On Optimizing Distributed Tucker Decomposition for Sparse Tensors

Venkatesan T. Chakaravarthy[1], Jee W. Choi[1], Douglas J. Joseph[1], Prakash Murali[*2], Shivmaran S. Pandian[1], Yogish Sabharwal[1], and Dheeraj Sreedhar[1]

[1] IBM Research
{*vechakra,prakmura,ysabharwal,dhsreedh*}@in.ibm.com,
{*jwchoi,djoseph,xliu*}@us.ibm.com
[2] Princeton University
pmurali@cs.princeton.edu

Abstract

The Tucker decomposition generalizes the notion of Singular Value Decomposition (SVD) to tensors, the higher dimensional analogues of matrices. We study the problem of constructing the Tucker decomposition of sparse tensors on distributed memory systems via the HOOI procedure, a popular iterative method. The scheme used for distributing the input tensor among the processors (MPI ranks) critically influences the HOOI execution time. Prior work has proposed different distribution schemes: an offline scheme based on sophisticated hypergraph partitioning method and simple, lightweight alternatives that can be used real-time. While the hypergraph based scheme typically results in faster HOOI execution time, being complex, the time taken for determining the distribution is an order of magnitude higher than the execution time of a single HOOI iteration. Our main contribution is a lightweight distribution scheme, which achieves the best of both worlds. We show that the scheme is near-optimal on certain fundamental metrics associated with the HOOI procedure and as a result, near-optimal on the computational load (FLOPs). Though the scheme may incur higher communication volume, the computation time is the dominant factor and as the result, the scheme achieves better performance on the overall HOOI execution time. Our experimental evaluation on large real-life tensors (having up to 4 billion elements) shows that the scheme outperforms the prior schemes on the HOOI execution time by a factor of up to 3x. On the other hand, its distribution time is comparable to the prior lightweight schemes and is typically lesser than the execution time of a single HOOI iteration.

---

[*]Research conducted while the author was at IBM Research.

1 Introduction

Tensors are the higher dimensional analogues of matrices that are useful in representing data in three or higher dimensions. Different tensor decompositions have been proposed, among which the two most prominent are the Tucker decomposition [28] and CP (canonical polyadic) decomposition [5, 8]. The Tucker decomposition represents high-rank data in the form of a low-rank structure: given an N-dimensional input tensor $\mathcal{T}$, the decomposition (approximately) expresses $\mathcal{T}$ as the product of a small N-dimensional core tensor, and a set of N factor matrices. It can be viewed as a generalization of the SVD (Singular Value Decomposition) to higher dimensions. While the factor matrices represent the most significant information along the different dimensions, the core captures the interaction among them. Figure 1 provides a pictorial depiction in 3-D. The CP decomposition generalizes rank factorization and can be viewed as a constrained form of Tucker decomposition, wherein the core is a diagonal tensor with uniform length across all the dimensions.

The Tucker decomposition has been used in performing tasks such as data compression and principal component analysis. It finds application in diverse domains from signal processing [22] to text analytics [21]. We refer to the survey by Kolda and Bader [17] for a detailed discussion on Tucker decomposition and its applications. The decomposition has been well-studied in sequential, shared memory, map-reduce and distributed settings, for both dense and sparse tensors [1, 2, 3, 6, 11, 15, 18, 25, 27, 26].

HOOI Procedure. The HOOI (Higher Order Orthogonal Iterator) procedure [20] provides a popular iterative method for constructing the Tucker decomposition of a given tensor. The procedure transforms any given decomposition to a more refined decomposition and it is usually invoked multiple times until a suitable convergence criterion is attained. The procedure is bootstrapped with an initial decomposition obtained via methods such as the HOSVD (Higher Order SVD) algorithm [19]; alternatively, a random set of factor matrices can also be used. Given a decomposition, the HOOI procedure (a single invocation) constructs the new decomposition in N iterations, wherein each iteration involves a TTM-Chain (Tensor-times-matrix chain) operation, followed by an SVD step. Finally, the newly constructed factor matrix rows are communicated among the processors to be used in the subsequent HOOI invocation.

Our goal is to develop an efficient implementation of the Tucker decomposition for sparse tensors on distributed memory systems. We build on a prior framework of Kaya and Uçar [15], which provides mechanisms for implementing the TTM and SVD operations in a distributed manner. The execution time of the HOOI procedure critically depends on the scheme used for distributing the input tensor among the processors (MPI ranks).

Prior Schemes. We consider distribution schemes proposed for Tucker decomposition, as well as the related CP decomposition. The schemes can be classified into three types: (i) coarse-grained schemes [7, 23, 15] that partition the tensor into large chunks (sub-tensors) and assign the chunks to the processors; (ii) fine-grained schemes that assign individual tensor elements [15]; (iii) a medium grained scheme [25] that strikes a balance between the
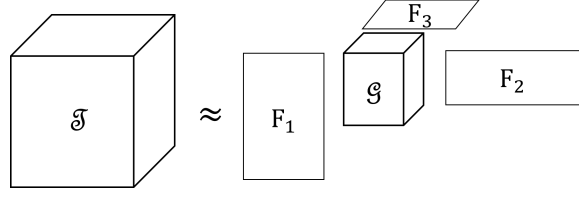
2

Figure 1: Illustration for the Tucker decomposition in 3-D. $\mathcal{G}$ is the core tensor, and F_1, F_2 and F_3 are the factor matrices.

two. Among the above methods, hypergraph partitioning (a fine grained schme) [15] typically offers the best HOOI execution time. However, hypergraph partitioning is expensive and the time taken for distributing the tensor is significantly higher than the execution time of a single HOOI invocation. On the other hand, the other schemes are real-time, lightweight procedures with much faster distribution time (comparable to HOOI execution).

Our Contributions. Our main goal is to demonstrate that high performance on the HOOI execution time can be achieved via lightweight schemes:

- We present a lightweight distribution scheme called Lite that is easy to implement and parallelize.

- We define certain fundamental metrics (implicit in the prior work) associated with the HOOI procedure and prove that Lite is near-optimal on all these metrics. As a result, the scheme is near-optimal on computational load, load balance and communication volume associated with the TTM and the SVD components.

- Lite outperforms the prior schemes on real-life tensors in terms of the HOOI execution time. Lite may incur higher overall communication volume, because of higher factor matrix data transfer. Nevertheless, in contrast to the CP decomposition, the computation time is the dominant factor in the Tucker decomposition and as a result, Lite achieves better HOOI execution time.

We present a detailed experimental study evaluating the different schemes over a benchmark of large real-life tensors having up to 4 billion elements. The results show that the new scheme achieves the best of both worlds:

- On HOOI execution time, Lite outperforms the hypergraph based scheme by a factor of up to 4x. Taking the best of prior schemes in each test case, the gain is upto a factor of 3x. HOOI scales well under the scheme: on MPI ranks from 32 to 512, the speedup is in the range of 8.7x to 15.5x.

- Lite distributes tensors with billions of elements in real-time, with its distribution time comparable to the prior lightweight schemes and a single HOOI invocation.

3

> **Input:** A tensor $\mathcal{T}$ and a decomposition $\{\mathcal{G}, \mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N\}$
>   Size of tensor $\mathcal{T}$: $L_1 \times L_2 \times \cdots \times L_N$
>   Size of core $\mathcal{G}$: $K_1 \times K_2 \times \cdots K_N$,
>   Size of factor matrix $\mathbf{F}_n$: $L_n \times K_n$
> **Output:** A new refined decomposition $\{\tilde{\mathcal{G}}, \widetilde{\mathbf{F}}_1, \widetilde{\mathbf{F}}_2, \ldots, \widetilde{\mathbf{F}}_N\}$
>   Size of core $\tilde{\mathcal{G}}$ and factor matrices $\widetilde{\mathbf{F}}_n$: same as input.
> **Procedure:**
> For each mode n from 1 to N
>   TTM-Chain: Perform TTM along all the modes, except n.
> $\mathcal{Z} \leftarrow \mathcal{T} \times_1 \mathbf{F}_1^T \times \cdots \times_{n-1} \mathbf{F}_{n-1}^T \times_{n+1} \mathbf{F}_{n+1}^T \times \cdots \times_N \mathbf{F}_N^T$.
>   SVD: $\widetilde{\mathbf{F}}_n \leftarrow$ leading K_n left singular vectors of $\mathbf{Z}_{(n)}$
> New core: $\tilde{\mathcal{G}} \leftarrow \mathcal{T} \times_1 \widetilde{\mathbf{F}}_1^T \times \cdots \times_N \widetilde{\mathbf{F}}_N^T$
> **return** $\{\tilde{\mathcal{G}}, \widetilde{\mathbf{F}}_1, \widetilde{\mathbf{F}}_2, \ldots, \widetilde{\mathbf{F}}_N\}$.

Figure 2: HOOI Procedure

Related Work. Tucker decomposition has been studied under various settings. In the case of sparse tensors, the direct evaluation of TTM-Chain via computing intermediate tensors leads to memory blowup. To address the issue, Kolda and Sun [18] proposed a memory efficient approach (MET) and Baskaran [3] developed semi-dense structures. Recently, Smith and Karypis [26] used the compressed sparse fiber representation (CSF) to reduce the computational load, while limiting the memory blowup. The above implementations target sequential and shared-memory settings. Kaya and Uçar [15] presented the first distributed memory implementation; we build on their framework. The TTM component of the framework is a special case of the MET approach, wherein no intermediate tensors are computed.

For the Tucker decomposition of dense tensors, MATLAB [2], single-machine [30] and distributed [1, 6] implementations have been proposed. Prior work has also studied the Tucker decomposition on the MapReduce platform [11]. Other tensor decompositions such as CP factorization have been explored as well (e.g.,[13, 16, 12, 25, 14]).

2 Tucker Decomposition

In this section, we briefly describe tensor concepts pertinent to our discussion, and present the HOOI procedure.

2.1 Tensors

Fibers. Tensors are multi-dimensional arrays. Consider an N-dimensional tensor $\mathcal{T}$ of size $L_1 \times L_2 \times \cdots \times L_N$. The elements of $\mathcal{T}$ can be canonically indexed by a coordinate vector $\langle l_1, l_2, \ldots, l_N \rangle$, where each l_j belongs to $[1, L_j]$. For $1 \leq n \leq N$, a *mode-n fiber* refers to the vector obtained by fixing all the coordinates except n, i.e., $\langle l_1, \ldots, l_{n-1}, *, l_{n+1}, \ldots l_N \rangle$. These fibers have length L_n and the number of fibers is $\widehat{L}_n = \Pi_{j \neq n} L_j$. Each fiber can be

4

identified by an index $\langle l_1, l_2, \ldots, l_{n-1}, l_{n+1}, \ldots, l_N \rangle$. In the analogous case of matrices, two types of fibers can be found: row vectors and column vectors.

