Scalable Unsupervised ML: Latency Hiding in Distributed Sparse Tensor Decomposition

Nabil Abubaker, M. Ozan Karsavuran, and Cevdet Aykanat

Abstract—Latency overhead in distributed-memory parallel CPD-ALS scales with the number of processors, limiting the scalability of computing CPD of large irregularly sparse tensors. This overhead comes in the form of sparse reduce and expand operations performed on factor-matrix rows via point-to-point messages. We propose to hide the latency overhead through embedding all of the point-to-point messages incurred by the sparse reduce and expand into dense collective operations which already exist in the CPD-ALS. The conventional parallel CPD-ALS algorithm is not amenable for embedding so we propose a computation/communication rearrangement to enable the embedding. We embed the sparse expand and reduce into a hypercube-based `ALL-REDUCE` operation to limit the latency overhead to $O(\log_2 K)$ for a K-processor system. The embedding comes with the cost of increased bandwidth overhead due to the multi-hop routing of factor-matrix rows during the embedded-`ALL-REDUCE`. We propose an embedding scheme that takes advantage of the expand/reduce properties to reduce this overhead. Furthermore, we propose a novel recursive bipartitioning framework that enables simultaneous hypergraph partitioning and subhypergraph-to-subhypercube mapping to achieve subtensor-to-processor assignment with the objective of reducing the bandwidth overhead during the embedded-`ALL-REDUCE`. We also propose a bin-packing-based algorithm for factor-matrix row to processor assignment aiming at reducing processors' maximum send and receive volumes during the embedded-`ALL-REDUCE`. Experiments on up to 4096 processors show that the proposed framework scales significantly better than the state-of-the-art point-to-point method.

Index Terms—sparse tensor, tensor decomposition, CANDECOMP/PARAFAC, canonical polyadic decomposition, latency hiding, embedded communication, communication cost, concurrent communication, recursive bipartitioning, hypergraph partitioning

1 Introduction

Tensor decomposition has emerged as a successful tool for analyzing multi-way data. Canonical polyadic (or CANDECOMP/PARAFAC) decomposition (CPD) is one of the popular tensor decompositions that extends singular value decomposition to tensors and is a fundamental tool in unsupervised learning setting [1], [2], [3], [4]. It has also become an integral part of different machine learning fields either as a method (e.g., regression [5], supervised classification [6]), or as a support tool (e.g., compression for Deep Learning [7], [8], [9]) and more [10]. CPD decomposes a tensor into its constituent rank-one tensors thus revealing latent factors to be used for data analysis.

Several algorithms exist for calculating the CPD for a given decomposition rank R, among which alternating least squares (ALS) is the most popular and used in practice. Matricized Tensor Times Khatri-Rao Product (MTTKRP) operation, which is performed to compute decomposition factor matrix for each mode, is the bottleneck operation in CPD-ALS. In distributed-memory parallel CPD-ALS, each MTTKRP operation needs sparse reduce and expand communications as well as two dense reduce communications. The sparse reduce/expand are irregular due to the sparsity pattern of the tensor and they are performed with point-to-point (P2P) messages. On the other hand, the dense reduce communications involve data of sizes R and R^2 which are required by all processors and thus are performed using the collective `ALL-REDUCE` operation from the MPI primitives.

• *Authors are with the Department of Computer Engineering, Bilkent University, Ankara, Turkey.*

The bandwidth overhead of MTTKRP scales with both tensor size and decomposition rank, whereas latency overhead increases with increasing number of processors as well as with increasing irregularity in the sparsity pattern of the tensor. That is, CPD-ALS becomes latency bound for small decomposition rank values. Although current distributed-memory parallel CPD-ALS algorithms, which utilize P2P communication scheme [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], scale well up to a certain number of processors, these algorithms fail to scale after some number of processors. We empirically find this number to be around 512–1024 processors as also reported in [12], [19]. Thus, optimizing the latency overhead is a key point for scaling CPD-ALS for large number of processors.

In this work, we propose hiding the latency overhead of sparse expand and reduce operations of CPD-ALS by embedding them into `ALL-REDUCE`. Although CPD-ALS has an `ALL-REDUCE` for each sparse expand and reduce communication, it is not possible to embed each sparse expand/reduce due to the dependencies between the sparse operations and `ALL-REDUCE`. We propose a novel computation/communication rearrangement scheme of the CPD-ALS that removes the dependencies and enables embedding each of the sparse expand/reduce operations into an `ALL-REDUCE`.

We use the hypercube-based `ALL-REDUCE` which utilize the E-cube routing for embedding and we denote the embedding scheme by EMB hereafter. The utilized hypercube topology is virtual and transparent to the actual network topology of the target system. In the naive implementation of EMB, each individual P2P message of sparse expand/reduce operation is considered separately. This may

lead to multiple copies of the same expanded/reduced factor-matrix row be in the same message between two processors during the embedded-ALL-REDUCE. Therefore, we propose an expand and reduce aware embedding in which each message contains only one copy of a factor-matrix row in each step of ALL-REDUCE. We also extend the communication duality between sparse reduce and expand operations into EMB by proposing to use increasing dimension E-cube routing during the expand-embedded-ALL-REDUCE, while using decreasing dimension E-cube routing during reduce-embedded-ALL-REDUCE, or vice versa.

The proposed EMB totally avoids the latency overhead associated with the sparse expand and reduce operations and reduces both maximum and average number of messages handled by a processor to $2\log_2 K$ for each MTTKRP for a K-processor system independent of the sparsity pattern of the tensor. The only trade-off between the proposed EMB and conventional P2P schemes is the increase in the communication volume incurred by embedding the P2P communications into the ALL-REDUCE communications.

In order to model the communication requirement of EMB, we define a concurrent communication cost metric which counts how many times each shared factor-matrix row is concurrently communicated along hypercube dimensions during the E-cube routing. Then we propose a novel recursive bipartitioning (RB) framework that enables simultaneous hypergraph partitioning (HP) and subhypergraph-to-subhypercube mapping to achieve task-to-processor assignment which encodes minimizing the concurrent communication volume metric. In this HP model, we propose and use sibling subnet removal and net-anchoring schemes at each level of RB. We also propose a novel bin-packing adaptation for the factor-matrix row to processor assignment in order to minimize the maximum volume handled by a processor during both expand-embedded and reduce-embedded ALL-REDUCE operations. The proposed extension of duality to EMB enables the proposed bin-packing to encode the minimization of maximum volume for only one sparse embedding which holds for the other.

Experimental results with ten tensors on up to 4096 processors show the validity of the proposed models and methods. These results show that EMB scales well up to 4096 processors, whereas state-of-the-art P2P scales down after 1024 processors.

The rest of the paper is organized as follows: Sec. 2 contains the background material. The proposed rearrangement scheme that enables embedding is discussed in Sec. 3. Sec. 4 presents the proposed embedding scheme. The proposed RB-based HP model for task-to-processor assignment is described in Sec. 5. Sec. 6 displays and discusses the experimental results. The related work is given in Sec. 7. Finally, Sec. 8 concludes the paper.

2 Background

2.1 Tensors, Notations and CPD

A tensor is denoted by calligraphic ($\mathcal{X}$) while matrices by bold capital ($\mathbf{U}$) letters. An M-mode tensor has M dimensions $I_1, I_2, \cdots, I_M$ and can be unfolded into a matrix shape along one of its modes. This is called matricization and a matricized tensor is of size $I_m \times I_1 \cdots I_{m-1} I_{m+1} \cdots I_M$ and denoted by $\mathbf{X}^{(m)}$ where $m \in [1..M]$.

The CPD decouples a tensor into R rank-1 components as $\mathcal{X} \approx \sum_{i=1}^{R} \mathcal{X}_i$, where rank-1 component $\mathcal{X}_i$ is the outer product of M vectors $\mathbf{u}_i^{(1)} \circ \mathbf{u}_i^{(2)} \circ \cdots \circ \mathbf{u}_i^{(M)}$. The R vectors along mode m are combined to form a factor matrix $\mathbf{U}^{(m)} \in \mathbb{R}^{I_m \times F}$ along mode m. Here, R is called the decomposition rank. A row in $\mathbf{U}^{(m)}$ is referred to as r_i^m. When the mode of the tensor is irrelevant to the discussion, we use r_i to refer to a row in a factor matrix along any mode.

The goal of an algorithm computing the CPD is to find the best approximation of a tensor $\mathcal{X}$ using R components that minimizes a norm of $\mathcal{X} - \sum_{i=1}^{R} \lambda_i \mathcal{X}_i$. Here, the vectors used to construct $\mathcal{X}_i$ are normalized to length 1, and the value λ_i is used as a scaling factor to the normalized $\mathcal{X}_i$ rank-1 component tensor. The matricized tensor along mode m can be approximated as the product $\mathbf{U}^{(m)}(\odot_{i \neq m} \mathbf{U}^{(i)})^\top$ where $\odot$ denotes a Khatri-Rao product. Using ALS, $\mathbf{U}^{(m)}$ is calculated by fixing the other $M-1$ factor matrices and solving for $\mathbf{U}^{(m)}$. The formulation to compute $\mathbf{U}^{(m)}$ can be given as $\mathbf{X}^{(m)}(\odot_{i \neq m} \mathbf{U}^{(i)})(*_{i \neq m} \mathbf{U}^{(i)\top} \mathbf{U}^{(i)})^{-1}$, where $*$ denotes the Hadamard product. The term $\mathbf{X}^{(m)}(\odot_{i \neq m} \mathbf{U}^{(i)})$ is the MTTKRP operation. We refer the reader to the excellent survey by Kolda and Bader [21] for a more comprehensive coverage of the CPD and its computation.

2.2 Parallel CPD-ALS

We adopt nonzero-based parallelization of CPD-ALS. In this parallelization, tensor nonzeros are distributed among processors and processors locally compute (partial) results for factor matrices using those nonzeros according to the owner-computes rule. For processor p_k, the factor matrix rows are classified into three categories according to the tensor nonzeros distribution as follows: Factor-matrix row r_i is said to be local if the nonzeros that contribute to its computation reside in p_k. r_i is said to be local-shared if the nonzeros that contribute to its computation reside in a set of sharing processors $\hat{S}_i \subseteq P, |\hat{S}_i| > 1 \wedge p_k \in \hat{S}_i$, and the processor responsible for holding the final value of r_i is p_k. In such case, p_k is called the owner of r_i and denoted by $owner(r_i)$. We use $S_i = \hat{S}_i \setminus owner(r_i)$ to identify the set of sharing processors without the owner. A local factor matrix that contains local and local-shared rows is denoted by $\mathbf{U}_k^{(m)}$. r_i is said to be nonlocal if p_k has one or more nonzeros that contribute to its computation but p_k is not its owner. A local factor matrix that contains $\mathbf{U}_k^{(m)}$ in addition to nonlocal rows is distinguished by the hat as $\hat{\mathbf{U}}_k^{(m)}$.

