Fast and Scalable Distributed Tensor Decompositions

Muthu Baskaran, Thomas Henretty, James Ezick
Reservoir Labs Inc.
New York, NY 10012
Email: {baskaran,henretty,ezick}@reservoir.com

Abstract—Tensor decomposition is a prominent technique for analyzing multi-attribute data and is being increasingly used for data analysis in different application areas. Tensor decomposition methods are computationally intense and often involve irregular memory accesses over large-scale sparse data. Hence it becomes critical to optimize the execution of such data intensive computations and associated data movement to reduce the eventual time-to-solution in data analysis applications. With the prevalence of using advanced high-performance computing (HPC) systems for data analysis applications, it is becoming increasingly important to provide fast and scalable implementation of tensor decompositions and execute them efficiently on modern and advanced HPC systems. In this paper, we present distributed tensor decomposition methods that achieve faster, memory-efficient, and communication-reduced execution on HPC systems. We demonstrate that our techniques reduce the overall communication and execution time of tensor decomposition methods when they are used for analyzing datasets of varied size from real application. We illustrate our results on HPE Superdome Flex server, a high-end modular system offering large-scale in-memory computing, and on a distributed cluster of Intel Xeon multi-core nodes.

I. INTRODUCTION

Tensors are multi-dimensional arrays that present a logical representation for multi-attribute data. Tensor decomposition is a prominent technique for analyzing multi-attribute data and is being increasingly used for data analysis in different application areas for extracting and gaining insights into the latent properties of data. Tensor decompositions have been successfully used in a range of applications including, but not limited to: cybersecurity [1], geospatial analysis [2], genomics [3], chemometrics [4], computer vision [5], data mining [6] and precision medicine [7].

Tensor decomposition methods are computationally intense and often involve irregular memory accesses over large-scale sparse data. Hence it becomes critical to optimize the execution of such data intensive computations and associated data movement to reduce the eventual time-to-solution in data analysis applications. Further, one of the biggest challenges in data analysis applications is the growing volume of data to be analyzed, which adds non-trivial complexity to the methods for data analysis with respect to quicker turnaround time and feasibility to handle large data.

With the prevalence of using advanced HPC systems for data analysis applications, it is becoming increasingly important to provide fast and scalable implementation of tensor decompositions and execute them efficiently on modern and advanced HPC systems such as clusters of multi-core and many-core systems, high end memory-driven computing servers, and future exascale systems. In addition to utilizing the large processing capability of HPC systems, it is crucial to improve memory and communication performance factors such as overall memory usage, communication cost, synchronization cost, memory reuse, and data locality.

In this paper, we present techniques that enable optimized distributed tensor decomposition methods that achieve faster, memory-efficient, and communication-reduced execution on HPC systems. We implement the optimizations for distributed decomposition methods such that they complement existing state-of-the-art optimizations for sparse tensor computations targeted towards faster and memory-efficient execution on shared-memory systems. Specifically, we integrate the optimizations presented in this paper into ENSIGN [8], a commercially available tensor decomposition package containing state-of-the-art high-performance C implementations of a multitude of tensor decomposition methods. ENSIGN tensor methods are implemented using optimized sparse tensor data structures, namely, mode-specific sparse (MSS) and mode-generic sparse (MGS) tensor data structures [9], and are parallelized and optimized for memory usage, load balancing and data locality [9], [10], [11], [12], [13] in multi-core systems.

We present how we use ENSIGN MGS and MSS data structures along with an effective data distribution strategy for the implementation of distributed decomposition methods. We show how our strategy reduces the overall volume and frequency of communication needed for the execution of such methods on large HPC systems (for example, clusters and high-end in-memory computing servers). Further, we discuss our approach to reduce communication and computation by utilizing the sparsity in the input and generated data in the decomposition methods.

Our contributions in this paper include:

1) Implementation of distributed version of three different widely applicable tensor decomposition methods (and not just the workhorse decomposition method that is not necessarily useful for many applications).

2) Improvements to reduce communication and synchronization cost, and execution time of tensor decomposition methods, and enable the use of powerful tensor decomposition methods in critical applications across multiple domains.
3) Integration of the distributed (high-performance MPI+OpenMP) implementation of the decomposition methods into a commercial tensor package (ENSIGN).
4) Evaluation & demonstration of performance on real data on a modern high-end server and a cluster.