Tensor Unfolding. A *mode-n unfolding* refers to the matrix of size $L_n \times \widehat{L}_n$ obtained by arranging the mode-n fibers as the columns. We adopt a standard lexicographic ordering of the fibers, wherein the fiber indexed $\langle l_1, l_2, \ldots, l_{n-1}, l_{n+1}, \ldots, l_N \rangle$ is placed in the column numbered $\Sigma_{j \neq n} l_j (\Pi_{i \neq n, i < j} L_i)$. The matrix is denoted as $\mathbf{T}_{(n)}$.

Tensor-Times-Matrix Multiplication (TTM). The tensor $\mathcal{T}$ can be multiplied along mode-n by any matrix $\mathbf{A}$ provided $\mathbf{A}$ has size $K \times L_n$ (for some K) and the operation is denoted $\mathcal{Z} = \mathcal{T} \times_n \mathbf{A}$. Conceptually, the operation applies the linear transformation $\mathbf{A}$ to all the mode-n fibers. It is realized via the matrix-matrix multiplication $\mathbf{A} \times \mathbf{T}_{(n)}$, and taking the output matrix to be the mode-n unfolding of $\mathcal{Z}$. The output tensor retains the same length along all modes, except mode n, where its length becomes K, (i.e., $\mathcal{Z}$ has size $L_1 \times \cdots \times L_{n-1} \times K \times L_{n+1} \times \cdots \times L_N$).

TTM-Chain. The *TTM-chain* operation refers to multiplying $\mathcal{T}$ along multiple distinct modes $S = \{n_1, n_2, \ldots, n_r\}$ by matrices $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_r$, where $\mathbf{A}_j$ has size $K_j \times L_{n_j}$. The output is a tensor $\mathcal{Z}$ whose length remains L_j, for all modes $j \notin S$ and changes to K_j, for all modes $j \in S$. We denote the operation as $\mathcal{Z} = \mathcal{T} \times_{n_1} \mathbf{A}_1 \times \cdots \times_{n_r} \mathbf{A}_{n_r}$. The operation is commutative, namely the r TTMs can be performed in any order [20]..

2.2 HOOI Procedure

The Tucker decomposition of $\mathcal{T}$ approximately represents the tensor as the product of a small *core tensor* $\mathcal{G}$ and a set of *factor matrices* $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N$:

$$\mathcal{T} \approx \mathcal{Z} = \mathcal{G} \times_1 \mathbf{F}_1 \times_2 \mathbf{F}_2 \times \cdots \times_N \mathbf{F}_N.$$

The core is of size $K_1 \times K_2 \times \cdots \times K_N$, with each $K_n \leq L_n$, which are user-specified. Each factor matrix $\mathbf{F}_n$ has size $L_n \times K_n$, We write the decomposition as $\{\mathcal{G}; \mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N\}$.

The HOOI procedure [20] is a popular method that generalizes SVD to higher order tensors and produces a Tucker decomposition with the additional property that the factor matrices are orthonormal. The procedure takes as input any decomposition $\{\mathcal{G}, \mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_N\}$ and produces a new, more refined decomposition $\{\widetilde{\mathcal{G}}, \widetilde{\mathbf{F}}_1, \widetilde{\mathbf{F}}_2, \ldots, \widetilde{\mathbf{F}}_N\}$. It can be invoked multiple times to obtain better refinements until a suitable convergence criterion is reached (such number of invocations fixed a priori). The process must be bootstrapped by providing an initial decomposition, which we can find using methods such as HOSVD [19]; alternatively, random factor matrices can also be used.

The HOOI procedure, a single invocation, is shown in Figure 2. For computing each new factor matrix $\widetilde{\mathbf{F}}_n$, the procedure utilizes the alternating least squares paradigm and works in two steps. First, it performs a TTM-chain operation by skipping mode n and multiplying $\mathcal{T}$ by the transposes of all the other factor matrices $\mathbf{F}_j$ (with $j \neq n$) and obtains a tensor $\mathcal{Z}$. The tensor $\mathcal{Z}$ has length compressed from L_j to K_j along all modes $j \neq n$. The mode-n unfolding of $\mathcal{Z}$, denoted $\mathbf{Z}_{(n)}$, is a matrix of size $L_n \times \widehat{K}_n$, where $\widehat{K}_n = \Pi_{j \neq n} K_j$. In
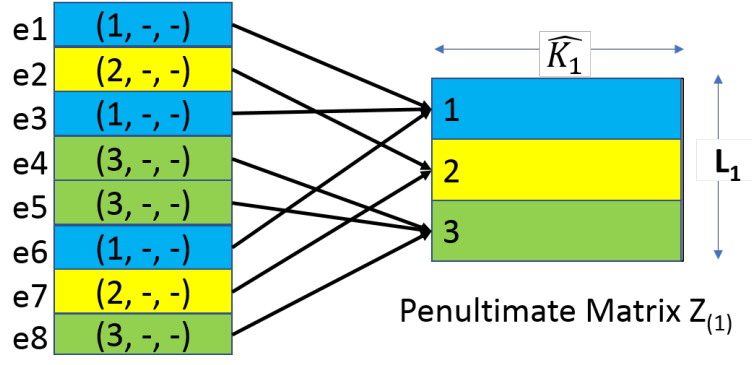
Figure 3: TTM-Chain reformulation

the next step, the procedure performs an SVD on $\mathbf{Z}_{(n)}$ and obtains the leading K_n singular vectors of $\mathbf{Z}_{(n)}$. These singular vectors are arranged in the form of columns to obtain the new factor matrix $\widetilde{\mathbf{F}}_n$. We refer to the matrix $\mathbf{Z}_{(n)}$ as the *penultimate matrix*, since it is one TTM short of a full TTM chain.

The new core tensor is computed in the last step. However, we can see that the refinement procedure only involes the factor matrices, and so it is not necessary to compute the core in each invocation. Instead, it suffices to compute the core only once after all the invocations are completed. Consequently, we focus on optimizing the computation of the TTM-chain and the SVD components.

3 Distributed Framework

As mentioned in the introduction, we build on the distributed framework of Kaya and Uçar [15]. Here, we present an outline of the framework focusing on the aspects critical for our discussion.

The input sparse tensor $\mathcal{T}$ is represented in the coordinate format. Let $\mathcal{E}$ denote the set of all non-zero elements. Each element $e \in \mathcal{E}$ is represented by a coordinate vector $(l_1, l_2, \ldots, l_N)$ (where each $l_n \in [1, L_n]$) and a value $\mathsf{val}(e) \in \mathbb{R}$. Consider a distributed setting consisting of P processors (MPI ranks), numbered $0, 1, \ldots, P-1$.

The HOOI procedure involves of N iterations. Consider the computation along any mode $n \in [1, N]$, consisting of a TTM-Chain operation that generates the penultimate matrix $\mathbf{Z}_{(n)}$ of size $L_n \times \widehat{K}_n$, followed by an SVD operation on the matrix. In order to evaluate the TTM-Chain in a distributed manner, the framework uses a reformulation via the Kronecker product.

Reformulation. We partition the elements into groups based on the nth coordinate, called *slices*: for each $l \in [1, L_n]$, define Slice_n^l as the set of elements having the nth coordinate as l. The reformulation is based on the observation that any row $\mathbf{Z}_{(n)}[l, :]$ is determined only by the contributions from the elements in Slice_n^l. Figure 3 provides an illustration using a 3-D tensor with eight elements, by considering the TTM-Chain operation along the first
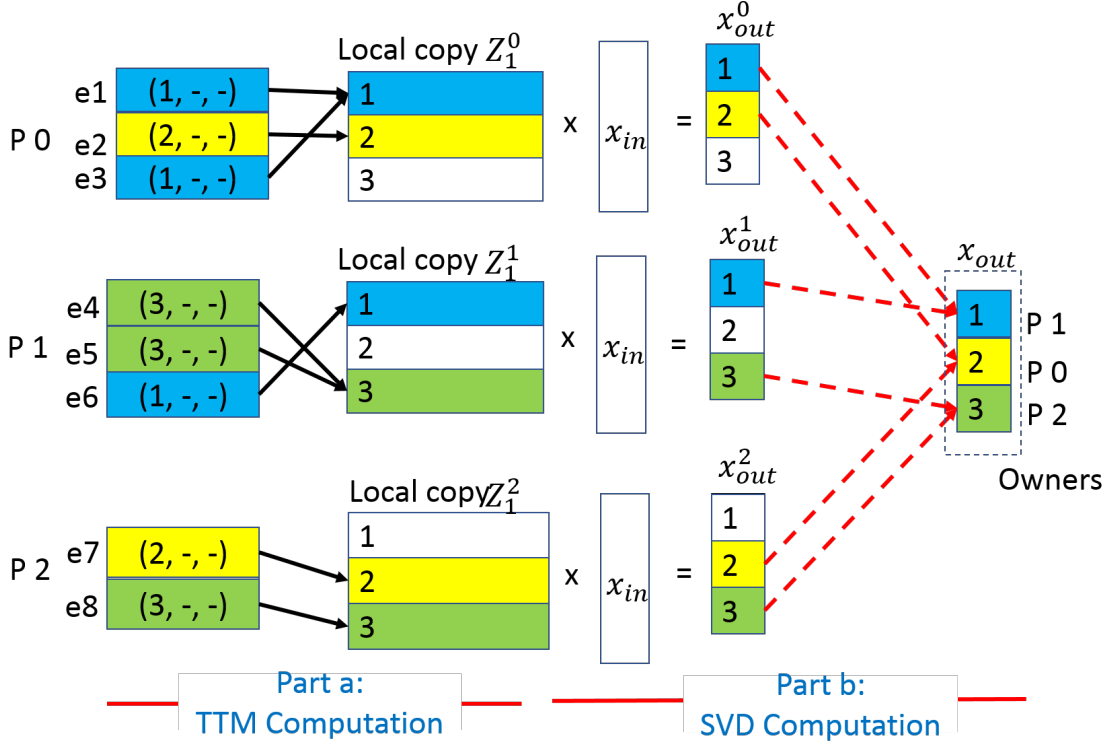
6

Figure 4: Framework illustration

mode ($n = 1$). In this example, $L_1 = 3$ and so, there are three slices: $\mathsf{Slice}_1^1 = \{e_1, e_3, e_6\}$, $\mathsf{Slice}_1^2 = \{e_2, e_7\}$ and $\mathsf{Slice}_1^3 = \{e_4, e_5, e_8\}$. The rows to which the elements contribute are shown by arrows. Each slice and the corresponding row are assigned the same color.