We use 3-mode tensors here and in Sec. 3 for a convenient presentation. The discussions easily extend to higher dimensional tensors (i.e., $M > 3$). Algorithm 1 describes the parallel CPD-ALS for 3-mode tensors. In the algorithm, $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ respectively represent $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}$ and $\mathbf{U}^{(3)}$. The communication requirement in this algorithm is detailed for updating $\mathbf{A}$ per processor p_k as follows. After the local MTTKRP (line 3), partial results of local-shared factor matrix rows are received while partial results of nonlocal rows are sent to their owner processors. The received partial results are reduced using an associative operation to form the up-to-date local-shared rows. This communication operation is referred to as *sparse reduce*. Using the up-to-date local and

Algorithm 1 Parallel CPD-ALS ($\mathcal{X}$) for 3-mode Tensors

1: Randomly initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
2: **while** not converged **do**
3: $\hat{\mathbf{A}}'_k \leftarrow \mathbf{X}^{(1)}_k(\hat{\mathbf{B}}_k \odot \hat{\mathbf{C}}_k)$ ▷ MTTKRP
4: Sparse REDUCE on shared $\mathbf{A}$-matrix rows
5: $\mathbf{A}_k \leftarrow \mathbf{A}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{B}^\top\mathbf{B})^{-1}$
6: ALL-REDUCE to normalize cols of $\mathbf{A}$ into λ
7: Sparse EXPAND on shared $\mathbf{A}$-matrix rows
8: ALL-REDUCE to compute $\mathbf{A}^\top\mathbf{A}$
9: $\hat{\mathbf{B}}'_k \leftarrow \mathbf{X}^{(2)}_k(\hat{\mathbf{A}}_k \odot \hat{\mathbf{C}}_k)$ ▷ MTTKRP
10: Sparse REDUCE on shared $\hat{\mathbf{B}}$-matrix rows
11: $\mathbf{B}_k \leftarrow \mathbf{B}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{A}^\top\mathbf{A})^{-1}$
12: ALL-REDUCE to normalize cols of $\mathbf{B}$ into λ
13: Sparse EXPAND on shared $\mathbf{B}$-matrix rows
14: ALL-REDUCE to compute $\mathbf{B}^\top\mathbf{B}$
15: $\hat{\mathbf{C}}'_k \leftarrow \mathbf{X}^{(3)}_k(\hat{\mathbf{A}}_k \odot \hat{\mathbf{B}}_k)$ ▷ MTTKRP
16: Sparse REDUCE on shared $\hat{\mathbf{C}}$-matrix rows
17: $\mathbf{C}_k \leftarrow \mathbf{C}'_k(\mathbf{B}^\top\mathbf{B} * \mathbf{A}^\top\mathbf{A})^{-1}$
18: ALL-REDUCE to normalize cols of $\mathbf{C}$ into λ
19: Sparse EXPAND on shared $\mathbf{C}$-matrix rows
20: ALL-REDUCE to compute $\mathbf{C}^\top\mathbf{C}$
21: **return** $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$

local-shared factor matrix values, the product in line 5 can be computed locally. Then, column normalization requires computing λ that depends on all factor matrix columns through ALL-REDUCE in line 6. The normalized local-shared row r_i is needed by the processors in S_i for the computation of the factor matrix along the next tensor mode. Therefore, the local-shared rows are sent (expanded) to and the nonlocal rows are received from their respective owner processors (line 7). This operation is referred to as *sparse expand*. Finally, the partial $\mathbf{A}^\top\mathbf{A}$ product can be computed locally using local and local-shared rows and an ALL-REDUCE operation is used for computing the final product (line 8).

2.3 Hypergraph Partitioning (HP) Problem

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as the set $\mathcal{V}$ of vertices and the set $\mathcal{N}$ of nets. Each net n connects a subset of vertices denoted by $Pins(n)$. Each vertex v is assigned a weight and each net n is assigned a cost $c(n)$.

Let $\Pi(\mathcal{H}) = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ denote a K-way vertex partition of $\mathcal{H}$. The weight $W(\mathcal{V}_k)$ of part $\mathcal{V}_k$ in Π is defined as the sum of the weights of the vertices in $\mathcal{V}_k$. $\Pi(\mathcal{H})$ satisfies the partitioning constraint if $W(\mathcal{V}_k) \leq W_{avg}(1+\epsilon)$ for each part $\mathcal{V}_k$ in Π, for a given maximum allowed imbalance ratio ϵ. Here W_{avg} denotes the average part weight.

In a given partition $\Pi(\mathcal{H})$, net n is said to connect part $\mathcal{V}_k$ if it connects at least one vertex in $\mathcal{V}_k$. The connectivity set of net n, $Con(n)$, is defined as the set of parts connected by n. The connectivity of n, $con(n)$, denotes the number of parts connected by n. Net n is said to be cut if $con(n) > 1$, and uncut otherwise. Then the connectivity cutsize is defined as $cutsize(\Pi) = \sum_{n \in \mathcal{N}}(con(n) - 1)c(n)$. In HP, the partitioning objective is to minimize the cutsize while maintaining the partitioning constraint. In HP with fixed vertices, part assignment of some vertices are given priori to partitioning.

Algorithm 2 Rearranged Parallel CPD-ALS ($\mathcal{X}$) for 3-mode Tensors

1: Randomly initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
2: **while** not converged **do**
3: $\hat{\mathbf{A}}'_k \leftarrow \mathbf{X}^{(1)}_k(\hat{\mathbf{B}}_k \odot \hat{\mathbf{C}}_k)$ ▷ MTTKRP
4: Sparse REDUCE on shared $\mathbf{A}$-matrix rows
5: ALL-REDUCE to compute $\mathbf{C}^\top\mathbf{C}$
6: $\mathbf{A}_k \leftarrow \mathbf{A}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{B}^\top\mathbf{B})^{-1}$
7: $\lambda'_c \leftarrow \langle \mathbf{A}_k(:,c), \mathbf{A}_k(:,c) \rangle,\ \forall c \in [1..R]$
8: ALL-REDUCE to compute λ'
9: Sparse EXPAND on shared $\mathbf{A}$-matrix rows
10: $\lambda_r \leftarrow \sqrt{\lambda'_c},\ \forall c \in [1..R]$
11: $\hat{\mathbf{A}}_k(:,c) \leftarrow \hat{\mathbf{A}}_k(:,c)/\lambda_c,\ \forall c \in [1..R]$
12: $\hat{\mathbf{B}}'_k \leftarrow \mathbf{X}^{(2)}_k(\hat{\mathbf{A}}_k \odot \hat{\mathbf{C}}_k)$ ▷ MTTKRP
13: Sparse REDUCE on shared $\mathbf{B}$-matrix rows
14: ALL-REDUCE to compute $\mathbf{A}^\top\mathbf{A}$
15: $\mathbf{B}_k \leftarrow \mathbf{B}'_k(\mathbf{C}^\top\mathbf{C} * \mathbf{A}^\top\mathbf{A})^{-1}$
16: $\lambda'_c \leftarrow \langle \mathbf{B}_k(:,c), \hat{\mathbf{B}}_k(:,c) \rangle,\ \forall c \in [1..R]$
17: ALL-REDUCE to compute λ'
18: Sparse EXPAND on shared $\mathbf{B}$-matrix rows
19: $\lambda_c \leftarrow \sqrt{\lambda'_c},\ \forall c \in [1..R]$
20: $\hat{\mathbf{B}}_k(:,c) \leftarrow \hat{\mathbf{B}}_k(:,c)/\lambda_c,\ \forall c \in [1..R]$
21: $\hat{\mathbf{C}}'_k \leftarrow \mathbf{X}^{(3)}_k(\hat{\mathbf{A}}_k \odot \hat{\mathbf{B}}_k)$ ▷ MTTKRP
22: Sparse REDUCE on shared $\mathbf{C}$-matrix rows
23: ALL-REDUCE to compute $\mathbf{B}^\top\mathbf{B}$
24: $\mathbf{C}_k \leftarrow \mathbf{C}'_k(\mathbf{B}^\top\mathbf{B} * \mathbf{A}^\top\mathbf{A})^{-1}$
25: $\lambda'_c \leftarrow \langle \hat{\mathbf{C}}_k(:,c), \hat{\mathbf{C}}_k(:,c) \rangle,\ \forall c \in [1..R]$
26: ALL-REDUCE to compute λ'
27: Sparse EXPAND on shared $\mathbf{C}$-matrix rows
28: $\lambda_c \leftarrow \sqrt{\lambda'_c},\ \forall c \in [1..R]$
29: $\hat{\mathbf{C}}_k(:,c) \leftarrow \hat{\mathbf{C}}_k(:,c)/\lambda_c,\ \forall c \in [1..R]$
30: **return** $[\![\lambda; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$

3 Rearrangement of Parallel CPD-ALS to Enable Embedding

In the parallel CPD-ALS shown in Algorithm 1, there are two sparse reduce and expand operations per tensor mode to satisfy the computational requirement of the MTTKRP operation. The dual sparse reduce and expand operations (respectively in lines 4 and 7) are performed to complete the computation of local and local-shared $\mathbf{A}$-matrix rows. Similarly, the dual sparse reduce and expand in lines 10, 13 and lines 16, 19 do so respectively for $\mathbf{B}$- and $\mathbf{C}$-matrix rows. Furthermore, there are two ALL-REDUCE operations attached with the computation of factor matrices along each mode. Despite having an ALL-REDUCE for each sparse expand/reduce, it is not possible to embed each sparse expand/reduce in the current form of Algorithm 1. This is due to the dependencies of the two ALL-REDUCE operations in lines 6 and 8 to the sparse reduce in line 4. That is, the sparse reduce cannot be embedded into the ALL-REDUCE in line 6 because the $\mathbf{A}_k$ rows, which are computed in line 5, are required for the computation of λ. Furthermore, the sparse expand in line 7 cannot be embedded into the ALL-REDUCE in line 6 because distributed column normalization need to be performed before the expand. On the other hand,

the sparse expand can be embedded into the ALL−REDUCE in line 8. Although embedding the sparse expand alone is important, it is insufficient for hiding latency since the sparse reduce, performed as P2P, will still be a bottleneck due to the high number of messages.

