Efficient and Effective Sparse Tensor Reordering

Jiajia Li
jiajia.li@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Bora Uçar
borauca@ens-lyon.fr
UMR5668, CNRS and ENS Lyon
France

Umit V. Çatalyürek
umit@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Jimeng Sun
jsun@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Kevin Barker
kevin.barker@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Richard Vuduc
richie@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

ABSTRACT

This paper formalizes the problem of reordering a sparse tensor to improve the spatial and temporal locality of operations with it, and proposes two reordering algorithms for this problem, which we call BFS-MCS and Lexi-Order. The BFS-MCS method is a Breadth First Search (BFS)-like heuristic approach based on the maximum cardinality search family; Lexi-Order is an extension of doubly lexical ordering of matrices to tensors. We show the effects of these schemes within the context of a widely used tensor computation, the CANDECOMP/PARAFAC decomposition (CPD), when storing the tensor in three previously proposed sparse tensor formats: coordinate (COO), compressed sparse fiber (CSF), and hierarchical coordinate (HiCOO). A new partition-based superblock scheduling is also proposed for HiCOO format to improve load balance. On modern multicore CPUs, we show Lexi-Order obtains up to 4.14x speedup on sequential HiCOO-Mttkrp and 11.88x speedup on its parallel counterpart. The performance of COO- and CSF-based Mttkrp also improves. Our two reordering methods are more effective than state-of-the-art approaches. The code is released as part of Parallel Tensor Infrastructure (ParTI): https://github.com/hpcgarage/ParTI.

KEYWORDS

Sparse tensor, tensor decomposition, reordering, hierarchical coordinate, HiCOO

ACM Reference Format:

1 INTRODUCTION

We consider the problem of how to improve memory reference locality for sparse tensor computations [4, 19]. A tensor is a multiway (N-way) array, with a vector and a matrix being the 1-way and 2-way (rows and columns) specializations thereof. It is sparse if most of its entries are zero and, therefore, need not be stored nor explicitly computed upon. A sparse tensor is often a natural way to represent a multifactor or multirelational dataset, and has found numerous applications in data analysis and mining [4, 7, 9, 9, 11, 13, 14, 18, 19, 21, 25, 27, 33, 34, 37, 40, 43, 44, 44] for health care [14, 47], natural language processing [16], machine learning [1, 3, 30], and social network analytics [32], among many others.

Like sparse matrices, there are a variety of data structures and techniques for storing and operating efficiently with sparse tensors [4, 24, 25, 41]. Moreover, one technique is to reorder the tensor, which means relabeling the indices—and, therefore, reorganizing the nonzero structure of the tensor—in the hopes of improving spatial or temporal locality when operating with it. One form of reordering explored in prior work is to “sort” the input tensor [18, 25, 44]. In this paper, we formalize the reordering problem and propose and evaluate new heuristic algorithms for it.

While tensor reordering can improve locality, it can also aggravate load balance as measured by nonzeros assigned to each thread. We observe that reordering increases imbalance by as much as 6.7x in practice. In this paper, we propose improvements to the data structure and corresponding algorithms that address this problem.

Our evaluation is in the context of a widely used tensor computation known as the CANDECOMP/PARAFAC decomposition (CPD), a kind of generalization of the (truncated) singular value decomposition (SVD) from matrices to tensors [19]. Within a CPD, the most costly computational kernel is the matricized tensor-times-Khatri-Rao product, or Mttkrp [19, 24, 25, 44]. Thus, we examine the effects of reordering on the locality of sparse Mttkrp operations, which we then evaluate within a CPD on a modern multicore CPU platform. We also consider this evaluation when using three state-of-the-art tensor storage formats, known as coordinate (COO), compressed sparse fiber (CSF), and hierarchical COO (HiCOO).

Thus, this evaluation gives practitioners a sense of when reordering pays off in commonly occurring scenarios.

The main claimed contributions of this work are as follows.

- We propose two heuristic tensor reordering schemes, BFS-MCS and Lexi-Order, to enhance data locality by relabeling mode indices. (§ 3)
• We improve the state-of-the-art parallel strategy, superblock scheduling [24], by suggesting a partition-based version thereof. (§ 4)
• **LEXI-ORDER** achieves up to 4.14× speedup on sequential HiCOO-Mttkrp and 11.88× speedup on its parallel counterpart. BFS-MCS obtains up to 1.88x speedup on sequential HiCOO-Mttkrp and 1.94x speedup on its parallel case. (§ 5)
• For COO and CSF formats, LEXI-ORDER improves the performance of sequential COO- and CS-Mttkrps by up to 4.29× and 2.33× and parallel COO- and CS-Mttkrps by up to 1.48x and 1.86x respectively. BFS-MCS and LEXI-ORDER behave better than the state-of-the-art reordering based on graph and hypergraph partitionings [44] on CS-Mttkrp. (§ 5)
• We further parallelize the reordering and HiCOO conversion to reduce the pre-processing overhead. When CPD iterates for tens of times, which is common, then the reordering is worthwhile. (§ 5)