While the row to which e contributes is determined by its nth coordinate, the contribution is determined by the other $(N-1)$ coordinates of e. The contribution, denoted $\mathsf{contr}_n(e)$, is a vector of length $\widehat{K}_n$, the same as the length of rows of $\mathbf{Z}_{(n)}$. It is computed via the Kronecker (or outer) product of the rows indexed by the above $(N-1)$ coordinates in the corresponding factor matrices. We vectorize the resultant $(N-1)$-dimensional tensor and scale by $\mathsf{val}(e)$ to get $\mathsf{contr}_n(e)$; the details are provided in the Appendix. The reformulation states that for any row-index $l \in [1, L_n]$, the row l^{th} is given by:

$$\mathbf{Z}_{(n)}[l, :] = \sum_{e \in \mathsf{Slice}_n^l} \mathsf{contr}_n(e). \tag{1}$$

TTM Component. We distribute the input tensor using a *distribution policy* (a mapping) $\pi : \mathcal{E} \to [0, P-1]$ that assigns each element e to a processor $p = \pi(e)$, called the *owner* of e. Equivalently, the policy partitions the set of elements $\mathcal{E}$ into P parts $\mathcal{E}^0, \mathcal{E}^1, \ldots, \mathcal{E}^P$, where $\mathcal{E}^p$ denotes the set of elements assigned to the processor p. Given a policy π, each processor p computes a local copy of the penultimate matrix $\mathbf{Z}_{(n)}^p$ by considering only the contributions made by the elements owned by it. Namely, we initialize $\mathbf{Z}_{(n)}^p$ to all 0 and

for each element $e \in \mathcal{E}^p$, we add the vector $\text{contr}_n(e)$ to the row $\mathbf{Z}^p_{(n)}[l,:]$, where l is the nth coordinate of e. The global penultimate matrix $\mathbf{Z}_{(n)}$ is simply the sum of the local copies, i.e., $\mathbf{Z}_{(n)}$ is sum-distributed.

Part (a) of Figure 4 provides an illustration over three processors using a simple policy π that partitions the elements in a lexicographic manner: $\mathcal{E}^0 = \{e_1, e_2, e_3\}$, $\mathcal{E}^1 = \{e_4, e_5, e_6\}$ and $\mathcal{E}^2 = \{e_7, e_8\}$. The local copies are also shown.

In the above procedure, a processor may not contribute to all the rows of $\mathbf{Z}_{(n)}$. For a row-index $l \in [1, L_n]$, we say that a processor p *shares* Slice^l_n, if it owns at least one element from the slice, i.e., $\mathcal{E}^p \cap \mathsf{Slice}^l_n \neq \emptyset$. The processor contributes only to the rows corresponding to the slices shared by it; the other rows are said to be empty. Let $\mathtt{R}^p_n$ denote the number of slices shared by p, or equivalently, the number of rows to which p contributes. By omitting the empty rows, we can represent the local copy succinctly as a matrix of size $\mathtt{R}^p_n \times \widehat{K}_n$. Apart from reducing the memory footprint, the above truncation provides significant advantages in optimizing the subsequent SVD operation. In Figure 4, the empty rows are colored white; here, $L_1 = 3$ and $\mathtt{R}^p_n = 2$ for all the processors.

SVD Component. While the matrix $\mathbf{Z}_{(n)}$ can be constructed explicitly by aggregating the local copies, the approach may lead to high volume of communication. Instead, the framework performs the SVD operation directly over the local copies by employing the Lanczos bidiagonalization method [9], an iterative matrix-free procedure. The Lanczos method can be explained using an *oracle* (i.e., query-answering) model. The method works iteratively, wherein each iteration generates two query vectors, a column vector $\overrightarrow{x}_{\text{in}}$ and a row vector $\overrightarrow{y}_{\text{in}}$, and our task is to evaluate the matrix-vector products $\overrightarrow{x}_{\text{out}} = \mathbf{Z}_{(n)} \cdot \overrightarrow{x}_{\text{in}}$ and $\overrightarrow{y}_{\text{out}} = \overrightarrow{y}_{\text{in}} \cdot \mathbf{Z}_{(n)}$ and return the answers $\overrightarrow{x}_{\text{out}}$ and $\overrightarrow{y}_{\text{out}}$ to the method.

Regarding the first product, each processor computes the local answer $\overrightarrow{x}^p_{\text{out}} = \mathbf{Z}^p_{(n)} \cdot \overrightarrow{x}_{\text{in}}$. These get aggregated by a global point to point reduction operation, as follows. The framework uses a suitable *row-index mapping* $\sigma_n : [1, L_n] \to [0, P-1]$ that assigns each row-index l to a processor $\sigma_n(l)$ called the owner of l. The owner is chosen to be one among the processors sharing Slice^l_n. The owners accumulate partial contributions received from the other processors. Thus, the global answer $\overrightarrow{x}_{\text{out}}$ is output in a distributed manner according to σ_n. In Figure 4, processors 1, 0 and 2 are the owners and communication is shown by dashed arrows.

The second product $\overrightarrow{y}_{\text{out}} = \overrightarrow{y}_{\text{in}} \cdot \mathbf{Z}_{(n)}$ is executed in an analogous manner. For each coordinate $l \in [1, L_n]$, the owner $\sigma_n(l)$ sends the value $\overrightarrow{y}_{\text{in}}(l)$ to all the processors sharing Slice^l_n. Upon receiving the above values, each processor p assembles a partial vector $\overrightarrow{y}^p_{\text{in}}$ of length $\mathtt{R}^p_n$ and performs the product $\overrightarrow{y}^p_{\text{out}} = \overrightarrow{y}^p_{\text{in}} \mathbf{Z}^p_{(n)}$. We obtain the answer vecotor $\overrightarrow{y}_{\text{out}}$ by doing the summation $\overrightarrow{y}_{\text{out}} = \sum_p \overrightarrow{y}^p_{\text{out}}$ (via `MPI_Allreduce`). Figure 5 illustrates the process.

Factor Matrix Transfer. The Lanczos algorithm produces the factor matrix $\widetilde{\mathbf{F}}_n$ in a distributed manner, wherein each row $\widetilde{\mathbf{F}}_n[l,:]$ gets generated at the owner $\sigma_n(l)$. These rows are needed for the TTM computation of the next HOOI invocation. Towards that goal, the owner $\sigma_n(l)$ sends the row $\widetilde{\mathbf{F}}_n[l,:]$ to all the processors that would require the row for the subsequent TTM computation.
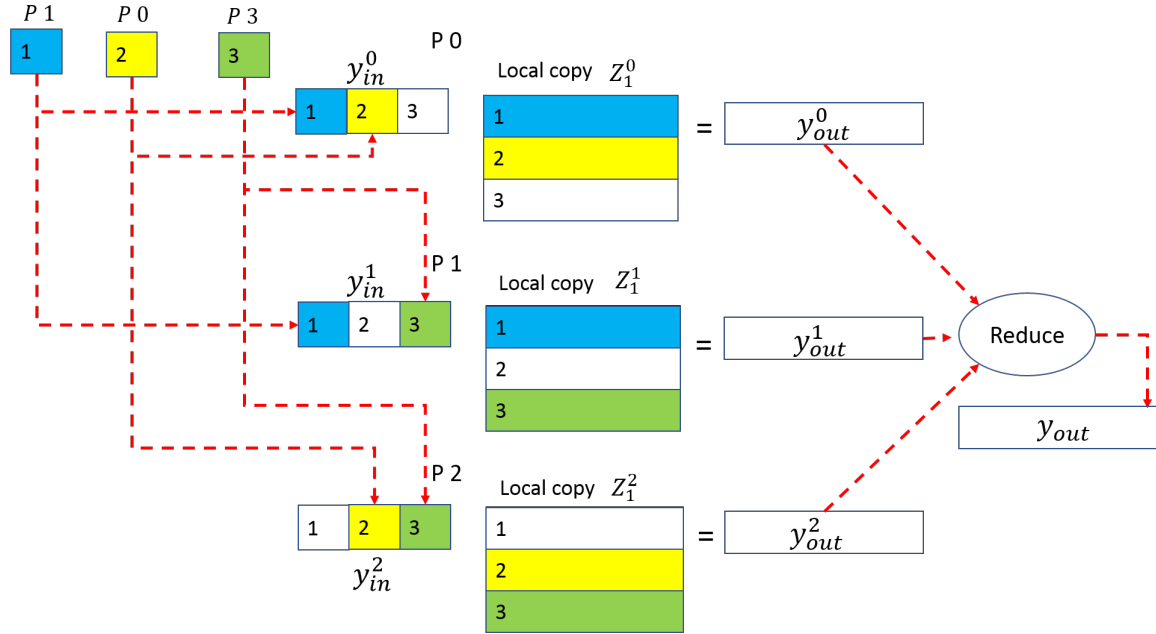
Figure 5: Vector-Matrix Product

Distribution Schemes. The execution time of the HOOI procedure critically depends on the choice of the distribution policy π, since the policy determines the parameters of computational load (FLOPs), load balance and communication volume. An efficient policy must optimize the above parameters with respect to the HOOI computation along all the N modes. The task becomes easier, if we use N distribution policies, each customized with respect to the computation along a single mode. We call such a sequence of N policies $(\pi_1, \pi_2, \ldots, \pi_N)$ as a *distribution scheme*. If a single policy π is used across all modes, we refer to the scheme as *uni-policy scheme*, and the general case as *mulit-policy scheme*. Uni-policy schemes need to store only a single copy of the input tensor (distributed among the processors), whereas multi-policy schemes must store N copies, one along each mode. However, multi-policy schemes offer more flexibility and opportunities for optimization as different policies may be appropriate along different modes.

4 Performance Metrics

In this section, we identify certain fundamental metrics that determine the computational load and communication volume incurred by the HOOI procedure, under a given distribution scheme $(\pi_1, \pi_2, \ldots, \pi_N)$. The HOOI procedure consists of three components, TTM, SVD and factor matrix transfer, of which the first two involve computation and the latter two involve communication. We analyze the efficacy of a scheme along each mode n separately. The cumulative performance across all modes can be computed via suitable aggregation.

9

4.1 Computational Load

We generalize the notations $\mathcal{E}^p$ and R_n^p to multi-policy schemes in a natural manner. Let $\mathcal{E}_n^p$ denote the the set of elements owned by processor p along mode n, i.e., $\mathcal{E}_n^p = \{e : \pi_n(e) = p\}$. For a row-index $l \in [1, L_n]$, we say that p *shares* Slice_n^l, if it owns at least one element from the slice with respect to the policy π_n, i.e., $\mathcal{E}_n^p \cap \text{Slice}_n^l \neq \emptyset$. Let R_n^p denote the number of slices shared by p along mode n.