We propose to rearrange the computation and communication steps in Algorithm 1 to enable the embedding of all sparse expand/reduce operations without any dependency issues. We highlight two important observations that facilitate the rearrangements for successful embedding.

First observation: It is possible to expand non-normalized $\mathbf{A}_k$-matrix rows just after the operation in line 5, and then normalize $\hat{\mathbf{A}}_k$-matrix rows. In other words, instead of expanding normalized local-shared $\mathbf{A}_k$ rows, which requires the λ vector to be ready in advance, the non-normalized local-shared $\mathbf{A}_k$ rows are expanded while computing global λ using ALL−REDUCE. The extra cost here is that each processor will take the responsibility of normalizing nonlocal rows in addition to local and local-shared rows. With this observation the dependency between the ALL−REDUCE (line 6) and the sparse expand (line 7) can be removed, allowing the latter operation to be embedded into the former. The same argument applies to the normalization of $\mathbf{B}$- and $\mathbf{C}$-matrix columns in lines 12 and 18, respectively.

Second observation: The $\mathbf{A}^\top\mathbf{A}$ product (line 8 of Algorithm 1) is not required until the operation in line 11. The associated ALL−REDUCE neither has dependency to the sparse expand in line 7 nor to the sparse reduce in line 10, thus it can be used to embed the sparse reduce of *the next mode*. Similar discussion holds for the ALL−REDUCE associated with $\mathbf{B}^\top\mathbf{B}$. The $\mathbf{C}^\top\mathbf{C}$ product (line 20) is not required until the operation in line 5 *of the next iteration* The associated ALL−REDUCE neither has dependency to the sparse expand in line 19 nor to the sparse reduce in line 4 of the next iteration, and therefore it can be placed anywhere between line 19 of the current iteration to before the operation in line 5 of the next iteration. This inter-mode and inter-iteration rearrangement is similar to the *software pipelining* used in compiler design and operating systems.

Algorithm 2 shows the rearranged version of Algorithm 1. Lines 7, 8, 10, 11 of Algorithm 2 realize the column normalization of matrix $\mathbf{A}$, performed in line 6 of Algorithm 1, utilizing the first observation. In a similar way, lines 16, 17, 19, 20 and 25, 26, 28, 29 realize the column normalization of $\mathbf{B}$ and $\mathbf{C}$, respectively. The ALL−REDUCE operation for computing $\mathbf{A}^\top\mathbf{A}$ is shifted forward to be a neighbor to the sparse reduce of the second mode (lines 13, 14). The same applies to $\mathbf{B}^\top\mathbf{B}$ and the sparse reduce of the third mode (lines 22, 23). On the other hand, $\mathbf{C}^\top\mathbf{C}$ is shifted to be a neighbor to the sparse reduce of the first mode in the next CPD-ALS iteration. The highlighted boxes show the sparse operations to be embedded in the preceding/following ALL−REDUCE. Since there are two sparse reduce and expand operations per tensor mode, the rearranged algorithm shows six boxes to indicate that all sparse operations are to be embedded for 3-mode tensors.

4 Embedding Sparse Expand and Reduce

In order to realize the sparse expand and reduce operations using P2P messages, processor p_x should maintain two processor sets: workers set (WS) and masters set (MS) respectively defined as

$$\text{WS}(p_x) = \bigcup_{i \ni r_i \text{ is local-shared}} S_i,$$
$$\text{MS}(p_x) = \{owner(r_i) \mid r_i \text{ is nonlocal}\}.$$

That is, WS(p_x) contains the processors that contribute to the computation of any row that p_x owns, whereas MS(p_x) contains the processors that p_x is partially contributing to the computation of a row they own. Then, a sparse expand (reduce) on row r_i is achieved as messages from (to) p_x to (from) every processor in WS(p_x)(MS(p_x)).

4.1 Naive P2P Embedding

The hypercube-based ALL−REDUCE can be performed in $\log_2 K$ steps for a system with $K = 2^D$ processors. The K processors are virtually organized as a D-dimensional hypercube topology H. In H, each processor is represented by a D-bit binary number. We interchangeably use p_x to refer both index of a processor and its D-bit binary representation. Two processors are said to be neighbors along dimension i if their binary representation differ only in least significant bit i. In a D-dimensional hypercube, a d-dimensional subcube ($0 \leq d < D$) is represented by d don't care bits (X) and $D-d$ fixed $0/1$ bits thus having 2^d processors. Tearing along dimension i is defined as halving H into two disjoint $(D-1)$-dimensional subcubes such that the processors in the two sets are identified by the ith bit. For example, a tearing along dimension $i=1$ on processor set P_{XXXX} organized as a 4-dimensional hypercube can be shown by processor sets P_{XX0X} and P_{XX1X}. The hypercube-based ALL−REDUCE is well known and comes with several names such as E-cube routing, bidirectional exchange and exchange-add [22], [23], [24]. We adopt this ALL−REDUCE scheme and we use $\mathfrak{R}(H)$ to refer to it hereafter. A step s_i of $\mathfrak{R}(H)$ represents the exchange of messages between neighboring processors along dimension i.

The naive embedding of P2P into ALL−REDUCE utilizing $\mathfrak{R}(H)$ is described as follows: A message $m(p_x, p_z)$ originating from p_x is sent from p_x to the neighbor at dimension i where i is the position of the least significant 1 bit in the XOR product $p_x \oplus p_z$. If the neighbor p_y at dimension i is the destination processor ($p_y = p_z$), then $m(p_x, p_z)$ is received and need not to be in any exchange in any upcoming step. Otherwise, p_y stores $m(p_x, p_z)$ in a forward buffer and sends it to its neighbor at dimension $j > i$, where j is the position of the least significant 1 bit in $p_y \oplus p_z$. A message is guaranteed to arrive to its destination in at most D steps.

4.2 Expand-and-Reduce-Aware Embedding

Consider expanding a local-shared factor matrix row r_i from p_0 to p_3 and p_5. In the naive EMB implementation, this expand consists of two different messages $m(p_0, p_3)$ and $m(p_0, p_5)$. Using $\mathfrak{R}(H)$, these messages will respectively take the routes $p_0 \rightarrow p_1 \rightarrow p_3$ and $p_0 \rightarrow p_1 \rightarrow p_5$. This means that r_i is sent (forwarded) twice in the message from p_0 to p_1. In general, a message between processor p_x and its neighbor p_y in any step can contain up to $D-1$ duplicates of the same row r_i. This is because the naive

EMB described in Sec. 4.1 is unaware of the nature of the sparse expand and reduce. We can reduce the increase in the communication volume in EMB by exploiting the nature of the sparse expand and reduce operations via avoiding transmitting the same row more than once in a message between hypercube neighbors.

We propose an intelligent expand-and-reduce-aware EMB that avoids transmitting more than one copy of any row between hypercube neighbors as follows: During an embedded sparse expand, multiple copies of row r_i at step s of $\mathfrak{R}(H)$ are sent only once. During an embedded sparse reduce, multiple copies of row r_i at step s of $\mathfrak{R}(H)$ are reduced locally, and then sent as one copy. So, the reduce on r_i in the intelligent EMB is done during the routing steps of $\mathfrak{R}(H)$, whereas in naive EMB it is done at the receiving end by $owner(r_i)$ when all reduce messages are received.

4.3 Communication Duality in Embedding

In CPD-ALS, each shared factor-matrix row r_i is reduced from processors in S_i to $owner(r_i)$ and then the updated r_i (through local operations) is expanded from the same $owner(r_i)$ to the same set of processors S_i. That is, the same set of processors contribute to and need row r_i. We call such reduce and expand operations as dual communications.

In the P2P implementation, dual communications incur dual communication patterns. That is, if processor p_x sends r_i to p_y in the reduce communication, p_x will receive r_i from p_y in the expand communication. This means that the maximum expand send volume is equal to the maximum reduce receive volume. The same holds for maximum expand receive and maximum reduce send volumes.

We extend the duality definition of the P2P implementation to the EMB implementation as follows: The embeddings Γ_e and Γ_r of dual P2P expand/reduce are said to be dual if for each send message at step s_i of Γ_e, there exists a step s_j of Γ_r which involves a receive message with the same constituent rows, and vice versa. This duality ensures that the maximum send/receive volumes at step s_i of Γ_e are equal to the maximum receive/send volumes at step s_j of Γ_r, and both Γ_e and Γ_r incur the same amount of communication, including the forwarding overhead due to message routing.

According to the definition of duality in EMB, if both embeddings Γ_r and Γ_e utilize the $\mathfrak{R}(H)$ routing then they are not dual. Here we propose an EMB implementation that satisfies the duality definition and attains the nice properties of the dual reduce-and-expand communications. As the E-cube routing algorithm $\mathfrak{R}(H)$ defined earlier proceeds in increasing dimension order, we then define an inverse routing algorithm $\mathfrak{R}^{-1}(H)$ that proceeds in decreasing dimension order. That is, in step s_i of $\mathfrak{R}(H)$ neighboring processors exchange messages along dimension i, whereas in step s_i of $\mathfrak{R}^{-1}(H)$ processors exchange messages along dimension $D-i-1$, for $i = 0, \cdots, D-1$. The following theorem shows duality in the proposed EMB implementation.

Theorem 1. *Utilizing $\mathfrak{R}(H)$ for embedding P2P sparse expand and $\mathfrak{R}^{-1}(H)$ for embedding a dual P2P sparse reduce (or vice versa) incurs dual embedded expand and reduce.*

Proof. In $\mathfrak{R}(H)$, each message $m(p_x, p_z)$ of the P2P expand of row r_i from $p_x = owner(r_i)$ to $p_z \in S_i$ routes through a certain path $\rho = p_x \rightarrow \cdots \rightarrow p_y \rightarrow \cdots \rightarrow p_z$. By the definition of $\mathfrak{R}(H)$ and $\mathfrak{R}^{-1}(H)$, a message $m(p_z, p_x)$ of the dual P2P reduce of row r_i follows the same path with reverse order $\rho^{-1} = p_z \rightarrow \cdots \rightarrow p_y \rightarrow \cdots \rightarrow p_x$. This means that for each expanded row in the message from p_y to its neighbor p_t in step s of $\mathfrak{R}(H)$, there is a dual reduced row in the message from p_t to p_y in step $D-s-1$ of $\mathfrak{R}^{-1}(H)$. Therefore, the constituent rows of the message from p_y to its neighbor p_t in step s of $\mathfrak{R}(H)$ are the same as those in the message from p_t to p_y in step $D-s-1$ of $\mathfrak{R}^{-1}(H)$. □

Duality in the EMB implementation, as well as in the P2P implementation, of expand and reduce is pivotal in reducing the problem size for intelligent partitioning models that encode decreasing communication cost metrics. Furthermore, the duality in EMB enables halving the storage overhead required for routing the data. That is, without the duality property there will be an explicit need for separate forward buffers during embedded expand and reduce operations.