Table 1: List of symbols and notation.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{|n}$</td>
<td>Matricized tensor $X$ in mode-$n$</td>
</tr>
<tr>
<td>A, B, C, A</td>
<td>Dense matrices</td>
</tr>
<tr>
<td>$\alpha$, $\beta$, $\gamma$</td>
<td>Element indices of $A$</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>Weight vector</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>Khatri-Rao product between two matrices</td>
</tr>
<tr>
<td>$\odot$</td>
<td>Outer product between two vectors</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>Bit-length of a single-precision floating point value</td>
</tr>
<tr>
<td>$R$, $N$, $I$, $J$, $K$</td>
<td>Tensor order</td>
</tr>
<tr>
<td>$M$</td>
<td>Superblock scheduler of $X$</td>
</tr>
<tr>
<td>$R$</td>
<td>Approximate tensor rank (usually a small value)</td>
</tr>
<tr>
<td>$\beta_{\text{int}}, \beta_{\text{long}}, \beta_{\text{bit}}$</td>
<td>Bit-length of a byte or character</td>
</tr>
<tr>
<td>$\beta_{\text{int}}$</td>
<td>Bit-length of an integer</td>
</tr>
<tr>
<td>$\beta_{\text{bit}}$</td>
<td>Bit-length of a single-precision floating point value</td>
</tr>
<tr>
<td>$\text{inds}, \text{cptrs}, \text{plinds}$</td>
<td>Indices of COO, block pointers of COO, block indices of COO</td>
</tr>
<tr>
<td>$\text{bptr}$</td>
<td>Block pointers of HiCOO and CSF</td>
</tr>
<tr>
<td>$\text{lschr}$</td>
<td>Superblock scheduler of HiCOO</td>
</tr>
<tr>
<td>$\text{blas}$</td>
<td>Partition-based superblock scheduler of HiCOO</td>
</tr>
<tr>
<td>$\text{val}$</td>
<td>Nonzero value array of COO, CSF, HiCOO</td>
</tr>
</tbody>
</table>

2 BACKGROUND

We summarize the related symbols and notation of tensors, CPD, and the three state-of-the-art tensor formats (COO, CSF, HiCOO) in Table 1.

2.1 Tensors and CPD

The order of a tensor, $N$, is the number of its dimensions or modes. We follow the notation in Kolda and Bader’s survey [19]. A first-order tensor ($N = 1$) is a vector, denoted by a boldface lowercase letter, e.g., $\mathbf{v}$; a second-order tensor ($N = 2$) is a matrix, denoted by a boldface capital letter, e.g., $\mathbf{A}$. Higher-order tensors ($N \geq 3$) are denoted by bold capital calligraphic letters, e.g., $\mathcal{X}$. We show an example of a sparse third-order tensor, $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, in Figure 1. In this example, a scalar entry of $\mathcal{X}$ at position $(i, j, k)$ is $x_{ijk}$. We assume an example sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with $M$ nonzeros in the following context.

CPD decomposes a tensor into a sum of component rank-one tensors [19]. It approximates an $N$th-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ as

$$
\mathcal{X} = \sum_{r=1}^{R} \lambda_r \alpha_r^{(1)} \odot \cdots \odot \alpha_r^{(N)} \equiv [\lambda, \alpha^{(1)}, \ldots, \alpha^{(N)}],
$$

where $R$ is the canonical rank of tensor $\mathcal{X}$, taken as the number of component rank-one tensors [19]. In a low-rank approximation, $R$ is usually chosen to be a small number less than 100. The outer product of the vectors $\mathbf{a}_{(r)}^{(1)}$, $\mathbf{a}_{(r)}^{(2)}$, $\ldots$, $\mathbf{a}_{(r)}^{(N)}$ produces rank-one tensors, and $\mathbf{A}^{(n)} = \mathbf{A}^{(n)} \odot \mathbf{R}$, $n = 1, \ldots, N$ are the factor matrices, each one formed by taking the corresponding vectors as its columns, i.e., $\mathbf{A}^{(n)} = [\mathbf{a}_{(1)}^{(n)}, \mathbf{a}_{(2)}^{(n)}, \ldots, \mathbf{a}_{(R)}^{(n)}]$. We normalize these vectors to unit magnitude and store the factor weights in the vector $\mathbf{A} = [\lambda_1, \ldots, \lambda_R]$. The bottleneck of CPD is the matricized tensor-times-Khatri-Rao product (Mttkrp). Given an $N$th-order tensor $\mathcal{X}$ and matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, the mode-$n$ Mttkrp is

$$
\mathcal{X}^{(n)} \leftarrow X_{\|n} \left( \mathbf{A}^{(N)} \otimes \cdots \otimes \mathbf{A}^{(n+1)} \otimes \mathbf{A}^{(n)} \right),
$$

where $X_{\|n}$ is the mode-$n$ matricization (or unfolding) of tensor $\mathcal{X}$, $\otimes$ is the Khatri-Rao product. Mode-$n$ matricization reshapes a tensor into an equivalent matrix [19]. The Khatri-Rao product is a “matching column-wise” Kronecker product between two matrices. Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, their Khatri-Rao product is denoted by $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ where $\mathbf{C} \in \mathbb{R}^{I \times J \times R}$,

$$
\mathbf{C} = \mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \odot \mathbf{b}_1, \mathbf{a}_2 \odot \mathbf{b}_2, \ldots, \mathbf{a}_R \odot \mathbf{b}_R],
$$

where $\mathbf{a}_r$ and $\mathbf{b}_r$, $r = 1, \ldots, R$ are columns of $\mathbf{A}$ and $\mathbf{B}$, $\odot$ is the outer product of vectors, a special case of Kronecker product [19]. In particular for a third-order tensor, Mttkrp multiplies each nonzero entry $x_{i,j,k}$ with the $R$-vector formed by the entry-wise product of the $j$th row of $\mathbf{A}^{(2)}$ and $k$th row of $\mathbf{A}^{(3)}$ when computing $\mathbf{A}^{(1)}$. Detailed description can be found in Kolda and Bader’s survey [19]. In this paper, sparse Mttkrp will mean an Mttkrp between a sparse tensor and dense matrices.