TTM Computation. The TTM component has to evaluate $|\mathcal{E}|$ Kronecker products (one for each element). Thus, the TTM computation load is the same for all distribution schemes. However, the policy π_n may induce load imbalance by distributing the elements among the processors in a non-uniform manner. The TTM load imbalance along mode n is captured by our first metric:

$$\text{Metric 1:} \qquad \text{E}_n^{\texttt{max}} = \max_p |\mathcal{E}_n^p|.$$

The optimal value the metric is the average $\lceil |\mathcal{E}|/P \rceil$, which can be achieved by uniformly distributing the elements.

SVD Computation. For any matrix-vector product associated with a Lanczos query, each processor p incurs a computational load of $\text{R}_n^p \cdot \widehat{K}_n$, the size of its local copy. Let Q_n be the number of queries raised by the Lanczos procedure. Summed across all queries and all processors, the total oracle load is given by $Q_n \cdot \widehat{K}_n \cdot \sum_p \text{R}_n^p$. Since Q_n and $\widehat{K}_n$ are the same for all policies, the oracle load along mode n is captured by our second metric:

$$\text{Metric 2:} \qquad \text{R}_n^{\texttt{sum}} = \sum_p \text{R}_n^p.$$

Similarly, the load imbalance within the oracle computation along mode n is captured by our third metric:

$$\text{Metric 3:} \qquad \text{R}_n^{\texttt{max}} = \max_p \text{R}_n^p.$$

The above discussion has omitted other computations (such as internal to the Lanczos algorithm) that are common across all schemes.

The metric $\text{R}_n^{\texttt{sum}}$ is the aggregate number of times the slices are shared, across all processors. We say that a slice is *good*, if it is shared by only one processor; otherwise, the slice is said be *bad*. If a slice S is good, then the corresponding row is non-empty only in the local copy of the processor sharing S, whereas for a bad slice, the row becomes non-empty in multiple local copies. Thus, bad slices lead to redundancy in the penultimate matrix and SVD computation, and result in higher load. An optimal policy is to assign each slice in its entirety to a single processor, thereby making all the slices good. This would yield the optimal value of $\text{R}_n^{\texttt{sum}} = L_n$. In addition, if the processors are assigned an equal number of slices, we get the optimal value of $\lceil L_n/P \rceil$ on the metric $\text{R}_n^{\texttt{max}}$. For example, in Figure 4, each mode-1 slice is shared by two processors, leading to $\text{R}_n^{\texttt{sum}} = 6$, which results in a two-factor increase in the load compared to the setting where all slices are good.

4.2 Communication Volume

SVD Communication.   For each query raised by the Lanczos procedure, the SVD component aggregates the local answers to get the global answers. Consider the first product $\overrightarrow{x}_{\text{out}} = \mathbf{Z}_{(n)} \cdot \overrightarrow{x}_{\text{in}}$. For each row-index $l \in [1, L_n]$, each processor sharing Slice_n^l, except the owner $\sigma_n(l)$, sends one unit each of data (a real number) to the owner, where the data gets accumulated. The number of units is the same as the number of processors sharing the slice minus one. Summed across all row-indices, the communication volume per matrix-vector product is $\texttt{R}_n^{\text{sum}} - L_n$. The product $\overrightarrow{y}_{\text{in}} \cdot \mathbf{Z}_{(n)}$ can also be shown to incur the same volume. Summed across all the Q_n queries, the oracle communication volume along mode n is $Q_n \cdot (\texttt{R}_n^{\text{sum}} - L_n)$. Since Q_n and L_n are constants across schemes, we can measure the oracle volume by $\texttt{R}_n^{\text{sum}}$, the same metric that determines the oracle load. As before, we have omitted other communications that are common across the schemes.

Factor Matrix Transfer.   Each row $\mathbf{F}_n[l, :]$ of the factor matrix must be communicated to all the processors that would require the row for TTM computation in the next HOOI invocation. In the case of uni-policy schemes, a processor requires the row, if it shares Slice_n^l. Excluding the owner, the number of processors is $\text{Slice}_n^l - 1$. Each row consists of K_n entries. Summed across all rows, the total communication volume is $K_n \cdot (\texttt{R}_n^{\text{sum}} - L_n)$ units. Thus, for uni-policy schemes, the factor matrix transfer volume is also determined by the parameter $\texttt{R}_n^{\text{sum}}$.

The case of multi-policy schemes is more intricate: a processor requires the row $\mathbf{F}_n[l, :]$, if it owns an element $e \in \text{Slice}_n^l$ with respect to any of the $(N - 1)$ policies, excluding π_n. (i.e., there exists an element $e \in \text{Slice}_n^l$ and a mode $j \neq n$ such that $\pi_j(e) = p$). Hence, for multi-policy schemes, the factor matrix volume cannot be determined from our metrics. We shall measure the volume empirically.

The above metrics measure the efficacy of a scheme along a given mode n. The cumulative performance across all modes can be computed via suitable aggregation, considering the mode-specific factors such as the number of queries Q_n and the core length K_n.

Summary.   We identified the following parameters influencing the HOOI execution time and the associated metrics: TTM load balance ($\texttt{E}_n^{\text{max}}$), SVD computation load and communication volume ($\texttt{R}_n^{\text{sum}}$), SVD load balance ($\texttt{R}_n^{\text{max}}$). The factor matrix communication volume is also determined by $\texttt{R}_n^{\text{sum}}$ for uni-policy schemes.

4.3 Computation vs Communication

It is useful to understand the breakup of the HOOI execution time in terms of computation and communication time. Here, we present an intuitive comparison, taking as example 3-D tensors and uniform core size of $K_1, K_2, K_3 = K$. Along mode n, the TTM and SVD components involve $m \cdot K^2$ and $Q_n \cdot K^2 \cdot \texttt{R}_n^{\text{sum}}$ units of computation (FLOPs), respectively (m is the number of non-zero elements). In accordance with SLEPc [9], our implementation of the Lanczos method involves $2 \cdot K$ iterations, resulting in $Q_n = 4 \cdot K$ queries. On the other hand, the SVD component and the factor matrix transfer components incur $Q_n \cdot (\texttt{R}_n^{\text{sum}} - L_n)$ and $K_n(\texttt{R}_n^{\text{sum}} - L_n)$ units of communication, respectively. We can observe that
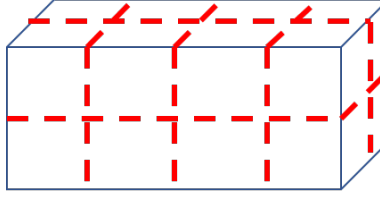
11

Figure 6: Example grid for $P = 16$. The grid is $4 \times 2 \times 2$.

the amount of computation is significantly larger than communication, especially for distribution schemes with low redundancy ($\mathtt{R}_n^{\mathtt{sum}}$ being close to L_n). The intuition is confirmed by our experimental evaluation, which shows that the computation time is dominant even for the multi-policy schemes considered in the study.

4.4 CP vs Tucker Decomposition

It is of interest to compare the CP and the Tucker decompositions. Both follow the ALS paradigm and the elements get distributed using a suitable scheme. The factor matrix transfer step is similar, but the other operations are significantly different. As an illustration, consider a 3-D tensor of size $L \times L \times L$ and core of size $K \times K \times K$. The main computation in CP is the matricized tensor times Khatri-Rao product (MTTKRP): for each element, the operation computes the Hadamard product of two K-length vectors ($O(K)$ FLOPs). The corresponding operation in HOOI is the Kronecker product ($O(K^2)$ FLOPs). In addition, HOOI computes the SVD of a large penultimate matrix of size $L \times K^2$. As a result, computation time is the dominant factor in HOOI. In the case of CP, load balance and communication volume are important. In the case of Tucker, load balance ($\mathtt{E}_n^{\mathtt{max}}$ and $\mathtt{R}_n^{\mathtt{max}}$) and SVD redundancy ($\mathtt{R}_n^{\mathtt{sum}}$) are important, and it is crucial to have low SVD redundancy, perhaps even at the cost of higher communication. Hence, design considerations for distribution schemes for Tucker become different. Schemes that work best for CP may not work as well for Tucker, and vice versa.

5 Prior Distribution Schemes

In this section, we discuss prior schemes proposed in the context of Tucker decomposition [15], as well the related CP decomposition [25]. The schemes can be categorized in to three types.

Coarse Grained Schemes. These are multi-policy schemes. Along each mode n, the policy π_n is constructed by assigning each slice in its entirety (all its elements) to a suitably chosen processor. All the slices are good and the metric $\mathtt{R}_n^{\mathtt{sum}}$ (capturing SVD load) attains the optimal value of L_n. However, these schemes typically perform poorly on the metric $\mathtt{E}_n^{\mathtt{max}}$ (capturing TTM load balance), since real-life tensors tend to have slices that are much larger than the average $\lceil |\mathcal{E}|/P \rceil$. The imbalance can be somewhat mitigated via careful slice assignement strategies such as below [25]: arrange the mode-n slices in a random order and allocate contiguous blocks of slices to the processors. Other strategies similar

12

in spirit have been proposed in prior work [15, 7, 23]. We denote the above scheme as CoarseG. Even with the above heuristics, coarse grained schemes tend to incur high TTM load imbalance.

Fine Grained Schemes. These are uni-policy schemes that address the TTM load imbalance by assigning individual elements, rather than entire slices. Here, the key issue is to ensure that each slice is shared by few processors so that SVD redundancy is low. Towards that goal, building on prior work [14], Kaya and Uçar [15] devised a fine grained scheme via reduction to hypergraph partitioning, a well-studied NP-hard problem. The idea is to construct a hypergraph by taking the elements as vertices and the slices (along all modes) as hyperedges. Then, we construct a (uni-policy) π by finding a balanced min-cut partitioning. The formulation models both the metrics $\text{E}^{\max}$ and R_n^{sum}. Though the scheme achieves good performance on the HOOI execution time, the time taken for hypergraph partitioning is significantly higher than the HOOI execution time (single invocation). Consequently, the scheme is used offline. We denote the schme as HyperG.