5 Task-to-Processor Mapping

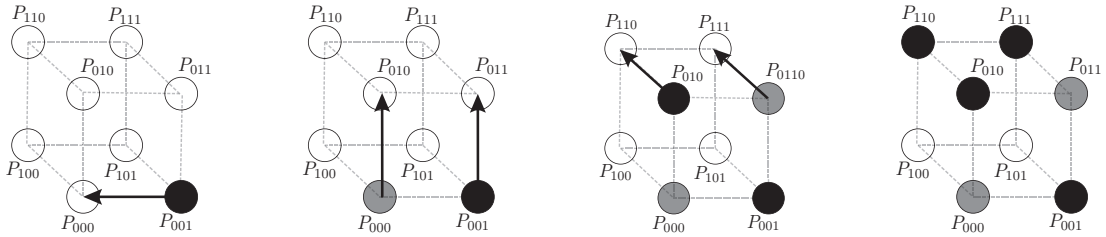
The objective in the proposed task partitioning and mapping is to minimize the communication volume overhead incurred by the embedding of the P2P communications into `ALL-REDUCE`. For this purpose, we define a communication cost metric which is set as the sum of the concurrent communication volume incurred by each shared factor-matrix row in EMB. In this concurrent communication cost metric, possibly multiple communications incurred by the same shared matrix row along the same dimension are counted as one. We preferred this communication cost metric in order to capture some form of volume concurrency involved in the expand and reduce operations associated with the shared factor-matrix rows during the `ALL-REDUCE` operations.

Fig. 1 shows a sample expand incurred by a shared factor-matrix row r_i from $owner(r_i) = p_2$ to $S_i = \{p_2, p_6, p_7\}$ for E-cube routing on a 3-dimensional hypercube. The gray processors denote the intermediate processors which do not need r_i but involve in expanding r_i in EMB. In the figure, two communication operations along dimension two contributes only one to the concurrent communication volume. Then concurrent communication volume is three.

5.1 Hypergraph Model

We propose a hypergraph model $\mathcal{H}$ to assign atomic tasks to the processors for reducing concurrent communication volume metric of EMB. In this hypergraph model, vertices represent atomic tasks, whereas nets represent factor-matrix rows. Here atomic tasks may refer to individual tensor nonzeros as well as disjoint nonzero clusters. The former case corresponds to the fine-grain [12], [20] tensor partitioning, whereas the latter case corresponds to the medium-grain [14], [19] tensor partitioning. Each vertex is associated with a weight equal to the number of nonzeros it represents and each net is associated with a cost of R.

In this hypergraph, consider a net n_i^m representing factor matrix row r_i^m along mode m. Then, pins of this net represent the set of atomic tasks that contribute to the computation of r_i^m during the MTTKRP operation along mode m. During the MTTKRP operations along each other



(a) Step 0: concurrent volume=1 total volume=1

(b) Step 1: concurrent volume=1 total volume=2

(c) Step 2: concurrent volume=1 total volume=2

(d) Expand done

Fig. 1. A sample expand operation for a row r_i from the $owner(r_i) = p_1$ to $S_i = \{p_2, p_6, p_7\}$ in the embedded communication with E-cube routing.

mode, the pins of this net represent the set of atomic tasks that need r_i^m for their associated computations along that mode. Thus, n_i^m can be considered as encoding reduce type of communication along mode m, whereas encoding expand type of communication along all other modes.

In a given partition $\Pi(\mathcal{H})$, if net n_i^m is internal to part $\mathcal{V}_k$ then row r_i^m is local to part/processor $\mathcal{V}_k/p_k$ since all atomic tasks that contribute to and use that factor-matrix row are assigned to that part/processor. If net n_i^m is cut, then row r_i^m becomes a shared row so that r_i^m is local-shared for the processor $owner(r_i^m)$, whereas it is nonlocal for the processors in $S_i = Con(n_i^m) \backslash owner(r_i^m)$.

For a cut net n_i^m, its connectivity set $Con(n_i^m) = \hat{S}_i$ denotes the set of processors that produce partial results for r_i^m during the MTTKRP operation along mode m. $\hat{S}_i$ also denotes set of processors that need r_i^m during MTTKRP operations along all other modes. Thus, in the former case, cut net n_i^m will incur reduce communication from the set of processors in S_i to the processor $owner(r_i^m)$, whereas it will incur expand communication from the processor $owner(r_i^m)$ to processors in S_i.

In this HP model, the partitioning constraint of maintaining balance among part weights encodes the computational load balance during each MTTKRP. For P2P, the partitioning objective of minimizing the cutsize encodes minimizing the sum of the total communication volume along all MTTKRP operations. In the following subsection, we describe the proposed RB-based model for many-to-one task mapping that considers reducing concurrent communication volume incurred by the shared rows in EMB.

5.2 Recursive-Bipartitioning Scheme

In the RB-based HP, the given hypergraph is bipartitioned into two vertex parts which induce two subhypergraphs. Then these two hypergraphs are further bipartitioned recursively until K vertex parts are obtained. Each subtensor corresponding to a vertex part at the last (leaf) level is assigned to a different processor. Here, without loss of generality, we assume that the number K of processors is an exact power of 2. This procedure produces a complete binary tree with $\log_2 K$ levels which is referred to as the RB tree. The RB levels are denoted as $d = 0, \cdots, \log_2 K - 1$, where $d = 0$ denotes the root (bipartitioning of the original hypergraph) and $d = \log_2 K - 1$ denotes the last internal level containing $K/2$ subhypergraphs. 2^d hypergraphs in the dth level are denoted by $\mathcal{H}_1^d, \ldots, \mathcal{H}_{2^d}^d$ from left to right for $0 \leq d < \log_2 K$. Note that the RB tree is constructed utilizing the breadth-first bipartitioning order.

The conventional RB-based HP framework utilizes the cut net splitting technique [25] after each RB step to encode connectivity cutsize metric in the K-way partition to be obtained at the end. Consider a bipartition $\Pi_2(\mathcal{H}) = \{\mathcal{V}_0, \mathcal{V}_1\}$ obtained in a particular RB step. Then this vertex bipartition is encoded as constituting subhypergraphs $\mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ and $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$ that are respectively induced by vertex parts $\mathcal{V}_0$ and $\mathcal{V}_1$. That is, $\mathcal{N}_0$ and $\mathcal{N}_1$ respectively contain the internal nets of $\mathcal{V}_0$ and $\mathcal{V}_1$ as well as the splitted subnets of the cut nets in $\mathcal{V}_0$ and $\mathcal{V}_1$. Each cut net n_i is splitted as n_i' with $Pins(n_i') = Pins(n_i) \cap \mathcal{V}_0$ and n_i'' with $Pins(n_i'') = Pins(n_i) \cap \mathcal{V}_1$ to the $\mathcal{H}_0$ and $\mathcal{H}_1$, respectively. These vertex-parts/subhypergraphs $\mathcal{V}_0/\mathcal{H}_0$ and $\mathcal{V}_1/\mathcal{H}_1$ are also called as left and right parts/hypergraphs, respectively.

The RB steps are encoded as subtensor/subhypergraph-to-subcube mappings as follows: The root of the RB tree corresponds to hypergraph $\mathcal{H}_0^0$ representing the given tensor, which is initially mapped to whole hypercube $P_{\text{x}\cdots\text{x}}$. At level $d = 0$, bipartitioning $\mathcal{H}_0^0$ into subhypergraphs $\mathcal{H}_0^1$ and $\mathcal{H}_1^1$ is encoded as mapping the subtensors represented by $\mathcal{H}_0^1$ and $\mathcal{H}_1^1$ respectively to the subcubes $P_{\text{x}\cdots\text{x}0}$ and $P_{\text{x}\cdots\text{x}1}$ of hypercube $P_{\text{x}\cdots\text{x}}$. At level $d = 1$, bipartitioning $\mathcal{H}_0^1$ into $\mathcal{H}_0^2$ and $\mathcal{H}_1^2$ is encoded as mapping the subtensors represented by $\mathcal{H}_0^2$ and $\mathcal{H}_1^2$ respectively to the subcubes $P_{\text{x}\cdots\text{x}00}$ and $P_{\text{x}\cdots\text{x}10}$ of hypercube $P_{\text{x}\cdots\text{x}0}$; and bipartitioning $\mathcal{H}_1^1$ into $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$ is encoded as mapping the subtensors represented by $\mathcal{H}_2^2$ and $\mathcal{H}_3^2$ respectively to the subcubes $P_{\text{x}\cdots\text{x}01}$ and $P_{\text{x}\cdots\text{x}11}$ of hypercube $P_{\text{x}\cdots\text{x}1}$. These two bipartitioning and mapping operations together corresponds to tearing hypercube along dimension $d = 1$. That is, $P_{\text{x}\cdots\text{x}00} \cup P_{\text{x}\cdots\text{x}01} = P_{\text{x}\cdots\text{x}0\text{x}}$ and $P_{\text{x}\cdots\text{x}10} \cup P_{\text{x}\cdots\text{x}11} = P_{\text{x}\cdots\text{x}1\text{x}}$. This process is repeated at each level of the RB tree. Fig. 2a shows simultaneous bipartitioning/mapping for a 3-dimensional hypercube. The RB-levels 0, 1 and 2 in the figure, respectively correspond to the tearing of the hypercube shown in Fig. 1 along dimensions 0, 1 and 2.

In order to encode the objective of concurrent communication volume minimization mentioned earlier, we utilize the above-mentioned recursive bipartitioning and mapping framework for modifying and enhancing the conventional cut net splitting scheme. The proposed enhancement is performed among the subnets of the same net within a same level, whereas conventional cut net splitting is continued to be applied across levels.