II. BACKGROUND

In this section, we give some background information on the basic definitions and algorithms related to tensor decompositions.

A tensor is a multi-dimensional array and the order of a tensor is the number of dimensions, also called modes, of the tensor. A tensor can be transformed into a matrix (i.e. matricized) by flattening it along any of its modes. The modes of a n-tensor is written as X(n).

There are two popular and prominent tensor decomposition models, namely, CANDECOMP/PARAFAC (CP) and Tucker decompositions. We will focus our discussion in this paper to CP decomposition, especially, three CP decomposition algorithms that have broad applicability in real applications.

There are several matrix operations that are used in tensor decomposition methods. One of the matrix operations that is used in the methods discussed in this work is the Khatri-Rao product. The Khatri-Rao product of two matrices A ∈ R(I×K) and B ∈ R(J×K) (denoted by A ◦ B) results in a matrix of size IJ×K and is defined by:

\[ A ◦ B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \ldots & a_{1K}b_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{J1} & a_{I2}b_{J2} & \ldots & a_{IK}b_{JK} \\ \vdots & \vdots & \ddots & \vdots \\ a_{IN}b_{JN} & a_{IN}b_{JN} & \ldots & a_{IK}b_{JK} \end{bmatrix} \]

A and B must have matching number of columns in order for the Khatri-Rao product to be defined. The element-wise multiplication (Hadamard product) and element-wise division of two matrices A and B are denoted by A • B and A ◊ B, respectively.

The CP decomposition decomposes a tensor into a sum of component rank-one tensors (a N-way tensor is called a rank-one tensor if it can be expressed as an outer product of N vectors). The CP decomposition that factorizes an input tensor X of size I_1 × \ldots × I_N into R components (with factor matrices A^{(1)} \ldots A^{(N)} and weight vector \lambda) is of the form: X = \sum_{r=1}^{R} \lambda_r a^{(1)}_{r} \otimes a^{(2)}_{r} \otimes \ldots \otimes a^{(N)}_{r} where a^{(n)}_{r} represents the r^{th} column of the factor matrix A^{(n)} of size I_n × R and \otimes represents the outer product of vectors.

a) CP-ALS Algorithm: The widely used workhorse algorithm for CP decomposition is the alternating least squares (ALS) method (presented in Algorithm 1).

b) CP-ALS-NN Algorithm: The algorithm for non-negative (NN) CP-ALS decomposition using multiplicative updates [14] is similar to Algorithm 1, but differs only in the way the factor matrices are updated at each step (Line 6). The computation in Line 6 of CP-ALS-NN algorithm looks like: A^{(n)} = A^{(n)} \cdot (U \odot A^{(n)} V)

Algorithm 1 CP-ALS Algorithm

1: Input: X, A^{(1)} \ldots A^{(N)}
2: repeat
3: for n = 1 \ldots N do
4: Compute: U = (X^{(n)} \odot A^{(n)} T ) A^{(m)}
5: Compute: A^{(n)} = U \odot V
6: end for
7: until convergence
8: Output: A^{(1)} \ldots A^{(N)}

Algorithm 2 CP-APR Algorithm

1: Input: X, A^{(1)} \ldots A^{(N)}
2: repeat
3: for n = 1 \ldots N do
4: repeat
5: Compute: \Phi = (X^{(n)} \odot (A^{(n)} \odot A^{(m)} T ) A^{(m)})
6: Compute inner convergence
7: Compute: A^{(n)} = A^{(n)} \cdot \Phi
8: until convergence
9: end for
10: Compute outer convergence
11: until convergence
12: Output: A^{(1)} \ldots A^{(N)}

When the input tensor is sparse, as in the case of most real application tensors, the computations and memory accesses are irregular and the optimizations for parallelization are driven by the sparsity of the input tensor. In our work, we focus on the sparse versions of the following (expensive) computations in the above-mentioned tensor decomposition methods:

- Sparse matricized tensor times Khatri-Rao product (MTTKRP) computation in CP-ALS and CP-ALS-NN algorithms: X^{(n)} \odot A^{(n)} T A^{(m)}
- Sparse \Phi computation (an “extended” MTTKRP computation) in CP-APR algorithm: X^{(n)} \odot (A^{(n)} \odot A^{(m)} T ) A^{(m)}

III. RELATED WORK

In this section, we discuss related work in the literature that have focused on optimizing tensor decompositions in general and specifically on optimizing sparse MTTKRP for shared- and distributed-memory systems. We are not aware of any work on optimizing CP-APR, other than our prior work.
of representation/arrangement of tensor indices with respect to any particular mode. On the other hand, the MSS tensor data structure provides a compressed format for representing and storing sparse tensors in which the tensor indices are ordered and arranged with an orientation towards a “candidate” mode. Both the MGS and MSS tensor formats allow efficient compressed representation and storage of dense sub-blocks within a tensor. The tensor indices in MGS and MSS tensors are optionally sorted and the order of sorting is specified within the data structure.

The data involved in tensor decomposition methods are, primarily, the factor matrices and the input tensor. The input tensor is distributed and represented in each processor element in the form of MGS and/or MSS tensors. The factor matrices are distributed and represented in each processor element as dense or compressed sparse row (CSR) matrices. The distribution of tensor and factor matrices across processor elements are determined by a distribution strategy that is based on the size of the modes of the tensor. It is to be noted that the processor element, here, refers to an element in a “distributed” processor grid. Let the number of distributed processor elements be $n_{\text{proc}}$ for the rest of the discussion.

We use three different data distribution strategies, namely, (1) replicated, (2) partitioned, and (3) distributed. A distribution strategy is selected for each mode of the tensor. The criteria for selection of a strategy for a mode is described below.

- $a)$ Replicated: If the size of the mode is less than $\alpha.n_{\text{proc}}$, $\alpha$ being a replication threshold factor, usually 1, then the mode is replicated. This means, the entire factor matrix corresponding to the mode is replicated across all processor elements.

- $b)$ Partitioned: If the size of the mode is greater than or equal to $\beta.n_{\text{proc}}$, $\beta$ (usually a large constant) being the partition threshold factor, then the mode is partitioned. This means the factor matrix rows are partitioned and distributed across processor elements such that a processor element “completely owns” the rows (or mode indices) in its partition. The input tensor is also partitioned such that the non-zero elements having mode indices corresponding to a partition are placed entirely in the “owner” processor element.

- $c)$ Distributed: If the size of the mode is greater than or equal to $\alpha.n_{\text{proc}}$ but less than $\beta.n_{\text{proc}}$, then the mode is distributed but not partitioned. This means the factor matrix rows are distributed across processor elements but the input tensor is not partitioned across the processor elements for them to completely own the rows in their partition. Hence for a distributed (but not partitioned) mode, the processor elements require data to be communicated from other processing elements for performing the computations associated with the mode.

In the case of distributed or partitioned modes, the factor matrices are distributed sequentially and uniformly across processor elements. So the number of rows that are owned by a processor element for mode $n$ of size $I_n$ is approximately $I_n/n_{\text{proc}}$.
We consider the distribution strategy to also make the selection of tensor format (MSS or MGS) for a mode. The MSS tensor format is specifically aligned to a mode and hence it is typically only used for the computations along the mode. So if all modes are chosen to be represented in the MSS tensor format, there will be \( N \) copies of the tensor (\( N \) being the number of modes). However the advantage of using the MSS tensor format is that it presents a “synchronization-free” (as opposed to a “reduction”) orientation of the tensor along the candidate mode and hence enables synchronization-free computation. We make a trade-off between memory (due to multiple MSS tensors) vs synchronization-free computation. If there is one or more replicated modes, we create a MGS tensor and reuse it for all the replicated modes. If a mode is partitioned, we create a MSS tensor for the mode. If a mode is distributed but not partitioned, we create a MSS tensor only if the size of the mode is greater than \( \gamma \cdot nproc \), \( \gamma \) (usually a large constant) being the MSS threshold factor, otherwise, we use the MGS tensor.