Medium Grained Scheme. For CP decomposition, Smith and Karypis [25] proposed a lightweight, a medium-grained scheme that strikes a tradeoff between the above to schemes. The idea is to factorize the number of processors P in a suitable manner $P = q_1 \times q_2 \times \cdots \times q_N$ and overlay a processor grid of the above size over the tensor. Then, each sub-tensor is assigned to a processor. The indices along each mode are randomly permuted to offset any skew in element distribution within the input tensor. The choice of the processor grid is crucial in determining the performance. Along mode n, each slice can be shared by up to P/q_n processors in the worst case and so, q_n is fixed in proportion to L_n. We denote the scheme as MediumG. Figure 6 provides an example grid.

Row-Index Mapping. As in prior work [25], we fix the row-index mapping σ_n as follows. For each row $\mathbf{F}_n[l, -]$, the owner is selected to be one among the processors sharing the slice Slice_n^l, taking into account communication load balance arising in the SVD and the factor matrix transfer operations.

6 Distribution Scheme Lite

Among the prior schemes, CoarseG is optimal on the metric R_n^{sum}, whereas MediumG and HyperG are superior on the metric $\text{E}_n^{\max}$. Uni-policy schmes (such as MediumG and HyperG) suffer from higher SVD redundancy, since they try to construct a single policy that can perform well on all the modes simultaneously. Multi-policy schemes can optimize the process better by constructing N distribution policies, each customized for the computation along a single mode. In this section, we present a lightweight, multi-policy scheme called Lite, which is provably near-optimal on all the three metrics $\text{E}_n^{\max}$, R_n^{sum} and $\text{R}_n^{\max}$, resulting in better computation time. Though the scheme may incur higher communication volume, it achieves better HOOI time, since computation time is the dominant factor.
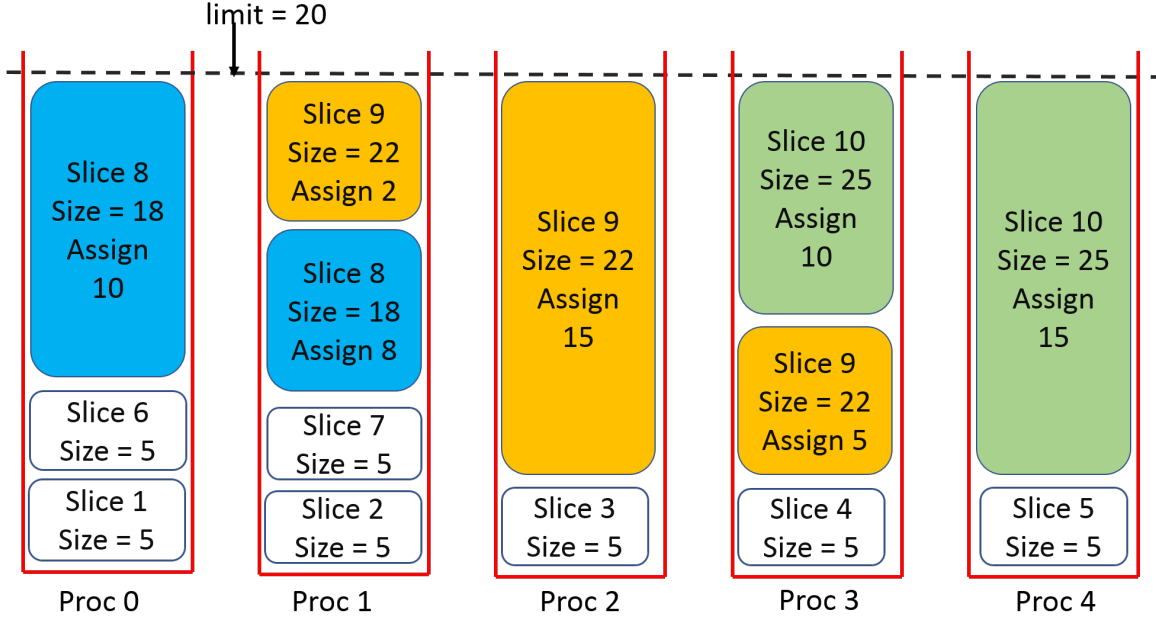
Figure 7: Illustration of Lite. Here, $|\mathcal{E}| = 100$ and $P = 5$. The limit is $\lceil |\mathcal{E}|/P \rceil = 20$. We have ten slices with sizes in the sorted order as: $5, 5, 5, 5, 5, 5, 5, 18, 22$ and 25. Seven slices get processed in a round robin fashion in the first stage; if we assign the eighth slice to processor 2, it would get 23 elements, violating the limit. For the three slices processed in the second stage, the number of elements assigned to each processor is also shown.

6.1 Lite Scheme

The intuition behind Lite is drawn from coarse grained schemes, which have optimal $\mathtt{R}_n^{\mathtt{sum}}$, but suffer from TTM load imbalance, because the elements may get distributed in a non-uniform manner. We can attempt to address the issue by carefully assigning the slices so that the maximum number of elements received by the processors (i.e., $\mathtt{E}_n^{\mathtt{max}}$) is minimized. The problem is the same as the classical makespan minimization on identical parallel machines [29]: assume that the processors are machines and each slice S is a task with execution time equal to $|S|$; we wish to assign the tasks to the machines so that the makespan (overall completion time) is minimized. The problem is NP-hard and heuristics with approximation guarantees are known. For instance, a well-known heuristic is the best processor fit (BPF) procedure: scan the slices and assign each slice to the currently least loaded processor. The above heuristic is guaranteed to output a solution within factor 2 of the optimal solution.

There are two issues with the above approaches. The first is that the tensor may have very large slices, in which case, even the optimal assignment of slices would incur high value of $\mathtt{E}_n^{\mathtt{max}}$ and TTM load imbalance. Secondly, the processors may receive an uneven number of slices, leading to high value of $\mathtt{R}_n^{\mathtt{max}}$ and SVD load imbalance. In designing Lite, we address the first issue by sharing the large slices among multiple processors, and show that the second issue can be addressed by sorting the slices in the increasing order of their

14

sizes.

We next describe the Lite distribution scheme along mode n. Imagine that the processors are bins that need to be filled by the elements. We wish to achieve the optimal value on the metric $\mathbf{E}_n^{\max}$, given by the average $\lceil |\mathcal{E}|/P \rceil$. We consider the above value to be a hard limit on the number of elements that can be added to a bin. The construction first sorts the mode-n slices by their cardinalities and proceeds in two stages.

In the first stage, we consider the slices in the increasing order of cardinalities and assign them to the processors in a round-robin fashion. We stop the process, if assigning a slice to the current bin would make it violate the limit. At this point, we move to the second stage. The remaining slices are large in size and we fill the bins to their limit by sharing the large slices among multiple processors. To this effect, we scan the large slices and the bins concurrently. If the whole of the current slice can be assigned to the current bin without violating the limit, we do so and move to the next slice. Otherwise, we arbitrarily select elements from the current slice and add to the current bin till the limit is reached, and then move to the next bin. Thus, the elements of each large slice get assigned to a contiguous set of processors.

In the above scheme, towards achieving fast tensor distribution time, we sort the slices using the parallel sample-sort algorithm [10], a divide-and-conquer strategy similar to quicksort. Given an array of keys, the idea is to use random sampling to derive a set of P keys called splitters. We then partition the array into P buckets based on the splitters and let each processor sort a bucket independently. Figure 7 provides an illustration. A pseudocode is given in Figure 8; it outputs the set of elements $\mathcal{E}_n^p$ assigned to each processor p.

6.2 Performance Guarantee and Discussion

Theorem 6.1. *For the scheme* Lite, *along any mode* n,

1. $\mathbf{E}_n^{\max} \leq \lceil |\mathcal{E}|/P \rceil$.

2. $\mathbf{R}_n^{\mathtt{sum}} \leq L_n + P$.

3. $\mathbf{R}_n^{\max} \leq \lceil L_n/P \rceil + 2$.

A proof sketch is provided in Section 6.3. At a high level, the first metric is explicitly ensured by setting the hard limit. Regarding the other two metrics, all the slices processed in the first stage are good and the round-robin process implies that every bin receives the same number of slices. We shall argue that at most P slices can remain in the second stage and that each processor can share at most two of them.

The theorem shows that the scheme is optimal on the metric $\mathbf{E}^{\max}$ and achieves perfect TTM load balance. Recall that the optimal values for the metrics $\mathbf{R}_n^{\mathtt{sum}}$ and $\mathbf{R}_n^{\max}$ are L_n and $\lceil L_n/P \rceil$, respectively. On the above two metrics, Lite is away from optimality only by additive factors of P and 2, respectively. Within the SVD component, the communication volume per matrix-vector product is given by $\mathbf{R}_n^{\mathtt{sum}} - L_n$ and so, the scheme incurs only P units of communication per matrix-vector product. Thus, the scheme is near-optimal on the computational load, load balanace and communication volume associated with the

```
L ← Lₙ
```
$L \leftarrow L_n$
Sort $\mathsf{Slice}_n^1, \mathsf{Slice}_n^2, \ldots, \mathsf{Slice}_n^L$ in increasing order of cardinality.
Let $S_1, S_2, \ldots, S_L$ be the slices in sorted order.
$\mathtt{limit} \leftarrow \lceil |\mathcal{E}|/P \rceil$.
For all $l \in [1, L]$, $\mathcal{E}_n^p \leftarrow \emptyset$.
Stage 1:
$p \leftarrow 0$
For $t = 1, 2, 3, \ldots$
    If $(|\mathcal{E}_n^p \cup S_t| > \mathtt{limit})$ then GOTO Stage 2
    else
        Assign all elements of S_t to p: $\mathcal{E}_n^p \leftarrow \mathcal{E}_n^p \cup S_t$.
        $p \leftarrow (p + 1) \bmod P$
Stage 2:
$p \leftarrow 0$
while$(p < P)$
    $g \leftarrow \mathtt{limit} - |\mathcal{E}_n^p|$.   // gap with respect to $\mathtt{limit}$
    If$(|S_t| \leq g)$ then
        Assign all elements of S_t to p: $\mathcal{E}_n^p \leftarrow \mathcal{E}_n^p \cup S_t$.
        $t \leftarrow t + 1$ // Move to next slice
    else
        $X \leftarrow$ Select any g elements from S_t.
        Assign selected elements to p: $\mathcal{E}_n^p \leftarrow \mathcal{E}_n^p \cup X$
        Remove selected elements from S_t: $S_t \leftarrow S_t \setminus X$
        $p \leftarrow p + 1$ // Move to next processor

Figure 8: Lite: Distribution along mode n

SVD component. We note that the scheme may incur overall higher communication volume, due to higher factor matrix data transfer. However, as shown in our experimental study, Lite outperforms the prior schemes on the overall HOOI execution time, since the computation time is the dominant factor.