Consider the case where the subhypergraphs at a par-

ticular RB-level d contains multiple subnets (splitted nets) $n'_i, n''_i, \cdots, n_i^{\prime\cdots\prime}$ of the same net n_i. Also consider the bipartitioning of the *first* level-d hypergraph $\mathcal{H}^d_x$ that contains the subnet n'_i of that net n_i. It is clear that there are three cases of net n'_i in the bipartition $\Pi_2(\mathcal{H}^d_x) = \{\mathcal{V}_0, \mathcal{V}_1\}$: n'_i is cut, n'_i is internal to left part $\mathcal{V}_0$ or right part $\mathcal{V}_1$.

1) n'_i is cut in $\Pi(\mathcal{H}^d_x)$: This means that shared-factor matrix row r_i is communicated along dimension d of the hypercube thus already encapsulating the concurrent communication volume metric along dimension d. Then we can safely remove its sibling nets $n''_i, \cdots, n_i^{\prime\cdots\prime}$ from the respective subhypergraph partitionings $\Pi(\mathcal{H}^d_{y>x})$ to be performed later at this level. Although these sibling nets are not considered in the respective subhypergraph partitionings, the bipartitioning results of these subhypergraphs will be utilized to apply conventional cut net splitting on these sibling nets for including them into the subhypergraphs to be bipartitioned at the further RB levels $\ell > d$.
2) n'_i is internal to left part $\mathcal{V}_0$ in $\Pi(\mathcal{H}^d_x)$: This means that shared factor-matrix row r_i will incur concurrent communication volume only if at least one of its sibling nets $n''_i, \cdots, n_i^{\prime\cdots\prime}$ connect the right part $\mathcal{V}_1$ in a bipartition $\Pi(\mathcal{H}^d_{y>x})$ to be obtained at the current level. This corresponds to the case where that sibling net is either cut or internal to right part $\mathcal{V}_1$ in that bipartition $\Pi(\mathcal{H}^d_{y>x})$. Unfortunately current HP methods only adopt the cut net metric in two-way partitionings thus they cannot encode the increase in the cutsize for nets that are either cut or internal to a part. For this purpose, we introduce the net-anchoring scheme which is realized as follows: we introduce two vertices v^F_0 and v^F_1 which are fixed to left and right parts $\mathcal{V}_0$ and $\mathcal{V}_1$, respectively. Then a net is said to be anchored to the left part if it connects v^F_0, whereas it is said to be anchored to the right part if it connects v^F_1. We utilize net-anchoring to encode the concurrent communication volume for such nets as follows: In each subhypergraph $\mathcal{H}^d_{y>x}$ that contains a sibling net n''_i of n'_i, we anchor n''_i to the left part $\mathcal{V}_0$. In this way, we enforce n''_i to connect left part in all bipartitions of those hypergraphs to be obtained at the current level. Thus, if n''_i connects part $\mathcal{V}_1$ in any bipartition $\Pi_2(\mathcal{H}^d_{y>x})$ then it will become cut and increasing the cutsize so that it will encode concurrent communication volume to be incurred correctly. After the first bipartition $\Pi_2(\mathcal{H}^d_{y>x})$ in which n''_i is cut at level d, all other further sibling nets $n'''_i, \cdots, n_i^{\prime\prime\prime\cdots\prime}$ will be removed from the respective subhypergraph $\mathcal{H}^d_{z>y}$ partitionings at level d in accordance with the Case 1.
3) n'_i is internal to right part $\mathcal{V}_1$ in $\Pi(\mathcal{H}^d_x)$: This case is handled in a dual manner with Case 2. That is, after the first bipartition $\Pi_2(\mathcal{H}^d_{y>x})$ in which n'_i is internal to $\mathcal{V}_1$ at level d, in each subhypergraph $\mathcal{H}^d_{y>x}$ that contains a sibling net n''_i of n'_i, we anchor n''_i to the right part $\mathcal{V}_1$.

Fig. 2 illustrates the conventional cut net splitting technique (Fig. 2a) as well as the proposed enhancements (Fig. 2b and Fig. 2c) for net n_i on 8-way partitioning with 3 RB levels. In all subfigures, at the root level bipartitioning, n_i is cut and thus splitted into its subnets n'_i and n''_i.

In Fig. 2a, at level-1, n'_i remains internal to $\mathcal{V}_1$ in $\Pi(\mathcal{H}^1_0)$, whereas it is cut in $\Pi(\mathcal{H}^1_1)$. At level-2, n'_i is cut in $\Pi(\mathcal{H}^2_1)$, whereas subnets n'''_i and n''''_i of n''_i remain internal to the left and right part in $\Pi(\mathcal{H}^2_2)$ and $\Pi(\mathcal{H}^2_2)$, respectively. Since n_i is cut three times, its $con(n)-1$ value is three with the connectivity set $Con(n_i) = \hat{S}_i = \{p_1, p_2, p_6, p_7\}$. Expanding this row is shown in Fig. 1 for $owner(r_i) = p_1$.

Fig. 2b shows Cases 2 and 3. At level-1 of the figure, since n'_i is internal to $\mathcal{V}_1$ in $\mathcal{H}^1_0$, n''_i is anchored to the right part $\mathcal{V}_1$ in $\mathcal{H}^1_1$. Similarly, at level-2, n'_i is internal to $\mathcal{V}_0$ thus n'''_i and n''''_i are anchored to the left part $\mathcal{V}_0$ in $\mathcal{H}^2_2$ and $\mathcal{H}^2_3$, respectively. Fig. 2c shows Case 1. At level-2 of the figure, n'_i is cut thus its sibling nets n'''_i and n''''_i, which are splitted from n''_i, are removed from $\mathcal{H}^2_2$ and $\mathcal{H}^2_3$.

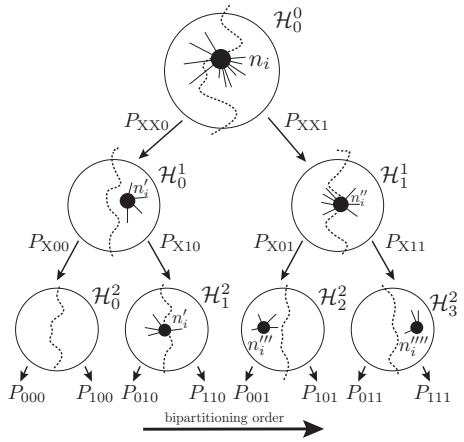
Algorithm 3 shows the steps of the proposed RB framework which realize the proposed enhancements. In the algorithm, $state(n)$ maintains if a net n becomes cut or internal to the left part ($L\text{-}internal$) or right part ($R\text{-}internal$) at the current level of the RB tree. $parent(n)$ denotes the parent net from which net n is obtained through splitting(s). That is, net n is effectively a subnet of $parent(n)$.

The outermost for loop in lines 4–32, performs the RB steps in breadth-first traversal order, whereas the inner for loop in lines 7–32 performs the bipartitionings at each level. The $state$ information of the nets are initialized to NIL at the beginning of each level (lines 5–6). Lines 8 and 9 introduce the fixed vertices into $\mathcal{H}^d_k$ for enabling the realization of the net-anchoring. The inner for loop in line 10–16 applies proposed net-removal and net-anchoring techniques before bipartitioning the current hypergraph $\mathcal{H}^d_k$ according to current states of the subnets involved in $\mathcal{H}^d_k$. The inner for loop in lines 18-25 computes the $state$ information for each net after bipartitioning. Lines 26–30 construct the left and right subhypergraphs $\mathcal{H}^{d+1}_{2k}$ and $\mathcal{H}^{d+1}_{2k+1}$ (to be bipartitioned at the next level $d+1$) from the current $\mathcal{H}^d_k$ using current bipartition $\Pi_2(\mathcal{H}^d_k)$ obtained in line 17 by utilizing the conventional cut net splitting. The for loop in lines 31–32 inherits the parent field of the cut nets to its split nets.

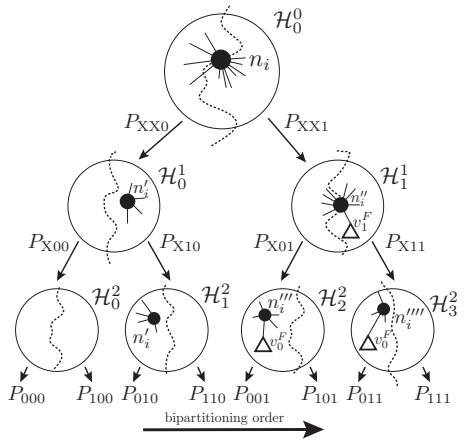
5.3 Factor-Matrix Row Assignment to Processors

The row-to-processor assignment problem corresponds to determining $owner(r_i)$ for each factor-matrix row r_i. For CPD utilizing P2P, the well known best-fit increasing heuristic used for solving the K-feasible bin-packing problem [26] is adopted [14], [19]. This method aims at balancing processors' volume loads without increasing the total communication volume. Here, we also adopt B-feasible bin-packing problem [26] for solving this assignment problem in EMB.

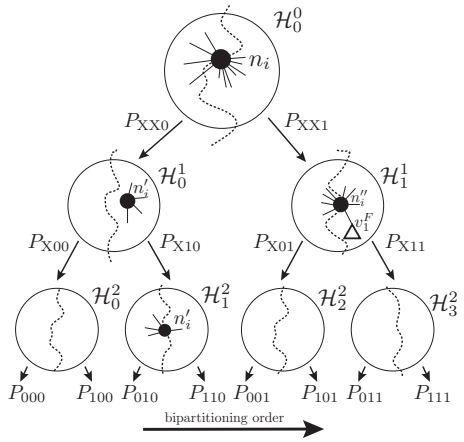
The main difference between the row-to-processor assignment problem encountered in P2P and EMB is that P2P involves a single communication step, whereas EMB involves loosely coupled $D = \log_2 K$ communication steps. So, in P2P, assignment of a row to a processor increases the volume load of only that processor, whereas in EMB it increases the volume loads of at most D processors in different communication steps. That is, if the distance between the owner and receiver processors is equal to the dimension D of the hypercube, there are $D-2$ intermediate processors which are only forwarding the factor-matrix row. So, each processor has a volume load at D different communication steps. This difference increases the number of bins from K in P2P to DK in EMB.



(a) Conventional cut net splitting



(b) Cases 2 and 3: proposed net anchoring



(c) Case 1: proposed net removal

Fig. 2. (a) Conventional cut net splitting, (b) and (c) proposed enhancements for net n_i on eight-way partitioning with three levels of RB steps.