2.2 Sparse Tensor Formats

We consider the three state-of-the-art tensor formats (COO, CSF, HiCOO) which are all for general unstructured sparse tensors.

2.2.1 COO. The coordinate (COO) format is the simplest yet arguably most popular format by far. It stores each nonzero value along with all of its position indices, shown in Figure 2 (a).
We propose two heuristics for reordering indices. The aim of the heuristics is to arrange the nonzeros close to each other, in all modes. If we were to look at matrices, this would correspond to reordering the rows and columns so that all nonzeros are clustered around the diagonal. This way, nonzeros in a row or column would be close to each other, and any blocking (by imposing fixed sized blocks) would have nonzero blocks only around the diagonal. The proposed heuristics are based on these observations and try to obtain similar behavior for tensors. The output of a reordering algorithm is the permutations for all modes being used to relabel tensor indices of them.

3.1 BFS-MCS

BFS-MCS is a Breadth First Search (BFS)-like heuristic approach based on the maximum cardinality search family [46]. We first construct a hypergraph for a sparse tensor, where vertices are tensor indices in all modes and hyperedges represent its nonzero entries. For a third-order sparse tensor \( X \in \mathbb{R}^{I_1 \times I_2 \times I_3} \) with \( M \) nonzeros, its hypergraph \( H = (V, E) \) consists of \( |V| = I_1 + I_2 + I_3 \) vertices and \( |E| = M \) hyperedges. A nonzero entry \( x_{i_1,i_2,i_3} \) connects the three vertices \( i_1, i_2, i_3 \). Figure 3 (b) shows an example of the hypergraph for a sparse tensor. Vertices are blank circles, and hyperedges are represented by grouping vertices.

For an \( N \)-th-order sparse tensor, we need to find the permutations for \( N \) modes. We determine a permutation for a given mode \( n \) (perm \( n \)) of a tensor \( X \in \mathbb{R}^{I_1 \times \ldots \times I_N} \) as follows. Suppose some of the indices of mode \( n \) are already ordered. Then, BFS-MCS picks the next to-be-ordered index as the one with the strongest connection to the currently ordered index set. In the case of ties, it selects the index with the smallest number of nonzeros in the corresponding sub-tensor. Intuitively, in the hypergraph of a sparse tensor, a stronger connection represents more common indices in the modes other than \( n \) among an unordered vertex and the already ordered ones. This means more data from factor matrices can be reused in \( M_{TTRP} \), if found in cache.

This process is implemented by maintaining a max-heap \( (H_n, \text{perm}_n) \) in Line 3 in Algorithm 1 for all mode-\( n \) indices (e.g., \( i_n \)) with two keys. The primary key of an index is the number of connections to the currently ordered indices. The secondary key is the number of nonzeros in the corresponding \( (N - 1) \)-th order sub-tensor, e.g., \( X(\ldots,i_n,\ldots) \), where the smaller secondary key values signify higher priority. Note that the secondary keys are static. Initially, \( H_n \) is constructed according to the secondary keys of mode-\( n \) indices. For each mode-\( n \) index (e.g., \( i_n \)) of this max-heap, BFS-MCS traverses all connected tensor indices in the modes except \( n \), calculates the number of connections of mode-\( n \) indices, and then updates the heap \( (H_n) \) using the primary and secondary keys.

Algorithm 1 makes BFS-MCS more precise. Let \( m_V \) denote a size-\(|V|\) array that tracks whether a vertex has been visited. They are
Algorithm 1 BFS-MCS ordering based on maximum cardinality search for a given mode.

Input: An Nth-order sparse tensor \( \mathbf{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N} \), hypergraph \( \mathcal{G} = (V, E) \) with vertex weights \( w \), mode \( n \);
Output: Permutation \( \text{perm}_n \);

1. Initialize \( \mathbf{w}_0 \) to zeros.
2. Initialize \( \mathbf{w}_x \) to the minus of number of nonzeros of each corresponding sub-tensor as a static secondary key.
3. Build max-heap \( H_n \) for mode-\( n \) indices with \( \mathbf{w}_p \) and \( \mathbf{w}_s \) as the primary and secondary keys respectively.
4. Initialize \( \mathbf{m}_v \) to zeros.
5. for \( i_n = 1, \ldots, I_n \) do
6. \( \nu_n^{(0)} = \text{GetHeapMax}(H_n); \)
7. \( \text{perm}_n(\nu_n^{(0)}) = i_n; \)
8. \( \nu_n = \nu_n^{(0)}; \)
9. for \( e_n \in \text{hyperedges of } \nu_n^{(0)} \) do
10. for \( v \in e_n, v \) is not in mode \( n \) and \( \mathbf{m}_v(v) == 00 \) do
11. \( \nu_n(v) = 1; \)
12. for \( e \) in hyperedges of \( \nu_n \) and \( e \neq e_n \) do
13. \( \nu_n = \text{vertex in mode } n \) of \( e; \)
14. if \( \nu_n \) in Heap(\( \nu_n \)) then
15. \( \nu_n = \text{vertex in mode } n \) of \( e; \)
16. heapUpdateKey(\( H_n, \mathbf{w}_p(\nu_n); \)
17. return \( \text{perm}_n \);

