Parallel and Distributed Processing Symposium*, pages 902–911, May 2016.

[22] Shaden Smith, George Karypis, et al. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. Available online, November 2015. `https://github.com/ShadenSmith/splatt`.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email—Internal ([redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.