In EMB, the cost of a row-to-processor assignment instance is defined as the sum of the volume load of the maximally loaded processor in each dimension. So, for the best-fit criterion we define the sum of squares function as

$$\sum_{d=1}^{d=D} (\sum_{k=1}^{k=K} B_{dk}^2)^2. \tag{1}$$

In the proposed algorithm, for each mode, factor-matrix

Algorithm 3 RB-based task-to-processor assignment

Require: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, K
1: $\mathcal{H}_0^0 = \mathcal{H}$
2: **for** each net $n \in \mathcal{N}_0^0$ **do**
3: $parent(n) = n$ ▷ initialize parent of the net as itself
4: **for** $d = 0$ **to** $\log_2 K - 1$ **do**
5: **for** each net $n \in \mathcal{N}_0^0$ **do**
6: $state(n) = NIL$ ▷ initial value for each net
7: **for** $k = 0$ **to** $2^d - 1$ **do**
8: $\mathcal{V}_k^d = \mathcal{V}_k^d \cup \{v_0^F, v_1^F\}$
9: fix v_0^F to $\mathcal{V}_0$, fix v_1^F to $\mathcal{V}_1$
10: **for** each net $n \in \mathcal{N}_k^d$ **do**
11: **if** $state(parent(n))$ is CUT **then**
12: $\mathcal{N}_k^d = \mathcal{N}_k^d \setminus \{n\}$
▷ remove n since it is already cut before at this level
13: **if** $state(parent(n))$ is $L\text{-}internal$ **then**
14: $Pins(n) = Pins(n) \cup \{v_0^F\}$ ▷ anchor n to left part
15: **if** $state(parent(n))$ is $R\text{-}internal$ **then**
16: $Pins(n) = Pins(n) \cup \{v_1^F\}$ ▷ anchor n to right part
17: $\Pi_2 =$ BIPARTITION$(\mathcal{H}_k^d = (\mathcal{V}_k^d, \mathcal{N}_k^d))$ ▷ $\Pi_2 = \{\mathcal{V}_0, \mathcal{V}_1\}$
18: **for** each net $n \in \mathcal{N}_k^d$ **do**
19: **if** n is a cut net **then**
20: $state(parent(n)) = CUT$
21: **if** $state(parent(n))$ is not CUT **then**
22: **if** n is internal to left part **then**
23: $state(parent(n)) = L\text{-}internal$
24: **if** n is internal to right part **then**
25: $state(parent(n)) = R\text{-}internal$
26: Form $\mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ induced by $\mathcal{V}_0$
27: $\mathcal{N}_0 = \{n' : n \in \mathcal{N}, pins(n) \cap \mathcal{V}_0 \neq 0 \; \exists \; pins(n') = pins(n) \cap \mathcal{V}_0\}$ ▷ conventional cut net splitting
28: Form $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$ induced by $\mathcal{V}_1$
29: $\mathcal{N}_1 = \{n'' : n \in \mathcal{N}, pins(n) \cap \mathcal{V}_1 \neq 0 \; \exists \; pins(n'') = pins(n) \cap \mathcal{V}_1\}$ ▷ conventional cut net splitting
30: $\mathcal{H}_{2k}^{d+1} = \mathcal{H}_0$, $\mathcal{H}_{2k+1}^{d+1} = \mathcal{H}_1$
31: **for** each cut net $n \in \mathcal{N}_k^d$ split as n' and n'' **do**
32: $parent(n') = parent(n'') = parent(n)$

rows are considered in decreasing order of their $|\hat{S}_i|$ values for assignment. The best-fit criterion for the assignment is to select the processor that incurs the minimum increase in (1). After each assignment, we increase the loads of the bins which are involved in the communication in terms of both send and receive volumes. In this way, (1) captures processors' send plus receive volume loads during expand communication which is equal to the sum of processors' send volume loads during expand and reduce communications thanks to the duality described in Sec. 4.3.

For P2P, each row is assigned to one of the processors which contributes/needs that factor matrix row. This ensures total communication volume does not increase with the assignment. On the other hand, for EMB, we can relax this constraint. That is, consider the processors that participate in the communication of a shared row but do not possibly contribute/need that row. Such processors can also be considered as candidate owners. Since these processors are already communicating that row such assignments might not increase the volume load. Obviously this relaxation is expected to further decrease the function in (1) because of

larger degree of freedom for each assignment. We should mention here that this relaxation in row-to-processor assignments does not affect the concurrent communication cost metric defined for individual shared factor-matrix rows and minimized by the scheme in Sec. 5.2.

6 Experiments

6.1 Setting

We performed experiments using three methods: P2P-mg, EMB-rand and EMB-hp. The term left to the hyphen denotes the parallel scheme used (P2P or EMB), whereas the right term denotes the nonzero partitioning method used. The mg in P2P-mg refers to partitioning the input tensor according to the state-of-the-art medium-grain HP model [19]. The rand in EMB-rand refers to partitioning the input tensor randomly in such a way that numbers of nonzeros assigned to processors differ by at most one. The hp in EMB-hp refers to partitioning and mapping the tensor nonzeros by using the method proposed in Sec. 5. For partitioning the hypergraph models in P2P-mg and EMB-hp, we use the tool PaToH [25], [27] with default parameters.

Parallel setup: The experiments are taken with up to 4096 processors on an Apollo 9000 HPC system. Each node in this system consists of two AMD EPYC 7742 processors, each with 64 cores, and 256 GB of memory. The nodes are connected with a Mellanox HDR Infiniband network. We use 16 cores per node in all our experiments.

Dataset: Our dataset consists of ten real-world sparse tensors with varying sizes. Table 1 shows the tensors and their properties. `Delicious`, `Enron`, `Flickr` and `NELL-1` are obtained from the FROSTT sparse tensors repository [28]. `1998DARPA` contains tuples that represent timestamps of connections made between source IP and destination IP. `Freebase-music` contains music-related (subject entity, object entity, relation) tuples from Freebase online database.`Gowalla` contains check-in data as (user, POI, check-in) tuples from the location-based social network Gowalla [29]. `Movies-amazon` contains user-movie-word tuples from the user reviews of movies in Amazon [30]. `Netflix` and `Yelp` are rating datasets that respectively contain (usr, business, rating) and (user, movie, rating) tuples.

The dataset also contains the largest three tensors from FROSTT, `Amazon-reviews`, `Patents` and `Reddit-2015`, each having more than 1B nonzeros. Since common HP tools such as PaToH and hMeTis [31] do not support 64-bit integers, these very large tensors are only used to evaluate the EMB framework (Sections 3 & 4) by comparing EMB-rand versus P2P-rand, whereas the rest of the tensors are used to evaluate all contributions.

TABLE 1
Properties of the Test Tensors

tensor	mode sizes				nnz	density
	I_1	I_2	I_3	I_4		
`1998DARPA`	23.8M	22.5K	22.5K		28.4M	2.37E-09
`Delicious`	533K	17.3M	2.47M	1.44K	140M	4.27E-15
`Enron`	6.07K	5.70K	244K	1.18K	54.2M	5.46E-09
`Flickr`	320K	28.2M	1.61M	731	113M	1.07E-14
`Freebase-music`	23.3M	0.17K	23.3M		100M	1.10E-09
`Gowalla`	1.3M	0.60K	107K		6.26M	7.65E-08
`Movies-amazon`	227K	4.40K	87.8K		15.0M	1.72E-07
`NELL-1`	25.5M	2.14M	2.90M		144M	9.05E-13
`Netflix`	480K	2.18K	17.8K		100M	5.40E-06
`Yelp`	773K	85.5K	687K		186M	4.09E-09
Very Large Tensors						
`Amazon-reviews`	4.82M	1.77M	1.80M		1.74B	1.13E-10
`Patents`	239K	46	239K		3.60B	1.37E-03
`Reddit-2015`	8.21M	176K	8.12M		4.69B	3.97E-10

6.2 Performance Results

Latency hiding: Table 2 displays the amount of latency hidden by EMB in terms of number of messages whose latency overheads are totally avoided. In the table, P2P columns show the number of messages only for the sparse expand and reduce operations during a CPD-ALS iteration. That is, latency overhead of `ALL-REDUCE` is not included in the P2P columns. The table also displays the latency overhead of `ALL-REDUCE` during a CPD-ALS iteration which is the only latency overhead in EMB. In the table, "max" and "avg" respectively refer to the maximum and average number of messages handled by processors during a CPD-ALS iteration. The maximum and average number of messages under the P2P columns are the sums of maximum and average number of messages required to perform the sparse expand and reduce operations in P2P for each tensor mode. The number of messages under the EMB column are the sum of messages during the two `ALL-REDUCE` operations for each tensor mode. Note that maximum and average values in EMB are equal due to the regularity of communication.

TABLE 2
Max/Avg number of messages in a CPD-ALS iteration on $K = 4096$

tensor	P2P		EMB
	max	avg	max(=avg)
`1998DARPA`	6,578	145	72
`Delicious`	25,413	13,332	96
`Enron`	19,755	2,149	96
`Flickr`	15,382	4,709	96
`Freebase-music`	14,739	868	72
`Gowalla`	6,245	907	72
`Movies-amazon`	9,590	1,786	72
`NELL-1`	22,543	15,434	72
`Netflix`	14,824	3,391	72
`Yelp`	20,375	6,112	72
average	**14,076**	**2,480**	**78**

Table 2 shows that sparse expand/reduce incur significantly large number of messages in P2P, thus rendering the parallel CPD-ALS as latency bound with increasing K. This is because the number of messages in P2P usually increases linearly with increasing K. On the other hand, the number of messages in EMB is significantly smaller and increase logarithmically with increasing K. The 72 and 96 values under EMB refer to the number of messages handled by a processor in 3-mode ($3{\times}2{\times}\log_2 K$) and 4-mode ($4{\times}2{\times}\log_2 K$) tensors, respectively.

TABLE 3
Improvement of Expand/Reduce-aware EMB against Naive EMB

method	K = 128	256	512	1024	2048	4096
EMB-rand	0.78	0.79	0.78	0.76	0.72	0.73
EMB-hp	0.90	0.86	0.84	0.81	0.82	0.76

Values are EMB runtimes normalized w.r.t. those by naive EMB.

As seen in Table 2, there is a significant imbalance between maximum and average number of messages in P2P. This disturbs the scaling performance since usually the maximum metric defines the runtime since there are global synchronizations (due to `ALL-REDUCE`) before/after the sparse P2P communication steps. On the other hand, this problem does not arise in EMB since the regular communication pattern of EMB naturally attains equal number of maximum and average messages. This is a clear advantage in favor of EMB since there is no need to consider reducing/balancing the number of messages when designing intelligent partitioning models allowing them to focus on reducing/balancing volume.