initialized to zeros (unvisited) and changed to 1s once after being visited. We record a mode-\( n \) index \( \nu_n^{(0)} \), obtained from the max-heap \( H_n \), to the permutation array \( \text{perm}_n \). The algorithm visits all its hyperedges (i.e., all nonzeros with \( i_n \), Line 9) and their connected and unvisited vertices from the other \( (N - 1) \) modes except \( n \) (Line 10). For these vertices, it again visits their hyperedges (e in Line 12) and then checks if the connected vertices in mode \( n \) (\( \nu_n \)) are in the heap (Line 14). If so, the primary key (connectivity) of \( \nu_n \) is increased by 1 and the max-heap \( H_n \) is updated. To summarize, this algorithm locates the sub-tensors of the neighbor vertices of \( \nu_n^{(0)} \) to increase the primary key values of the occurred mode-\( n \) indices in \( H_n \).

The BFS-MCS heuristic has low time complexity. We explain it using tensor terms. The innermost heap update operation (Line 16) costs \( O(\log(I_n)) \). Each sub-tensor is only visited once with the help of the marker array \( \nu_n \). For all the sub-tensors in one tensor mode, the heap update is only performed for \( M \) nonzeros (hyperedges). Overall, the time complexity of BFS-MCS is \( O(NM \log(I_n)) \) for an Nth-order tensor with \( M \) nonzeros, when computing \( \text{perm}_n \) for mode \( n \).

BFS-MCS does not exactly catch the memory access pattern of an \texttt{Mttkrp}. It treats the contribution from the indices of all modes except \( n \) equally to the connectivity, thus the connectivity might not match the actual data reuse. Also, it uses a greedy strategy to determine the next-level vertices, which could miss the optimal global orderings, as is common to greedy heuristics for hard problems.

3.2 Lexi-Order

Lexi-Order is an extension of doubly lexical ordering of matrices [26, 31] to tensors. A lexicographic ordering of an integer vector is the standard dictionary ordering of its elements, defined as follows. Given two equal-length vectors, \( x \) and \( y \), we say \( x \leq y \) if either (i) all elements are the same, i.e., \( x = y \); or (ii) there exists an index \( j \) such that \( x(j) < y(j) \) and \( x(i) = y(i) \) for all \( 0 \leq i < j \). For example, Figure 4 (a) shows the vector comparison of two zero-one vectors, \( x \equiv (1, 0, 1, 0) \) and \( y \equiv (1, 1, 0, 0) \). \( x \leq y \) because \( x(1) < y(1) \) and \( x(i) = y(i) \) for all \( 0 \leq i < 1 \).

A doubly lexical ordering of a matrix [26] is an ordering of the rows and columns of the matrix so that both the row and column vectors are in non-increasing lexicographic order. A row vector is read from left to right, and a column vector is read from top to bottom. Every real-valued matrix has a doubly lexicographic ordering, and such an ordering is not unique (the proof can be found in [26]). Figure 4 (b) shows a doubly lexical ordering of a zero-one example matrix. The row vectors are in non-increasing lexicographic order from top to bottom, and the column vectors are in non-increasing lexicographic order from left to right. Assume the ordered matrix in Figure 4 (b) is \( A \), then rows \( a(1, :) \geq a(2, :) \geq a(3, :) \geq a(4, :) \) and columns \( a(:, 1) \geq a(:, 2) \geq a(:, 3) \geq a(:, 4) \). Even from this simple example, the ordered matrix shows better data locality than the original one. Doubly lexical ordering has a number of applications, including the efficient recognition of totally balanced, subtree, and plaid matrices and (strongly) chordal graphs. To the best of our knowledge, the merits of this ordering for high performance have not been investigated for sparse matrices.

We propose an iterative algorithm called \texttt{matLexiOrder} for doubly lexical ordering of matrices, where in an iteration either rows or columns are sorted. The known doubly lexical ordering algorithms for matrices, which were first explored by Lubiw [26] and then improved by Paige and Tarjan [31], are “direct” ordering methods with a non-linear runtime of \( O(M \log(I + J) + J) \) and \( O(M + I + J) \) space, for an \( I \times J \) matrix with \( M \) nonzeros. We find the time complexity of these algorithms to be too high for our purpose. Furthermore, the data structures are too complex to allow an efficient generalized implementation for tensors. By appealing to a result of Lubiw [26, Claim 2.2], one can show that our \texttt{matLexiOrder} algorithm finds a doubly lexical ordering in a finite number of iterations. We do not show the proof, since we do not aim to obtain an exact doubly lexical ordering; a close-by ordering will likely suffice to improve the \texttt{Mttkrp} performance.