We now discuss the communications mandated by distributed tensor decomposition methods and the actual forms of communications that are needed as per our data distribution strategy. The MTTKRP computation in CP-ALS and CP-ALS-NN methods and the \( \Phi \) computation in CP-APR method are the dominant computations. These computations, since they operate on the tensor and the factor matrices, induce the communications needed to transfer the necessary rows of factor matrices required for the computation. Figures 1 and 2 present the structure and position of the communication calls in CP-APR and CP-ALS(-NN) methods with different distribution strategies - replicated, distributed, and partitioned.

![Fig. 1](image1.png)

Fig. 1. Communication calls in CP-APR with different distribution strategies - replicated, distributed, partitioned.

![Fig. 2](image2.png)

Fig. 2. Communication calls in CP-ALS and CP-ALS-NN with different distribution strategies - replicated, distributed, partitioned.

If there were no non-trivial data distribution strategy, then after every MTTKRP or \( \Phi \) computation and every factor matrix update step (“Update \( A^{(n)} \)” step in Figures 1 and 2), there would be an Allreduce and an Allgather communication, respectively, across all processor elements. From figures 1 and 2, it is clear that the communication calls are reduced with each distribution strategy. The partitioned mode offers the most advantage by reducing the communication call to one “rank-wise Gather” call in the second level of the three-level-deep computation in CP-APR method and in the first level of the two-level-deep computation in CP-ALS(-NN) methods. Further, if the number of partitioned modes is 1, then the communication is completely eliminated for the mode since, in that case, both the update (write) and use (read) of the factor matrix values owned by a processor element happen only within that processor element.

The communication cost involved in the “rank-wise Reduce” communication after the MTTKRP or \( \Phi \) computation is the communication initiated by each processor element to reduce the partial results corresponding to the rows of factor matrices that the processor element owns. The communication cost involved in the “rank-wise Gather” communication after the factor matrix update step is the communication initiated by each processor element to send the updated factor matrix values from the owner processor element to all the processor elements that use the values for the MTTKRP or \( \Phi \) computation. For a replicated mode (for which the size of the factor matrix is small), the communication after MTTKRP or \( \Phi \) computation is a relatively less-expensive Allreduce communication that is needed to update the copy of the factor matrix in each processor element.
B. Utilizing Sparsity in Input and Generated Data

We exploit the sparsity in the input tensor to provide operation-efficient and memory-efficient MTTKRP and Φ computations, as described in [13]. The communication cost in tensor decomposition methods is determined by the cost of communication of factor matrix elements during the methods. One of the facts of tensor decomposition methods on sparse tensors is that the factor matrices tend to become sparser as the method progresses. We exploit this fact and reduce the memory usage and volume (and cost) of factor matrix communication, by communicating only the non-zero matrix values and indices. Further, we reduce the number of operations during the computations by exploiting the sparsity of the factor matrices.

V. Experimental Results

We present the experimental results from the evaluation of our techniques to provide scalable distributed tensor decompositions with reduced communication cost and improved performance.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NNZ</th>
<th>Tensor Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyber-7G</td>
<td>350M</td>
<td>[40M, 10K, 90K, 40K]</td>
</tr>
<tr>
<td>Cyber-28G</td>
<td>1.4B</td>
<td>[160M, 12K, 107K, 40K]</td>
</tr>
<tr>
<td>Cyber-60G</td>
<td>2.8B</td>
<td>[320M, 15K, 110K, 40K]</td>
</tr>
</tbody>
</table>

TABLE I

Datasets: We use multiple datasets from real operational cyber application use case [26] for our evaluation. In other words, the datasets used in our experiments have been used for demonstration and application of fast scalable decompositions for cyber analysis. Hence the evaluation on these datasets not only validate experimental benefits but also crucial application benefits in a critical domain such as cybersecurity. Different CP decomposition methods are suited for different types of datasets and/or different domains and give qualitatively different results. In particular, we have seen CP-APR to be very useful for analyzing sparse count data and hence suited for cybersecurity.

Experimental Systems: We use the following two experimental testbeds for our evaluation:

a) Cluster: We use a 32-node cluster in which each node has 512GB of DDR4 DRAM and two 10-core Intel Xeon E5-2650v3 Haswell processors running at 2.30GHz. The cluster has an Infiniband FDR 56Gbit/s interconnect. We use OpenMPI v1.10.7 and GCC 8.2 for the experiments.

b) HPE Superdome Flex (SDF): We use a HPE SDF server with 4 chassis and 16 sockets (4 sockets per chassis). Each socket has a 18-core Intel Xeon Scalable processor for a total of 288 cores in the server. Each chassis has 48 DIMM slots of 64GB DDR4 memory for a total of 12TB memory. We use HPE-MPI v1.3 for the experiments.