The expand-and-reduce-aware embedding: Table 3 shows the benefit of using expand-and-reduce-aware EMB (Sec. 4.2) over naive EMB (Sec. 4.1) on both EMB-rand and EMB-hp. The values given in the table are CPD-ALS iteration times of the ten tensors taken with expand-and-reduce-aware EMB normalized with respect to those taken with naive EMB. The runtimes of all tensors are then averaged per K value for each method. As seen in the table, utilizing expand-and-reduce-aware EMB for sparse expand and reduce decreases the parallel CPD-ALS runtime, on average, by up to $21\% - 28\%$ when EMB-rand is used and by $10\% - 24\%$ when EMB-hp is used. Furthermore, the relative percent improvement of expand-and-reduce-aware EMB over naive EMB for both EMB-rand and EMB-hp generally increases with increasing K.

HP-based mapping: Table 4 shows the performance improvement attained by the HP-based mapping algorithm discussed in Sec. 5 against EMB-rand on $K = 4096$. The performance comparison is given in terms of maximum and concurrent volume metrics as well as parallel runtimes for $R = \{8, 32\}$. For each tensor, the first line displays actual values for EMB-hp, whereas the second line displays normalized values with respect to those of EMB-rand.

As seen in Table 4, EMB-hp achieves significant decrease in concurrent communication volume metric (90% on average) compared to EMB-rand. EMB-hp achieves also significant decrease in maximum communication volume handled by a processor (65% on average) compared to EMB-rand. These improvements in concurrent and maximum communication volume metrics lead to an approximately 68% and 71% improvement in CPD-ALS iteration time respectively for $R = 8$ and 32. Note that improvement in the maximum communication volume closely correlates with the improvement in the parallel runtime on average.

Factor-matrix row assignment: Table 5 shows the performance improvement of the proposed bin-backing based factor-matrix row assignment method (Sec. 5.3) against random assignment on $K = \{128, \cdots, 4096\}$. The performance comparison is given in terms of the maximum communication volume handled by processors obtained by bin-packing-based-assignment algorithm normalized with respect to those by random assignment. As seen in the table, the bin-backing algorithm attains considerable performance improvement (15% on average) against random assignment.

TABLE 4
Performance of EMB-hp against EMB-rand on $K = 4096$

tensor	volume $\times R$		runtime (ms)	
	max	concurrent	$R = 8$	$R = 32$
`1998DARPA`	9,155	94,639	28.609	35.794
	0.359	0.004	0.929	0.702
`Delicious`	178,695	15,014,082	79.236	201.793
	0.477	0.124	0.512	0.311
`Enron`	23,964	443,498	4.984	16.829
	0.525	0.145	0.321	0.219
`Flickr`	51,391	5,472,870	12.802	77.683
	0.139	0.025	0.091	0.142
`Freebase-music`	19,421	5,440,976	64.492	424.432
	0.053	0.016	0.444	0.507
`Gowalla`	7,152	1,056,561	1.966	7.306
	0.314	0.126	0.254	0.219
`Movies-amazon`	22,730	1,152,274	4.473	15.737
	0.766	0.504	0.526	0.343
`NELL-1`	230,823	27,511,288	50.343	262.416
	0.651	0.176	0.369	0.479
`Netflix`	60,170	3,092,374	11.275	45.305
	0.440	0.519	0.151	0.137
`Yelp`	180,303	6,374,191	35.990	172.814
	0.591	0.525	0.261	0.265
average	**41,708**	**2,566,274**	**16.688**	**62.751**
	0.349	**0.098**	**0.322**	**0.292**

For each tensor, the first line displays actual values for EMB-hp, whereas the second line displays the normalized values w.r.t. those of EMB-rand.

TABLE 5
Performance of Proposed Row-to-Processor Assignment

tensor	maximum volume					
	K = 128	256	512	1024	2048	4096
`1998DARPA`	1.07	1.08	1.04	0.90	1.00	0.99
`Delicious`	0.82	0.86	0.84	0.85	0.86	0.86
`Enron`	0.91	0.93	0.90	0.86	0.89	0.86
`Flickr`	0.87	0.87	0.89	0.86	0.86	0.84
`Freebase-music`	0.69	0.63	0.70	0.63	0.60	0.56
`Gowalla`	0.81	0.79	0.83	0.82	0.84	0.81
`Movies-amazon`	0.85	0.88	0.84	0.81	0.87	0.83
`NELL-1`	0.80	0.84	0.83	0.87	0.88	0.88
`Netflix`	0.82	0.82	0.85	0.88	0.91	0.93
`Yelp`	0.85	0.84	0.84	0.84	0.89	0.84
average	**0.84**	**0.85**	**0.85**	**0.83**	**0.85**	**0.83**

Values are normalized w.r.t. those of random assignment.

Strong scaling: Fig. 3 shows the strong scaling curves of the three methods on $K = \{128, \cdots, 4096\}$ processors with two different R values. As seen in Fig. 3, P2P-mg does not scale after $K = 1024$ for the tensors `Delicious`, `Flickr` and `NELL-1`, whereas it does not scale after $K = 512$ for rest of the tensors. Both EMB schemes scale much better than P2P for each tensor and for both R values.

As seen in Fig. 3, EMB-hp runs much faster than EMB-rand in all instances thus showing the validity of the task-to-processor mapping method proposed in Sec. 5. Furthermore, EMB-hp runs much faster than the state-of-the-art P2P-mg for all tensors and all R values on $K > 1024$.

Fig. 4 shows the strong scaling curves for the three very large tensors. As seen in the figure, for each large tensor, P2P-rand fails to scale after 1024 processors, whereas EMB-rand continues to scale up to 4096 processors.

7 Related Work

In the literature, there exists many shared- and distributed-memory parallel CPD-ALS algorithms [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [32], [33], [34], [35]. Here we briefly mention about distributed-memory parallel CPD-ALS algorithms.

Several works on scaling distributed-memory parallel CPD-ALS target at enhancing the MTTKRP operation and/or reducing the bandwidth overhead of the P2P sparse reduce and expand operations through intelligent combinatorial models or through multidimensional division methods. For instance, among combinatorial models for enhancing MTTKRP, [15], [18] and [20] are proposed. Among combinatorial models for reducing communication overhead, HP is utilized [12], [14], [19], [20]. However these HP models focus on reducing the bandwidth component of the communication. Multidimensional cartesian partitioning is utilized with a nice property of bringing upper bounds on both bandwidth and latency components costs [13], [14]. The HP model in [14] targets at reducing the bandwidth requirement of cartesian partitioning. [16] also considers partitioning factor matrices column-wise at the expense of tensor replication, whereas all other methods as well as our method involves row-wise partitioning of the factor matrices. There also exists toolkits for shared- and distributed-memory parallel systems [11], [13], [17], [35], [36].

Latency reduction and hiding is well-known in parallel iterative solvers, such as Conjugate Gradient and GMRES, through communication/computation overlapping [37], [38], pipelining [39], and embedding [40]. The embedding scheme proposed in [40] exploits the fact that each SpMV is followed by an inner product which involves the input and output vectors. They propose to embed sparse expand operations on the output vector entries to the following inner product realized with `ALL-REDUCE` by utilizing row-parallel SpMV. Our work differs from [40] in the following aspects: The rearrangements which enable the embedding are different because of the nature of the applications (CG vs. CPD-ALS); [40] embeds only sparse expand whereas we embed both sparse expand and reduce; [40] use naive embedding so that each message in the `ALL-REDUCE` may contain multiple copies of same output-vector entries, whereas we avoid this with the proposed expand-and-reduce-aware embedding; [40] uses conventional HP followed by a KL-based one-to-one mapping, whereas we propose a simultaneous partitioning/mapping algorithm. To our knowledge, our work is the first to use latency hiding in parallel tensor decomposition.

8 Conclusion

We proposed a framework for hiding the latency of P2P sparse expand and reduce operations during parallel CPD-ALS through embedding them into dense collective `ALL-REDUCE` operations which already exist in the CPD-ALS. The framework consists of a computation/communication rearrangement of the CPD-ALS which enables the embedding as well as an intelligent embedding scheme that helps reducing the increase in communication due to embedding. The recursive-bipartitioning-based hypergraph partitioning method proposed for subtensor-to-processor mapping as well as the bin-backing-based method proposed for factor-matrix row to processor mapping are found to be quite effective in reducing the bandwidth overhead in the embedded-`ALL-REDUCE`. We have obtained very good scaling results on up to 4096 processors for ten real-word tensors, whereas a state-of-the-art P2P implementation does not scale after 1024 processors due to large the latency overhead especially for small decomposition ranks. The proposed latency-hiding framework paves the way for scalable sparse tensor decomposition on exa-scale systems.

Acknowledgments

This work is supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-116E043.

References

[1] T. D. Nguyen, T. Tran, D. Phung, and S. Venkatesh, "Tensor-variate restricted boltzmann machines," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[2] E. Acar and B. Yener, "Unsupervised multiway data analysis: A literature survey," *IEEE transactions on knowledge and data engineering*, vol. 21, no. 1, pp. 6–20, 2008.

[3] S. Hosseinimotlagh and E. E. Papalexakis, "Unsupervised content-based identification of fake news articles with tensor decomposition ensembles," in *Proceedings of the Workshop on Misinformation and Misbehavior Mining on the Web (MIS2)*, 2018.

[4] S. Rabanser, O. Shchur, and S. Günnemann, "Introduction to tensor decompositions and their applications in machine learning," *arXiv preprint arXiv:1711.10781*, 2017.