We first describe \texttt{matLexiOrder} algorithm for a sparse matrix. This aims to show the efficiency of our iterative approach compared to others. A partition refinement technique is used to order the rows and columns alternatively. Given an ordering of the rows, the columns can be sorted lexically. This is achieved by an order preserving variant of the partition refinement method [31], which is called \texttt{orderLyRefine}. We briefly explain the partition refinement technique for ordering the columns of a matrix. Given an \( I \times J \) matrix \( A \), all columns are initially put into a single part. Then, \( A \)‘s nonzeros are visited row by row. At a row \( i \), each column part \( C \) is split into two parts \( C_1 = C \cap a(i,:) \) and \( C_2 = C \setminus a(i,:) \), and...
these two parts replace C in the order $C_1 > C_2$ (empty sets are discarded). Note that this algorithm keeps the parts in a particular order which generates an ordering of the columns. orderLyRefine is used to refine all parts that have at least one nonzero in row $i$ in $O(|a(i,:)|)$ time, where $|a(i,:)|$ is the number of nonzeros of row $i$. Overall, matLexiOrder costs a linear total time of $O(M + 1 + J)$ (for rows and columns ordering) per iteration and $O(f)$ space. We also observe that only a small number of iterations will be enough (will be shown in §5.8 for tensors), yielding a more storage-efficient algorithm compared to the prior doubly lexicographical ordering methods [26, 31]. matLexiOrder, in particular the use of orderLyRefine routine a few times, is sufficient for our needs. This is so, as we do not need a full lexicographic order, thereby making our approach faster and simpler to implement.

Algorithm 2 Lexi-Order for a given mode.

Input: An $N$th-order sparse tensor $X \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, mode $n$;
Output: Permutation $perm_n$;

1. quickSort($X$, coordCmp);
2. $r = \text{compose}(\text{inds}([-n], 1))$;
3. for $m = 1, \ldots, M$ do
4. \hspace{1em} $c = \text{inds}(n, m);$
5. \hspace{1em} if coordCmp$(X, m, m - 1) = 1$ then
6. \hspace{2em} $r = \text{compose}(\text{inds}([-n], m));$
7. \hspace{1em} \hspace{1em} $X(n)(r, c) = \text{val}(m);$
8. end if
9. Use a variation of partition refinement in [31];
10. $perm_n = \text{orderLyRefine}(X(n));$
11. return $perm_n$;

Function: coordCmp($X$, $m_1$, $m_2$);
12. for $n' = 1, \ldots, N$ do
13. \hspace{1em} if $\text{perm}_n(n') = n$ then
14. \hspace{2em} if $m_1(n') < m_2(n')$ then
15. \hspace{3em} return $-1$;
16. \hspace{2em} else if $m_1(n') > m_2(n')$ then
17. \hspace{3em} return $1$;
18. end if
19. \hspace{1em} else
20. \hspace{2em} return $0$;
21. end if
22. end for

Figure 5: The steps of Lexi-Order illustrated for mode 1.

To order tensors, we propose the Lexi-Order function as an extension of matLexiOrder. The basic idea of Lexi-Order is to determine the permutation of each tensor mode independently, while considering the order in other modes fixed. Lexi-Order sets the indices of the mode to be ordered as the columns of a matrix, the other indices as the rows and sorts the columns as described for matrices (with the order preserving partition refinement method). The precise algorithm appears in Algorithm 2, which we also illustrate in Figure 5 when applied to mode 1. Given a mode $n$, Lexi-Order first builds a matricized tensor in Compressed Sparse Row (CSR) sparse matrix format by a call to quickSort with the comparison function coordCmp and then by partitioning the nonzeros into the row segments (Lines 3–7). This comparison function coordCmp does a lexicographic comparison of all-but-mode-$n$ indices, which enables efficient matricization. In other words, sorting of the tensor $X$ is the same as building the matricized tensor $X(m)$ by rows in the fixed lexical ordering, where mode $n$ is the column dimension and the remaining modes constitute the rows. In Figure 5, the sorting step orders the COO entries by $(j, k)$ tuples, which then serve as the row indices of the matricized CSR representation of $X(n)$. Once the matrix is built, we construct zero-one row vectors in Figure 5 to illustrate its nonzero distribution, which could seamlessly call orderLyRefine function. Apart from the quickSort, the other parts of Lexi-Order are of linear time complexity (linear in terms of tensor storage). We use OpenMP Tasks to parallelize quickSort to accelerate Lexi-Order.

Like BFS-MCS approach, Lexi-Order also finds the permutations for $N$ modes of an $N$th-order sparse tensor. Figure 6 (a) illustrates the effect of Lexi-Order on an example $4 \times 4 \times 3$ sparse tensor. The original tensor is converted to a HiCOO representation with block size $B = 2$ consisting of 5 blocks, with maximum 2 nonzeros per block. After reordering with Lexi-Order, the new HiCOO has 3 nonzero blocks with up to 4 nonzeros per block. Thus, the blocks are denser, which should exhibit better locality behavior. However, this reordering scheme is heuristic. Figure 6 (b) shows HiCOO representation after reordering of the tensor shown in Figure 2. The number of blocks is unchanged (4), although the maximum number of nonzeros per block increases to 4. For this tensor, Lexi-Order may not show a big advantage.

Figure 7: The conversion between COO and HiCOO formats for an example third-order tensor. HiCOO uses $2 \times 2 \times 2$ blocks ($B = 2$) with word sizes marked above.
3.3 Analysis

We take Mttkrp operation to analyze reordering behavior for COO, CSF, and HiCOO formats. Recall that for a third-order sparse tensor $\mathbf{X}$, Mttkrp multiples each nonzero entry $x_{i,j,k}$ with the $R$-vector formed by the entry-wise product of the $j$th row of $\mathbf{A}^{(2)}$ and the $k$th row of $\mathbf{A}^{(3)}$ when computing $\mathbf{A}^{(1)}$. The arithmetic intensity of Mttkrp algorithms on these formats is approximately $1/4$ [24]. Thus, Mttkrp can be considered memory-bound for most computer architectures, especially CPU platforms.