We run only CP-APR (the most expensive method, in terms of computation and communication, among the three methods) on HPE SDF, due to the scope of the engagement to use SDF. For our experiments on HPE SDF, we use a decomposition rank of 100 and we use the Cyber-28G and Cyber-60G datasets (that have 1 billion and 2 billion tensor entries, respectively). For our experiments on cluster, we use a decomposition rank of 10 and we use the Cyber-7G dataset. We use 288 MPI processes and 1 OpenMP thread per MPI process (a configuration that suited the HPE-MPI library) for the SDF experiments. We use 32 MPI processes and 20 OpenMP threads per MPI process for the cluster experiments. The performance of the state-of-the-art baseline distributed version of the methods (without an effective data distribution strategy, effective selection of data structure format, or exploitation of sparsity in input and generated data) is significantly lower (from 2x up to 8x higher execution time) than that of the optimized version presented in this paper.

A. Communication Reduction and Scaling

We measure and characterize the communication cost in the execution of the tensor decomposition methods on the experimental testbeds. Figure 5 shows that the communication performance of the most expensive CP-APR method scales on HPE SDF, when applied on the larger datasets, namely, Cyber-28G and Cyber-60G. The reduced volume and frequency of communication achieved via effective data distribution strategy along with appropriate ENSIGN sparse tensor data structure and exploitation of sparsity in input and generated data contribute to the scaled communication performance. The scaled-up low-latency memory technology of the HPE SDF server contributes to the low absolute value of the communication cost (and eventually, high computation:communication cost ratio). Figure 6 shows that the communication performance scales for all methods on Cyber-7G dataset on the 32-node cluster. However, the computation:communication cost ratio is lower compared to that of HPE SDF due to higher communication latency from the cluster interconnect.

B. Performance Improvement and Scaling

We characterize the performance of the tensor decomposition methods by measuring the overall execution time of the methods (and not just the core computation kernels). Figure 3 shows that the overall performance of the CP-APR method scales well with increasing number of processor cores on HPE SDF, when applied on Cyber-28G and Cyber-60G datasets. This is due to the reduced communication cost (as mentioned above) and near-ideal scaling of computation cost (as shown in Figure 4). Figure 6 shows that the overall performance scales for all methods on Cyber-7G dataset on the 32-node cluster. The computation scales very well and the communication also (as mentioned above) scales.

VI. Conclusion

In this paper, we have developed techniques that will enable the use of powerful tensor decomposition methods in very large-scale critical real-world applications. We have implemented high-performance distributed tensor decomposition methods that have been shown to exhibit improved performance in terms of parallelism, reduced memory usage and reduced communication cost, and scale well on modern HPC systems and high-end servers.
(a) Cyber-28G dataset (1 billion entry tensor)  
(b) Cyber-60G dataset (2 billion entry tensor)

Fig. 3. Performance scaling of CP-APR on Cyber-28G (1 billion entry tensor) and Cyber-60G (2 billion entry tensor) datasets on HPE Superdome Flex.

Fig. 4. Computation scaling of CP-APR on Cyber-28G (1 billion entry tensor) and Cyber-60G (2 billion entry tensor) datasets on HPE Superdome Flex. Note: the x-axis and y-axis are in log scale.

Fig. 5. Communication scaling of CP-APR on Cyber-28G (1 billion entry tensor) and Cyber-60G (2 billion entry tensor) datasets on HPE Superdome Flex. Note: the x-axis and y-axis are in log scale.

Fig. 6. Performance scaling of CP-APR, CP-ALS, and CP-ALS-NN methods on Cyber-7G dataset (350M entry tensor) on a 32-node cluster.
ACKNOWLEDGMENT

This material is partially based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0017081. We express our gratitude to Hewlett Packard Enterprise (HPE) for providing us with access to their high-end HPE Superdome Flex servers for running our experiments.

REFERENCES