We next briefly compare the memory requirements of Lite with that of prior schemes. Being a multi-policy scheme, Lite needs to store N copies of the input tensor, one along each mode. However, due to low SVD redundancy, the scheme requires lesser space for storing the penultimate matrices. Consequently, as shown by our experimental evaluation, Lite performs better or comparable to the prior schemes in terms of the memory requirements.

A recent work on the CP decomposition [27] tries to handle the large slices by uniformly distributing the elements of slices larger than a heuristically determined threshold. Our algorithm Lite solves the isuse by using a principled approach yielding near-optimal bounds.

6.3 Proof of Theorem 6.1

Part (1) is readily true, since the scheme explicitly ensures that the number of elements assigned to any processor is at most $\lceil |\mathcal{E}|/P \rceil$. We next prove part (2) of the theorem. Let $\hat{t}$ denote the iteration in which the procedure made the exit to the second stage. Let S_1

and S_2 denote the slices processed in the first and the second stages, respectively; namely, $S_1 = \{S_1, S_2, \ldots, S_{\hat{t}-1}\}$ and $S_2 = \{S_{\hat{t}}, S_{\hat{t}+1}, \ldots, S_L\}$, where $L = L_n$.

By the construction of the second stage, each slice $S \in S_2$ is assigned to a set of contiguous processors. We call the first among these processors as the *head* of S and the others are said to form the *tail* of S. As an illustration, in Figure 7, for slice 9, processor 1 is the head and the processors 2 and 3 form the tail. Observe that any processor can participate in the tail of at most one slice.

Let $\text{num}(S_1)$ denote the aggregate number of times the slices from S_1 are shared; define S_2 similarly. The slices in S_1 are all good and so, $\text{num}(S_1) = |S_1|$. The quantity $\text{num}(S_2)$ is same as the number of times the processors act as the heads plus the number of times they participate in tails. The first quantity is $|S_2|$, since every slice has a single head. The second quantity is at most P, since as we observed earlier, every processor participates in the tail of at most one slice. Therefore, $\text{num}(S_2) \leq |S_2| + P$. Put together, we get that R_n^{sum} is at most $L_n + P$, proving part (2) of the theorem. Moreover, since every slice is shared by at least one processor, the above result also implies that there can be at most P bad slices.

We next prove part (3) of the theorem by showing that for any processor p, $\text{R}_n^p \leq \lceil L_n/P \rceil + 2$. The first stage assigns the slices in a round-robin fashion and so, the number of slices from S_1 assigned to p is at most $\lceil |S_1|/P \rceil \leq \lceil L_n/P \rceil$. Regarding slices from S_2, an important issue is that while the slices in S_1 are good, those in S_2 can be potentially good or bad. We say that a slice S is *ugly*, if $S \in S_2$ and it is good. The ugly slices pose a difficulty: it is hypothetically possible that a large number of ugly slices get assigned to the processor p in the second stage, leading to a high value of R_n^p and load imbalance in the oracle computation. We eliminate the possibility by proving that ugly slices do not exist.

Towards that goal, we first argue that the first stage follows the best processor fit strategy: namely, in any iteration t, the slice S_t gets assigned to the processor having the least number of elements. Below, we formalize and the prove the claim. For $t \geq 1$, let p_t denote the processor that receives the slice S_t in iteration t, i.e., $p_t = (t \mod P)$. For an iteration t and processor p, let $h_t(p)$ denote the number of elements assigned to p till the beginning of iteration t (not including the assignment made during the iteration t); thus $h_1(p) = 0$, for all p.

Lemma 6.2. *For any iteration $t \geq 1$, we have that $h_t(p_t) \leq h_t(p)$, for all p.*

Proof. We prove the lemma by establishing a stronger statement that the number of elements assigned to the processors are in ascending order in a cyclic manner starting with p_t, and the difference between the largest and the smallest assignments does not exceed $|S_t|$. The proof goes via induction and the strengthening helps in the induction step. For any processor p, we write '$p \oplus 1$' and '$p \ominus 1$' to mean '$(p+1) \mod P$' and '$(p-1) \mod P$', respectively.

Claim: For any iteration $t \geq 1$: (a) for any $p \neq p_t$, $h_t(p) \geq h_t(p \ominus 1)$; (b) $h_t(p_t \ominus 1) - h_t(p_t) \leq |S_t|$.

The lemma follows from part (a) of the claim. We prove the claim by induction. The claim is trivially true for the base case $t = 1$, since all the processors are empty to start with. Assume that the claim is true for t and we prove it for $t + 1$. By the induction hypothesis, at the beginning of iteration t, p_t has the least and $(p_t \ominus 1)$ has the largest number of elements, and the difference is at most $|S_t|$. During the iteration t, S_t gets

17

Tensor	L_1	L_2	L_3	L_4	nnz	Sparsity
delicious	532K	17.2M	2.4M	1.4K	140M	4.2×10^{-15}
enron	6K	5K	244K	1K	54M	5.4×10^{-9}
flickr	319K	28M	1.6M	731	112M	1.0×10^{-14}
nell1	2.9M	2.1M	25.4M	-	143M	9.1×10^{-13}
nell2	12K	9K	28K	-	77M	2.4×10^{-5}
amazon	4.8M	1.7M	1.8 M	-	1.7B	1.1×10^{-10}
patents	46	239 K	239	-	3.5B	1.3×10^{-3}
reddit	8.2M	176K	8.1M	-	4.6B	3.9×10^{-10}

Figure 9: Tensor datasets

assigned to p_t. So, p_t becomes the processor with the largest number of elements and $p \oplus 1$ becomes the processor with the least number of elements. Barring p_t, the ordering among the other processors remains the same. Thus, for any $p \neq (p \oplus 1)$, $h_{t+1}(p) \geq h_{t+1}(p \ominus 1)$. Since $p_{t+1} = p \oplus 1$, we have proved part (a). Regarding part (b), p_t has lesser number of elements than $p \oplus 1$ in the beginning of iteration t and receives the slice S_t. Hence, $h_{t+1}(p_t) - h_{t+1}(p \oplus 1) \leq |S_t| \leq |S_{t+1}|$; rephrased, $h_{t+1}(p_{t+1} \ominus 1) - h_{t+1}(p_{t+1}) \leq |S_{t+1}|$. Part (b) is proved. $\qquad\square$

For a processor p, let $\widehat{h}(p)$ denote the number of elements assigned to p at the end of the first stage and let $\widehat{g}(p) = \lceil |\mathcal{E}|/P \rceil - \widehat{h}(p)$ denote the gap to the limit. We next use Lemma 6.2 to argue that any slice $S \in \mathcal{S}_2$ is too big to fit the gap of any processor.

Lemma 6.3. *For any $S \in \mathcal{S}_2$ and processor p, $|S| > \widehat{g}(p)$.*

Proof. Let $\widehat{p} = \widehat{t} \mod P$. The exit to the second stage implies that $|S_{\widehat{t}}| + \widehat{h}(\widehat{p}) > \lceil |\mathcal{E}|/P \rceil$, or equivalently $|S_{\widehat{t}}| > \widehat{g}(\widehat{p})$. Since the slices are sorted, $S_{\widehat{t}}$ has the least cardinality among the slices in $\mathcal{S}_2$. On the other hand, Lemma 6.2 implies that $\widehat{p}$ has the least number of elements at the beginning of iteration $\widehat{t}$, or equivalently, $\widehat{p}$ has the largest gap. The lemma is proved. $\qquad\square$

The above lemma shows that second stage cannot assign any slice from $\mathcal{S}_2$ in its entirety to a single processor; namely, ugly slices do not exist. Thus, all the slices in $\mathcal{S}_2$ are shared by at least two processors and hence, any processor p can act as the head of at most slice from $\mathcal{S}_2$. We observed earlier that p can participate in the tail of at most one slice from $\mathcal{S}_2$. Therefore, p can share at most two slices from $\mathcal{S}_2$. Since p shares at most $\lceil L_n/P \rceil$ from $\mathcal{S}_1$, we get that $\mathtt{R}_n^p \leq \lceil L_n/P \rceil + 2$. Part (3) of the theorem is proved.

7 Experimental Evaluation

In this section, we evaluate the performance of Lite and the prior schemes CoarseG, MediumG and HyperG, (the best known scheme under each category) on HOOI execution time, distribution time, memory usage and related statistics.

7.1 Experimental Setup

System. The experiments were conducted on a cluster of Power-8 nodes (20 cores, 256 GB, 4 GHz) connected via InfiniBand in a fat-tree topology. We launch 16 MPI ranks per node, each mapped to a core. We use 2 to 32 nodes, leading to 32 to 512 MPI ranks.

Tensor Datasets: The dataset consists of eight real-world tensors drawn from the FROSTT repository [24] that represent data from NLP datasets, social bookmarking and rating services. Of the eight tensors, five are medium-sized with at least 50 million elements and the other three are big tensors with more than billion elements. For each tensor, Figure 9 shows the length along each mode, the number of non-zero elements (nnz) and the sparsity (ratio of the number of non-zero elements to the total size of the tensor). While the first three are 4-dimensional, the others are 3-dimensional.

Implementation: Our implementation is based on MPI. We use the iterative Lanczos bidiagonalization method [9] for SVD operation. In accordance with SLEPc [9], we set the number of Lanczos iterations to be $2K$, where K is the number of singular vectors requested. We use ATLAS 3.10.1 for dense linear algebra. The compiler used is gcc 4.8.5.

For the HyperG scheme, we obtained the hypergraph partitioning using the parallel Zoltan library [4], used in prior work [25] as well. We could not obtain the partitioning for the three big tensors using the library. So, we consider the scheme only on the medium-sized tensors.

Core Size: The HOOI time is dependent on the size of the core tensor. As in prior work [15, 18, 3], we use a uniform core length of $K_n = K$ for all modes n and set $K = 10$ in all the experiments, except one, where we study the effect of increasing the core size.

7.2 HOOI Execution Time

We first compare the different schemes on the HOOI execution time (single invocation) on the medium-size tensors; the big tensors are considered separately later in the section. We consider three different configurations. Setting $K = 10$, the first two configurations consider the smallest (32) and the largest (512) number of ranks in our setup. The third configuration studies the effect of increasing the core size, and sets $K = 20$ and number ranks as 512.

HOOI Execution Time. The execution times are shown in Figure 10. Among the prior schemes, the scheme offering the least execution time varies across the test cases and overall HyperG has better performance. We can see that Lite offers the best performance on all the datasets and configurations. It outperforms CoarseG, MediumG and HyperG by factors upto 12x, 4.5x and 4.1x, respectively. Compared to the best prior scheme in each test case, the performance improves by a factor of up to 3x, with the performance gain increasing with increase in number of ranks and core size.