We analyze the memory access of HiCOO-Mttkrp algorithm in bytes (Equation (4), details in Equation (16) of [24]) to roughly reflect its real performance. $\beta_i$ is different bit-lengths for HiCOO (Table 1). The block ratio ($\alpha_b$) and the average slice size per tensor block ($\overline{\ell}$) are two critical parameters of HiCOO. Smaller $\alpha_b$ and larger $\overline{\ell}$ are favorable for good HiCOO-Mttkrp performance. (Readers could refer to [24] for more explanation.) Our reordering algorithms tend to closely pack nonzeros together and get denser blocks (larger $\overline{\ell}$), which potentially generates less tensor blocks (smaller $\alpha_b$), thus HiCOO-Mttkrp has less memory access, more cache hits, and better performance further. Take for example two nonzeros ($i_{j_2}, j_{k_2}$) and ($i_{j_3}, j_{k_3}$) again. A HiCOO representation after reordering could combine them in a single block. (A detailed example is shown in Figure 6.) Moreover, from the HiCOO storage equation (Equation (5), Equation (13) from [24]), smaller $\alpha_b$ can also reduce tensor memory requirement as a side benefit. That is, for HiCOO format a reordering scheme should increase the block density, reduce the number of blocks, and increase cache locality—three related performance metrics. Reordering is more beneficial for HiCOO format than COO and CSF formats.

$$\text{Bytes}_{\text{cache}} = \frac{M}{8} [2\alpha_b \cdot \beta_i + \beta_{byte} + R \min \left( \frac{1}{2\overline{\ell}}, 1 \right) \cdot \beta_{byte}] \cdot N$$ (4)

$$S_{\text{cache}} = M[\alpha_b \cdot \beta_{byte} + \alpha_b \cdot N \cdot \beta_i + N \cdot \beta_{byte}]$$ (5)

COO stores all nonzero indices, so relabeling does not make a difference in its storage. The same is also true for CSF; relabeling does not change the its tree structure, while the order of nodes may change. For an Mttkrp with tensors stored in these two formats, the performance gain from reordering is only the improved data locality and therefore cache behavior. Though it is hard to do theoretical analysis for them, the potential better data locality could also brings performance advantages, but might be less than HiCOO’s.

3.4 Parallelize Preprocessing

For an efficient parallel HiCOO-Mttkrp algorithm, superblocks, an extra blocking level above blocks, are proposed to increase the workload granularity of scheduling [24]. A superblock is essentially a “logical” subtensor that can potentially consist of many small blocks. During HiCOO format conversion in Figure 7, we first sort all nonzeros only by $i$ indices in mode 1 (“rowblock sorting”), to ensure that a mode-1 slice is not split between superblocks. Then $L \times \cdots \times L$ nonzero superblocks are partitioned with an additional array $\text{iptr}$ to store the beginning pointers of nonzero superblocks in size $n_t$, the number of superblocks. We treat them as independent subtensors to convert each to the physical HiCOO format with $B \times \cdots \times B$ blocks ($L \geq B$) through Z-Morton order sorting, partitioning, and compression steps. In the first rowblock sorting we do a quicksort for a single mode where the quicksort is parallelized. Afterwards, the operations on each superblock (Z-order sorting, partitioning, and compression) are naturally parallelized.

![Figure 8: Baseline [24] and partition-based superblock scheduling tables for mode-1 Mttkrp. Each box represents a superblock identified by its number, along with its number of nonzeros in parentheses.](image)

4 HYBRID PARALLEL HICOO-MTTKRP

An important side-effect of the proposed reordering schemes is that it can inadvertently create load imbalance, if one assumes the original “superblock scheduling” scheme proposed for HiCOO [24]. This section summarizes the issue and proposes a modification to HiCOO’s superblock scheduler to mitigate such imbalance.

The baseline HiCOO superblock scheduler operates schematically as shown in Figure 8 (a). Each box is a superblock, where the number of nonzeros inside the block is shown in parentheses. Superblocks in the same row all write to the same output area and so are dependent (arrows), while columns indicate independent superblocks. Two parallel strategies are employed in [24]: direct and privatized parallelization. Simply speaking, direct parallelization parallelizes rows while privatization strategy parallelizes iterations (or columns). Direct parallelization uses an owner-computer strategy to assign one (or more rows) to each available thread and then execute each column (”iteration”) of independent superblocks in a bulk-synchronous fashion. Privatized parallelization uses thread-local buffers to keep the updates of the superblocks from distinct iterations, and then accumulates these buffers together in parallel at the end. The superblock scheduler also varies the superblocks-to-threads assignment from row-to-row or iteration-to-iteration if necessary for the two strategies respectively to achieve better load balance. These two parallel strategies are proposed to obviate the need for locks or atomic operations and to explore sufficient parallel degree for irregular sparse tensors.