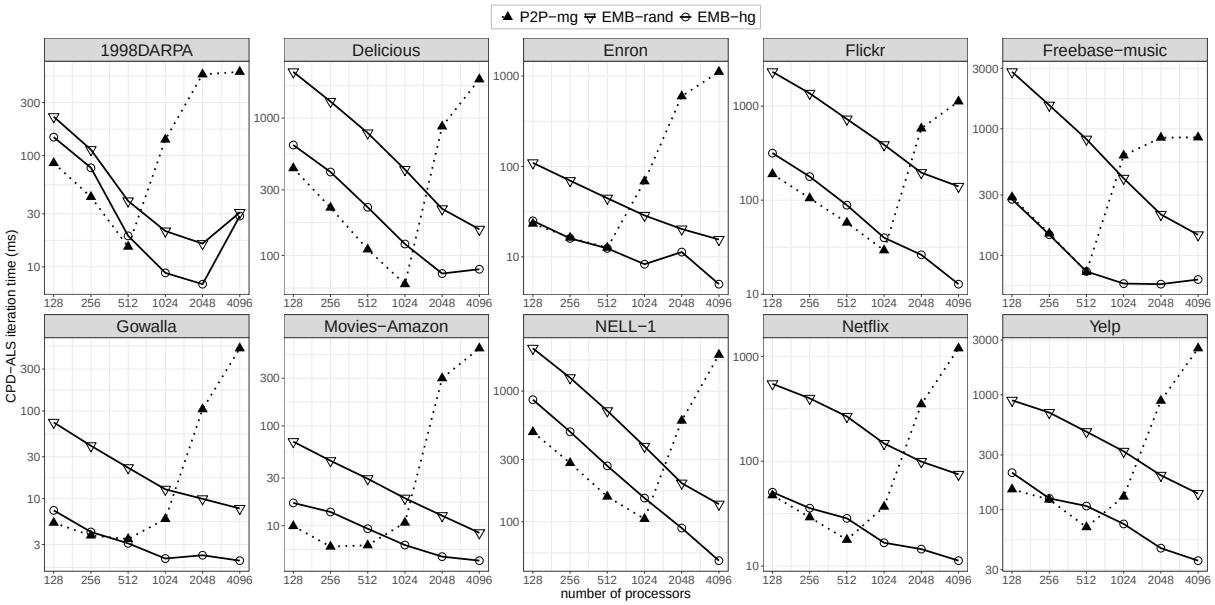
[5] H. Zhou, L. Li, and H. Zhu, "Tensor regression with applications in neuroimaging data analysis," *Journal of the American Statistical Association*, vol. 108, no. 502, pp. 540–552, 2013, pMID: 24791032.

[6] K. Makantasis, A. D. Doulamis, N. D. Doulamis, and A. Nikitakis, "Tensor-based classification models for hyperspectral data analysis," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 12, pp. 6884–6898, 2018.

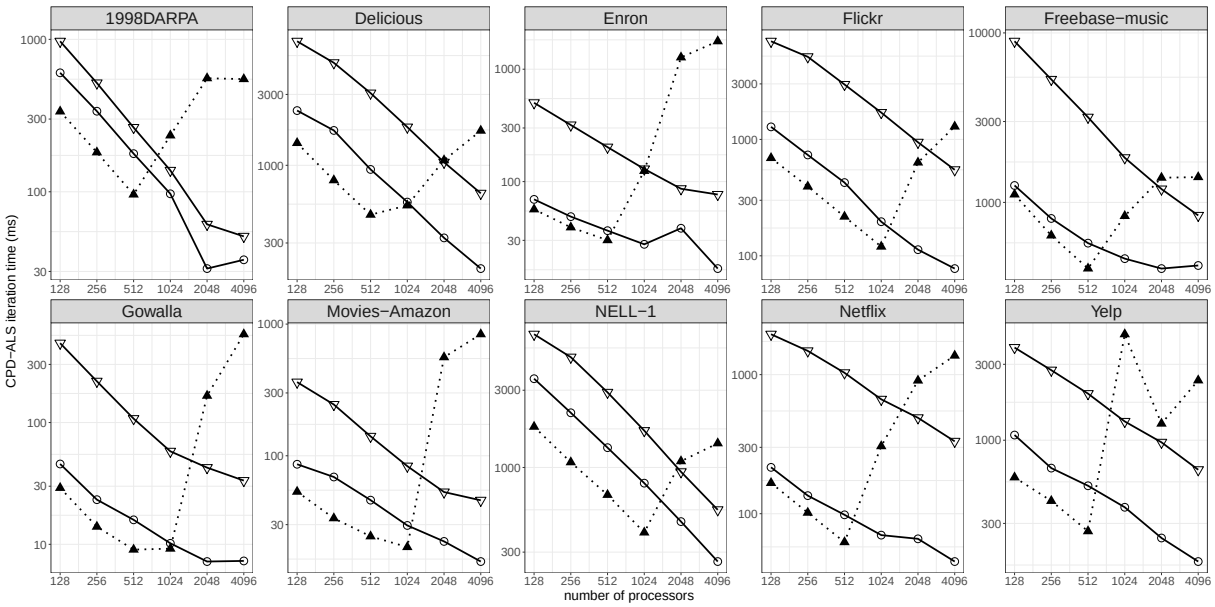
[7] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.

[8] Y. Wang, W. G. Guo, and X. Yue, "Tensor decomposition to compress convolutional layers in deep learning," *IISE Transactions*, p. 1–60, Apr 2021.

[9] D. Song, P. Zhang, and F. Li, "Speeding up deep convolutional neural networks based on tucker-cp decomposition," in *Proceedings of the 2020 5th International Conference on Machine Learning Technologies*, ser. ICMLT 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 56–61.



(a) $R = 8$



(b) $R = 32$

Fig. 3. Comparing Strong Scaling curves of P2P-mg, EMB-rand and EMB-hp with decomposition ranks $R = 8$ and $R = 32$.

[10] Y. Ji, Q. Wang, X. Li, and J. Liu, "A survey on tensor techniques and applications in machine learning," *IEEE Access*, vol. 7, pp. 162 950–162 990, 2019.

[11] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1296–1304.

[12] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–11.

[13] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 902–911.

[14] S. Acer, T. Torun, and C. Aykanat, "Improving medium-grain partitioning for scalable sparse tensor decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2814–2825, Dec 2018.

[15] O. Kaya and B. Uçar, "Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C99–C130, 2018.

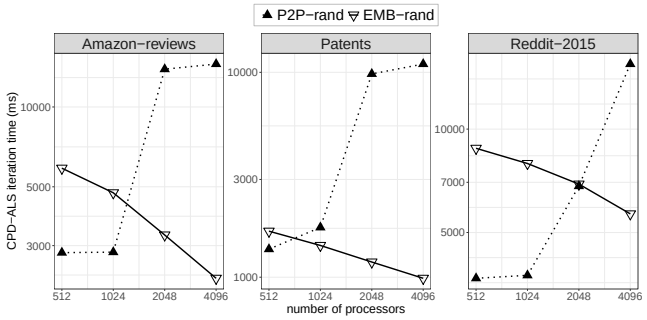
[16] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking optimization techniques for sparse tensor computation," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 568–577.

[17] M. Baskaran, T. Henretty, and J. Ezick, "Fast and scalable distributed tensor decompositions," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

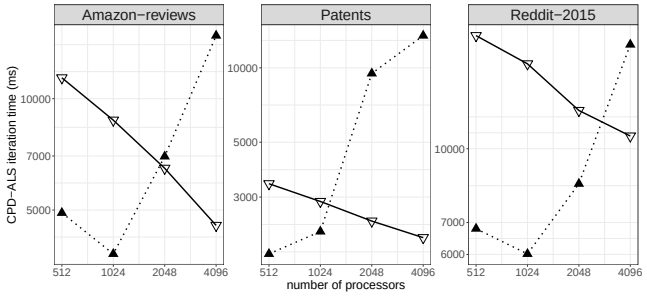
[18] L. Ma and E. Solomonik, "Efficient parallel cp decomposition with pairwise perturbation and multi-sweep dimension tree," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 412–421.

[19] M. O. Karsavuran, S. Acer, and C. Aykanat, "Partitioning models for general medium-grain parallel sparse tensor decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 147–159, 2021.

[20] N. Abubaker, S. Acer, and C. Aykanat, "True load balancing for matricized tensor times khatri-rao product," *IEEE Transactions on*



(a) $R = 8$



(b) $R = 32$

Fig. 4. Strong Scaling curves of EMB-rand versus P2P-rand on very large tensors with decomposition ranks $R = 8$ and $R = 32$.

Parallel and Distributed Systems, vol. 32, no. 8, pp. 1974–1986, 2021.

[21] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[22] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[23] Z. Chi, H. Yan, and T. Pham, *Fuzzy algorithms: with applications to image processing and pattern recognition*. World Scientific, 1996, vol. 10.

[24] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, "Iterative algorithms for solution of large sparse systems of linear equations on hypercubes," *IEEE Transactions on computers*, vol. 37, no. 12, pp. 1554–1568, 1988.

[25] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, July 1999.

[26] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD, USA: Computer Science Press, 1978.

[27] Ü. V. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*. Boston, MA: Springer US, 2011, pp. 1479–1487.

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

[29] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: user movement in location-based social networks," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1082–1090.

[30] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, 2013, pp. 165–172.

[31] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, March 1999.

[32] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 227–237.

[33] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:7.

[34] J. Li, J. Sun, and R. Vuduc, "HiCOO: Hierarchical storage of sparse tensors," in *SC18: Int. Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2018, pp. 238–252.

[35] K. D. Devine and G. Ballard, "GentenMPI: Distributed memory sparse tensor decomposition." August 2020.

[36] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 61–70.

[37] E. De Sturler and H. A. van der Vorst, "Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers," *Applied Numerical Mathematics*, vol. 18, no. 4, pp. 441–459, 1995.

[38] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a conjugate gradient solver with non-blocking collective operations," *Parallel Computing*, vol. 33, no. 9, pp. 624–633, 2007.

[39] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014.

[40] R. O. Selvitopi, M. M. Ozdal, and C. Aykanat, "A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 632–645, 2015.

Nabil Abubaker received the BS degree from An-Najah National University, Palestine, where he was an active IEEE student member and served as the vice-chair of the university's student branch. He received the MS degree from Bilkent University, Turkey where he is currently pursuing his PhD degree, all in Computer Engineering. His research interests include parallel and scientific computing, with focus on communication-efficient iterative algorithms.

M. Ozan Karsavuran received the BS, MS, and PhD degrees in 2012, 2014, and 2020, respectively, in computer engineering from Bilkent University, Turkey, where he is currently postdoctoral researcher. His research interests include combinatorial scientific computing, graph and hypergraph partitioning for sparse matrix and tensor computations, and parallel computing in distributed and shared memory systems.

Cevdet Aykanat received the BS and MS degrees from Middle East Technical University, Turkey, both in electrical engineering, and the PhD degree from Ohio State University, Columbus, in electrical and computer engineering. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Turkey, where he is currently a professor. His research interests mainly include parallel computing and its combinatorial aspects. He is the recipient of the 1995 Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as an Associate Editor of IEEE Transactions of Parallel and Distributed Systems between 2009 and 2013.