Towards understanding the above phenomenon, we analyze the HOOI components and the underlying metrics. For this purpose, we use the second configuration ($K = 10$ and ranks= 512) and the first three tensors as illustrative example.
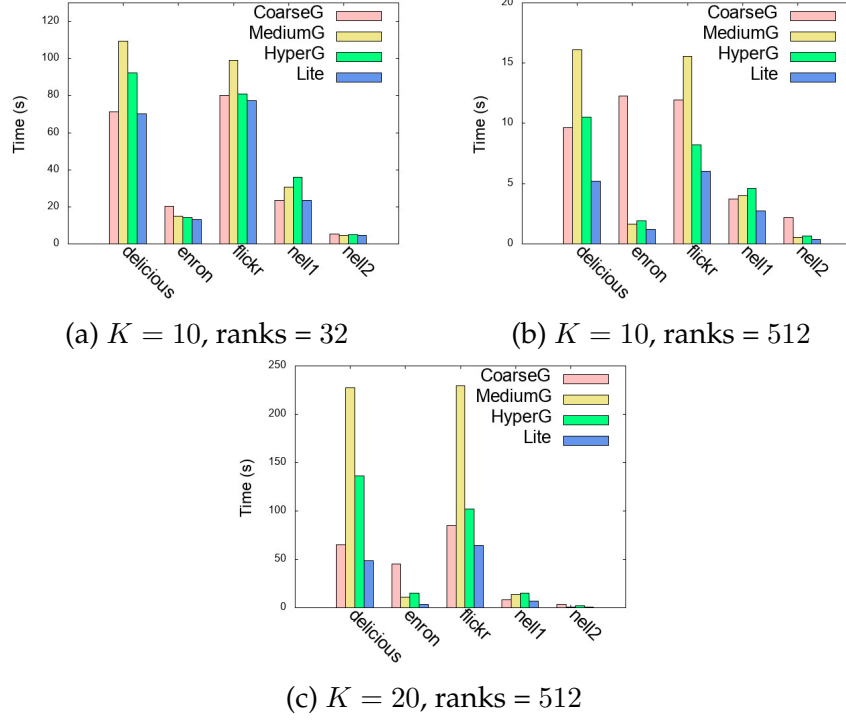
(a) $K = 10$, ranks = 32

(b) $K = 10$, ranks = 512

(c) $K = 20$, ranks = 512

Figure 10: HOOI execution time comparison on the medium-sized tensors.

(a) delicious

(b) enron

(c) flickr

Figure 11: HOOI execution time breakup for $K = 10$, number of ranks 512
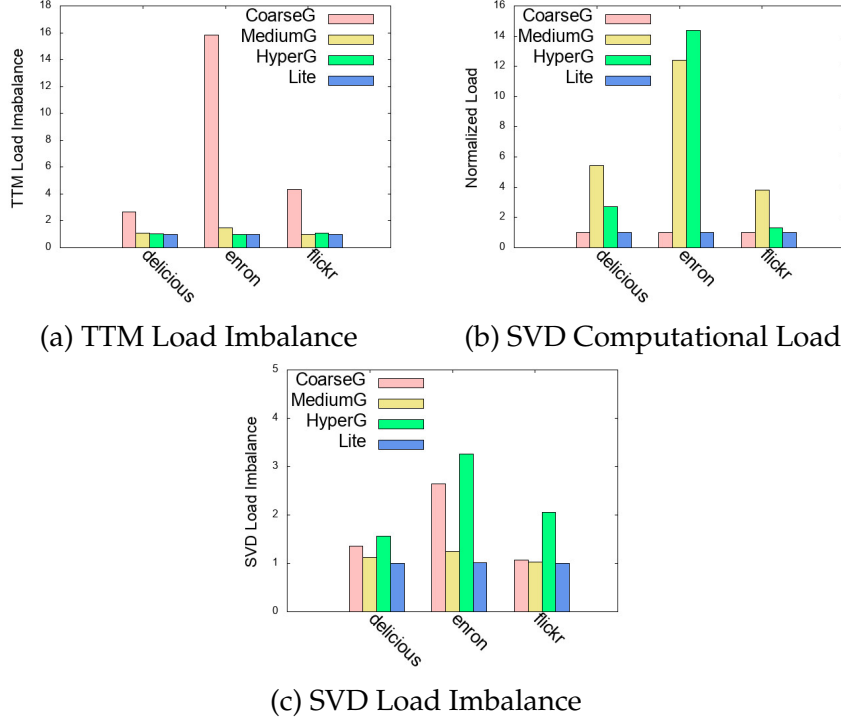
(a) TTM Load Imbalance   (b) SVD Computational Load

(c) SVD Load Imbalance

Figure 12: Analysis of computation time parameters at $K = 10$ and ranks $= 512$

Time Breakup. Figure 11 provides the breakup of HOOI execution time in terms of TTM and SVD computation time, and the total communication time (SVD plus the factor matrix transfer). We can see that the computation time dominates the overall execution time. While CoarseG is better on SVD, MediumG and HyperG are better on TTM computation. Lite performs well on both the components.

Computation Metrics. The above behavior can be understood by analyzing the underlying metrics, presented in Figure 12. The computation time is determined by the TTM load balance, SVD computational load and load imbalance. We measure the computational load (FLOPs) by taking the aggregate along all the modes. The load balance is given by the ratio of maximum to the average across the processors, with the optimal value being one.

From Figure 12 (a), we can see that MediumG, HyperG and Lite achieve near-perefct TTM load balance. The CoarseG scheme performs poorly, because it assigns entire slices to the processors and as a result, the processors receiving large slices induce load imbalance. For instance, Enron has 54M elements yielding an average of 105K elements per processor at 512 ranks, but the tensor has slices of size 5M elements. The above example shows that the severe load imbalance cannot be mitigated even by careful slice assignment mechanisms such as the best processor fit.

The optimal SVD load is attained when each slice is owned by a single processor. We measure the redundancy in the SVD computation by normalizing the load with re-
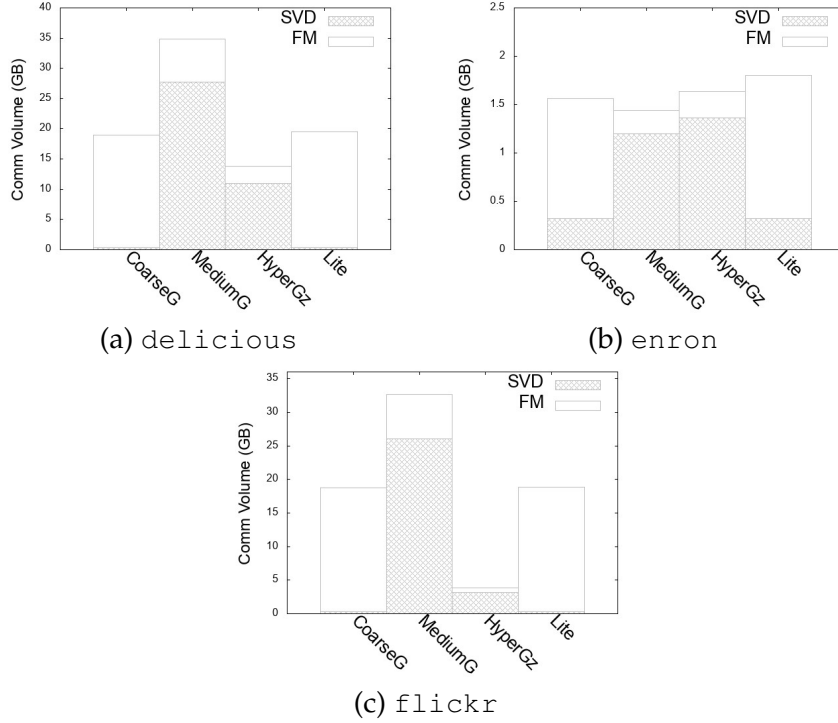
(a) `delicious`

(b) `enron`

(c) `flickr`

Figure 13: Communication volume for $K = 10$, number of ranks 512

Time(s)	CoarseG	MediumG	Lite
amazon	89.0	13.1	8.6
patents	96.4	15.5	14.2
reddit	232.1	23.6	21.6

Figure 14: HOOI execution time on the big tensors

spect to the optimal value; the normalized load in shown in Figure 12 (b). Being uni-policy schemes, MediumG and HyperG have to contend with computations across multiple modes simultaneously, leading to high redundancy. Recall that under MediumG, each mode-n slice can be shared by up to P/q_n processors in the worst case. Though not reaching the worst case bound, we can see that the redundancy is high under MediumG, resulting in higher HOOI execution time. In contrast, CoarseG achieves the optimal redundancy of one unit, since all the slices are good under the scheme. Being near-optimal on the metric $\mathtt{R}_n^{\mathtt{sum}}$, Lite attains redundancy close to one.

Regarding SVD load balance, we can see from Figure 12 (c) that Lite performs well, since it is guaranteed to be near-optimal on the metric $\mathtt{R}_n^{\mathtt{max}}$. The MediumG scheme also performs well.

Communication Volume. We observed that the computation time dominates the HOOI time. Here, we analyze the small communication time by considering the communication volume. Figure 13 shows the breakup in terms of the SVD and the factor matrix trans-

fer (FM) components. The SVD component involves communication during oracle query answering and other communication that are common across the schemes. The oracle communication volume along mode n is given by $\mathrm{R}_n^{\mathrm{sum}} - L_n$. The metric $\mathrm{R}_n^{\mathrm{sum}}$ is L_n for the CoarseG and close to L_n for Lite. Hence, the two schemes incur little SVD communication, but being multi-policy schemes, they have higher FM volume. In contrast, MediumG and HyperG incur lesser FM volume, but higher SVD volume. The HyperG scheme achieves good tradeoff and performs the best on the overall communication volume. Nevertheless, Lite outperforms HyperG on HOOI execution time, since the computation time is the dominant factor.