However, neither of these two strategies can obtain good load balance due to the uneven nonzero distribution of the superblocks in rows or columns, even under dynamic thread scheduling policy. This situation becomes worse after applying tensor reordering. We use the ratio of maximum number of nonzeros assigned to a thread to the average number of nonzeros per thread to represent the load balance. Under a random ordering, which might be expected to have good load balance, at the price of poor locality, most ratios are under 1.3. While after Lexi-ORDER reordering, using the baseline HiCOO superblock scheduler, the ratios become worse on most of tensors and raise to as high as 6.72, although it would be expected to have better locality. (Refer to details in Table 4.) Therefore, we are motivated to try to mitigate this imbalance.

Here is our scheme that tries heuristically to improve the balance ratio. The main idea behind our modified superblock scheduler is illustrated in Figure 8 (b). We simply aggregate superblocks in the
same row of the figure, which effectively creates “jagged” partition boundaries (portions 1 and 2 in Figure 8 (b)), with any remainder superblocks in their own final “tail” partition. To decide how many superblocks within each row may be aggregated at a time, there is an additional tuning parameter which is a target maximum number of nonzeros to aggregate at a time. In this example, that target is set to 8 nonzeros, superblocks 0 and 4 are aggregated, as are the pairs (2, 6) and (3, 7). Correspondingly, our scheme again uses two parallel strategies: direct and privatized parallelization. However, direct parallelization parcellizes rows in units of partitions, and privatization strategy parcellizes partitions by saving thread-local buffers. Under this aggregation scheme, the two partitions of this example combined represent a more balanced unit to schedule than the original superblock scheduler’s units of rows or columns (iterations). The number of nonzeros of our new scheduling unit is around 8 for both row and partition scheduling whereas that of the baseline row and column scheduling units varies in the range of 1 - 8. Note that our partitioning scheme does not split any superblock, this may introduce some imbalance. Imbalance also occurs in the last partition for the remaining. We also observe that the number of partitions in Figure 8 (b) could be easily smaller than the number of iterations in Figure 8 (a). This could lead to insufficient parallel degree for privatization strategy.

We record these partition-based partitions as blshr, compared with pilschr to record the partitioning positions. In practice, because the superblock sizes are relatively large, there are not many superblocks to schedule on our evaluation tensors. Therefore, it is not expected that these auxiliary metadata will increase the size of the overall HiCOO data structure.

5 EXPERIMENTS

5.1 Experimental Setup

Platform. We perform experiments on a Linux-based Intel Xeon E5-2698 v3 multicore server platform with 32 physical cores distributed on two sockets, each with 2.3 GHz frequency. The processor microarchitecture is Haswell, having 32 KiB L1 data cache and 128 GiB memory. The code artifact was written in the C language using OpenMP parallelization, and was compiled with icc 18.0.1.

Dataset. We use the sparse real-world applications, that appear in Table 2, ordered by decreasing nonzero density separately for third- and fourth-order tensors. Most of these tensors are included in The Formidable Repository of Open Sparse Tensors and Tools (FROSTT) dataset [39]. The darpa (source IP-destination IP-time triples), fb-m, and fb-s (short for “freebase-music” and “freebase-sampled”, entity-entity-relationship triples) are from the dataset of HaTen2 [15], and choa is built from the electronic health records (EHRs) of pediatric patients at Children’s Healthcare of Atlanta (CHOA) [34].

Configurations. We report the results under the best configurations for the following parameters for the highest MTtkRps performance with the three formats: (i) the number of reordering iterations, the superblock size L, and the block size B for HiCOO format; (ii) privatization or not for COO; and (iii) tiling or not for CSF. For HiCOO, B = 128 achieves the best results in most cases, and we use five reordering iterations which will be analyzed in § 5.8. All experiments use approximate rank of R = 16. The parallel experiments are run with 32 threads under dynamic scheduling strategy in units of one superblock/partition. We use the total execution time of MTtkRps in all modes for every tensor to calculate the speedup which is the ratio of the total MTtkRps time on a randomly reordered tensor over that using a specific reordering scheme. All the execution time is averaged over five runs.

Table 2: Description of sparse tensors.