Big Tensors. We next consider the three big tensors at 512 ranks and $K = 10$. The smallest of these tensors has 1.7 billion elements and the aggregate cardinalities of the hyperedges is three times bigger. Given the large size of the resulting hypergraph, we could not obtain the hypergraph partitioning on these tensors using the two libraries. The execution times for the other three schemes are shown in Figure 14. The number of non-zero elements in these tensors is much larger compared to their dimension lengths and as a result, the TTM computation time dominates. The CoarseG scheme performs poorly due to TTM load imbalance, since these tensors have very large slices. In contrast, by the virtue of good TTM load balance, the other two schemes perform well. We can see that Lite achieves the best execution time on all the three tensors and outperforms MediumG by a factor of up to $1.5x$.
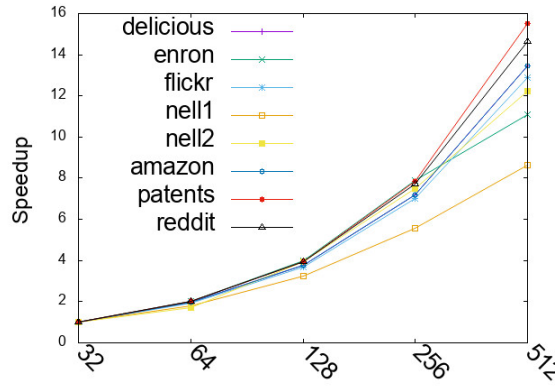
7.3   Scaling, Distribution Time and Memory

Strong scaling. We studied the scaling behavior of the HOOI procedure under the different schemes by varying the number of ranks from 32 to 512. The speedup results are reported in Figure 15. As the number of ranks increases, the average TTM load decreases, but the sizes of the large slices remain the same. As a result, CoarseG suffers from severe load imbalance and scales poorly. The other schmes scale comparatively better. The figure also includes the scaling for all ranks from 32 to 512 ranks under the Lite scheme. We see that the scheme exhibits the best scaling behavior: as against an ideal value of 16, Lite achieves speedup in the range $8.6 - 15.5x$, which translates to a scaling efficiency of $55 - 97\%$.

Distribution Time. We next evaluate the schemes on the time taken for distributing the input tensor under the configuration of $K = 10$ and 512 ranks; see Figure 16. We implemented the three lightweight schemes in parallel as part of the HOOI procedure and obtained the HyperG partitioning by executing the Zoltan library in parallel in an offline fashion. For the Lite scheme, the distribution time refers to the time spent in executing a parallel implementation of the procedure given in Figure 8. The HOOI execution time the under Lite scheme is also included for the sake of comparison. We can see that the distribution times of the three lightweight schemes are lesser or comparable to the HOOI execution time, whereas HyperG takes significantly higher time.

Speedup	CoarseG	MediumG	HyperG	Lite
delicious	7.4	6.8	8.8	13.4
enron	1.7	9.0	7.4	11.1
flickr	6.7	6.4	9.8	12.9
nell1	6.4	7.6	7.9	8.6
nell2	2.4	8.4	7.5	12.2
amazon	1.8	11.0	x	13.5
patents	2.7	14.5	x	15.5
reddit	1.8	14.2	x	14.6

(a) Speedup from 32 to 512 ranks

(b) Lite Strong scaling

Figure 15: Scaling study

Time (s)	CoarseG	MediumG	HyperG	Lite	HOOI
delicious	6.8	9.3	345	3.9	5.2
enron	0.1	0.08	125	0.1	1.1
flickr	10.9	14.0	203	5.5	6.0
nell1	10.5	13.9	356	6.2	2.7
nell2	0.07	0.05	91	0.07	0.3
amazon	2.9	5.5	x	2.5	8.7
patents	3.2	0.9	x	2.0	14.2
reddit	7.8	11.6	x	5.7	21.6

Figure 16: Distriution time

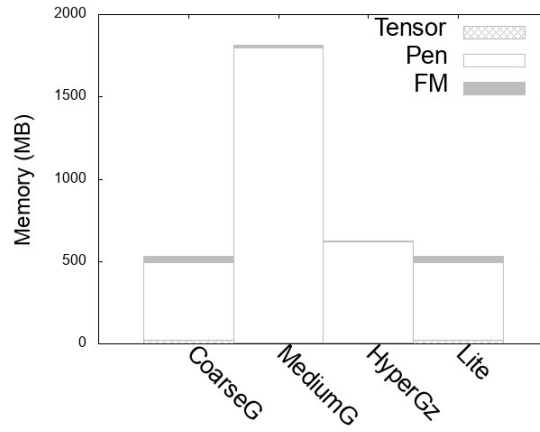| Memory (MB)	CoarseG	MediumG	HyperG	Lite
delicious	383	1748	881	385
enron	17	53	61	17
flickr	533	1813	625	533
nell1	112	197	151	113
nell2	12	6	19	12
amazon	348	371	x	350
patents	445	158	x	447
reddit	814	543	x	812

(a) Total memory

(b) delicious

(c) enron

(d) flickr

Figure 17: Memory Usage - Avergae memory (MB) per rank

Memory Usage. Th procedure needs to store the input tensor, the penultimate matrices and the factor matrices. We next evalute the above memory requirement under the different schemes. The results are shown in Figure 17. For the first three tensors, the figure also includes the breakup in terms of the three components. Being multi-policy schemes, CoarseG and Lite store N copies of the input tensor, and they do not actively minimize the factor matrix storage. However, due to low redundancy, they take lesser amount of space to store the penultimate matrices. In contrast, MediumG and HyperG store only a single copy of the tensor and are better on factor matrix storage, but due to higher redudancy, they take more space for storing the penultimate matrices. The size of the penultimate matrices get larger with increase in the number of dimension, whereas the space needed for storing the tensor increaes with the increase in density of the input tensor. We see that Lite and CoarseG require nearly the same amount of memory. They outperform MediumG and HyperG on the first three tensors, which are four dimensional. The other tensors are three dimensioanl, with the three large tensors being relatively denser. As a result, MediumG performs better on these tensors. Overall, we can see that Lite is better or comparable to MediumG and HyperG on the overall memory requirement.

8   Conclusions and Future Work

In this paper, we proposed an improved lightweight distribution scheme for the Tucker decomposition of sparse tensors. The scheme is provably near-optimal on certain fundamental metrics that determine the HOOI execution time. Our experimental evaluation demonstrates that the scheme performs well in terms of distribution time and outperforms prior schemes on the HOOI execution time by a factor of upto 3x. We identify two avenues for future work related to shared memory systems. While the new scheme is near-optimal on the TTM and the SVD components, it does not explicitly optimize the factor matrix communication volume. Since the above communication does not arise in shared memory systems, the new scheme may provide optimal strategies for partitioning work among the threads. Conversely, recent work on shared memory systems [26] has shown that the TTM computational load can be reduced using the compressed sparse fiber representation. The strategy may be useful in optimizing the TTM computations local to the processors.

References

[1] W. Austin, G. Ballard, and T. Kolda. Parallel tensor compression for large-scale scientific data. In *IPDPS*, 2016.

[2] B. Bader and T. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM J. on Sci. Comp.*, 30(1):205–231, 2007.

[3] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *HPEC*, 2012.

[4] E. Boman, K. Devine, L. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, and W. Mitchell. Zoltan home page, 1999. `http://www.cs.sandia.gov/Zoltan`.

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

[6] V. Chakaravarthy, J. Choi, D. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar. On optimizing distributed Tucker decomposition for dense tensors. In *IPDPS*, 2017.

[7] J. Choi and S. Vishwanathan. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*, 2014.

[8] R. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multimodal factor analysis. *UCLA Working Papers in Phonetics*, 1970.

[9] V. Hernandez, J. Roman, A. Tomas, and V. Vidal. Restarted Lanczos bidiagonalization for the SVD in SLEPc. *STR-8, Tech. Rep.*, 2007.

[10] W. Hightower, J. Prins, and J. Reif. Implementations of randomized sorting on large parallel machines. In *SPAA*, 1992.

[11] I. Jeon, E. Papalexakis, U. Kang, and C. Faloutsos. HaTen2: Billion-scale tensor decompositions. In *ICDE*, 2015.

[12] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries. In *KDD*, 2012.

[13] L. Karlsson, D. Kressner, and A. Uschmajew. Parallel algorithms for tensor completion in the CP format. *Parallel Computing*, 57:222–234, 2016.

[14] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *SC*, 2015.

[15] O. Kaya and B. Uçar. High performance parallel algorithms for the Tucker decomposition of sparse tensors. In *ICPP*, 2016.

[16] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *SC*, 2015.

[17] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51:455–500, 2009.

[18] T. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, 2008.

[19] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[20] L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higherorder tensors. *SIAM J. Matrix Analysis and Applications*, 21:1324–1342, 2000.

[21] N. Liu, B. Zhang, J. Yan, Z. Chen, W. Liu, F. Bai, and L. Chien. Text representation: From vector to tensor. In *ICDM*, 2005.

[22] D. Muti and S. Bourennane. Multidimensional filtering based on a tensor approach. *Signal Processing*, 85:2338–2353, 2005.

[23] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. In *ICDM*, 2014.

[24] S. Smith, J. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools. `http://frostt.io/`, 2017.

[25] S. Smith and G. Karypis. A medium-grained algorithm for sparse tensor factorization. In *IPDPS*, 2016.

[26] S. Smith and G. Karypis. Accelerating the Tucker decomposition with compressed sparse tensors. In *Euro-Par*, 2017.

[27] S. Smith, J. Park, and G. Karypis. Sparse tensor factorization on many-core processors with high-bandwidth memory. In *IPDPS*, 2017.

[28] L. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31:279–311, 1966.

[29] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[30] G. Zhou, A. Cichocki, and S. Xie. Decomposition of big tensors with low multilinear rank. *CoRR, arXiv:1412.1885*, 2015.

A Reformulation

Here, we describe how to compute the rows of the penultimate matrix. Let $u_1, u_2, \ldots, u_r$ be a sequence vectors of length $s_1, s_2, \ldots, s_r$, respectively. The Kronecker (or outer) product of the sequence is an r-dimensional tensor of size $s_1 \times s_2 \times \ldots \times s_r$, wherein the element with coordinate $(c_1, c_2, \ldots, c_r)$ takes the value $\Pi_{j=1}^r u_j[c_j]$. We represent the tensor as a vector of length $s_1 \cdot s_2 \cdot \ldots \cdot s_r$ by arranging the elements in a lexicographic order. Namely, the above value is placed in position $\sum_j c_j \left(\Pi_{i<j} \ell_i \right)$.

For an element e with coordinate $(l_1, l_2, \ldots, l_n \ldots, l_N)$ and value $\mathsf{val}(e)$, let $\mathsf{Kron}_n(e)$ denote the vector yielded by the Kronecker product of the following rows (vectors) of the factor matrices:

$$\mathbf{F}_n[l_1, :], \ldots, \mathbf{F}_{n-1}[l_{n-1}, :], \mathbf{F}_{n+1}[l_{n+1}, :], \ldots, \mathbf{F}_N[l_N, :].$$

Define $\mathsf{contr}_n(e) = \mathsf{val}(e) \cdot \mathsf{Kron}_n(e)$. For any $l \in [1, L_n]$, the row $\mathbf{Z}_{(n)}[l, :]$ is given by the summation shown in (1).