<table>
<thead>
<tr>
<th>Tensors</th>
<th>Order</th>
<th>Dimensions</th>
<th>#Nonza</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>vast</td>
<td>3</td>
<td>165K × 11K × 2</td>
<td>26M</td>
<td>6.9 × 10⁻³</td>
</tr>
<tr>
<td>ncl2</td>
<td>3</td>
<td>12K × 9K × 29K</td>
<td>77M</td>
<td>2.4 × 10⁻⁵</td>
</tr>
<tr>
<td>choa</td>
<td>3</td>
<td>712K × 10K × 767</td>
<td>27M</td>
<td>5.0 × 10⁻⁶</td>
</tr>
<tr>
<td>darpa</td>
<td>3</td>
<td>22K × 22K × 24M</td>
<td>28M</td>
<td>2.4 × 10⁻⁹</td>
</tr>
<tr>
<td>fb-m</td>
<td>3</td>
<td>23M × 23M × 166</td>
<td>100M</td>
<td>1.1 × 10⁻⁹</td>
</tr>
<tr>
<td>fb-a</td>
<td>3</td>
<td>39M × 39M × 332</td>
<td>140M</td>
<td>1.7 × 10⁻¹⁰</td>
</tr>
<tr>
<td>flickr</td>
<td>3</td>
<td>320K × 28M × 2M</td>
<td>113M</td>
<td>7.8 × 10⁻¹²</td>
</tr>
<tr>
<td>deli</td>
<td>3</td>
<td>533K × 17M × 3M</td>
<td>140M</td>
<td>6.1 × 10⁻¹²</td>
</tr>
<tr>
<td>netf1</td>
<td>3</td>
<td>2.9M × 2.1M × 25M</td>
<td>144M</td>
<td>9.1 × 10⁻¹⁵</td>
</tr>
<tr>
<td>crime</td>
<td>4</td>
<td>6K × 24 × 77 × 32</td>
<td>5M</td>
<td>1.5 × 10⁻¹²</td>
</tr>
<tr>
<td>uber</td>
<td>4</td>
<td>185K × 2 × 14000 × 1717</td>
<td>3M</td>
<td>3.9 × 10⁻⁴</td>
</tr>
<tr>
<td>nips</td>
<td>4</td>
<td>2K × 3K × 14K × 17</td>
<td>3M</td>
<td>1.8 × 10⁻⁴</td>
</tr>
<tr>
<td>eron</td>
<td>4</td>
<td>6K × 6K × 24K × 1K</td>
<td>54M</td>
<td>5.5 × 10⁻⁹</td>
</tr>
<tr>
<td>flickr4d</td>
<td>4</td>
<td>320K × 28M × 2M × 731</td>
<td>113M</td>
<td>1.1 × 10⁻¹⁴</td>
</tr>
<tr>
<td>deli4d</td>
<td>4</td>
<td>533K × 17M × 3M × 1K</td>
<td>140M</td>
<td>4.3 × 10⁻¹⁵</td>
</tr>
</tbody>
</table>

5.2 HiCOO-MTtkRps with Reordering

Figure 9 (a) shows the speedup of the proposed reordering methods on sequential and parallel HiCOO-MTtkRps respectively.Lexi-ORDER reordering obtains 0.99–4.14× speedup (2.12x on average); while BFS-MCS reordering gets 0.99–1.88× speedup (1.34× on average). Lexi-ORDER and BFS-MCS do not behave as well on fourth-order tensors as on third-order tensors. Tensor flickr4d is constructed from the same data with flickr, with an extra short mode (refer to Table 2). Lexi-ORDER obtains 4.14× speedup on flickr while only 3.02× speedup on flickr4d. The same phenomenon is also observed on tensors deli and deli4d. This phenomenon indicates that it is harder to get good data locality on higher-order tensors, which will be justified in Table 3.

Figure 9: Reordered HiCOO-MTtkRps speedup over a random ordering implementation.

Figure 9 (b) shows the speedup of the proposed reordering methods on multicore parallel HiCOO-MTtkRps. We set the same superblock size L before and after reordering for a fair comparison. Overall, Lexi-ORDER results in 0.70–11.81× speedup (2.12x on average) for parallel HiCOO-MTtkRps; while BFS-MCS reordering gets 0.25–1.94× speedup (0.98x on average). Thus, the benefit from reordering using either Lexi-ORDER or BFS-MCS is generally less than that in the sequential case. This observation may indicate that
it is harder to pursue a good load balance after reordering. The 11.8X speedup achieved on tensor flickrd is because Lexi-ORDER changes the nonzero distribution and hence the optimal superblock size \( L \). Thus, under the same \( L \), reordering gives a big performance improvement. It indicates that automatically tuning the parameters of HiCOO will be very helpful.

### 5.3 HiCOO Parameters

We investigate two critical parameters of HiCOO [24]: the block ratio (\( \alpha_b \)) and the average slice size per tensor block (\( \bar{cy} \)). Smaller \( \alpha_b \) and larger \( \bar{cy} \) are favorable for good HiCOO-MTTKRP performance. Table 3 lists the parameter values for all tensors before and after Lexi-ORDER, the HiCOO-MTTKRP speedup (as shown in Figure 9), and the storage ratio of HiCOO over random ordering. Generally, when \( \alpha_b \) is reduced and \( \bar{cy} \) is increased using Lexi-ORDER for a tensor, we see a good performance speedup and storage ratio. (Sequential speedup reflects better by involving less factors than parallel cases.) For the same data in different orders, e.g., flickr4d and flickr, the \( \alpha_b \) and \( \bar{cy} \) values are the same for random reordering, after Lexi-ORDER, these values of flickr are better than those of flickr4d. This fact justifies the phenomenon that getting good data locality is harder for higher-order tensors.

Table 3: HiCOO parameters change before and after Lexi-ORDER reordering.

<table>
<thead>
<tr>
<th>Tensors</th>
<th>Random ordering</th>
<th>Lexi-ORDER</th>
<th>Speedup</th>
<th>Storage ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>vast</td>
<td>0.004</td>
<td>1.758</td>
<td>0.004</td>
<td>1.562</td>
</tr>
<tr>
<td>null2</td>
<td>0.020</td>
<td>0.314</td>
<td>0.008</td>
<td>0.074</td>
</tr>
<tr>
<td>china</td>
<td>0.089</td>
<td>0.057</td>
<td>0.016</td>
<td>0.056</td>
</tr>
<tr>
<td>darpa</td>
<td>0.796</td>
<td>0.009</td>
<td>0.018</td>
<td>0.113</td>
</tr>
<tr>
<td>fb-m</td>
<td>0.985</td>
<td>0.008</td>
<td>0.086</td>
<td>0.022</td>
</tr>
<tr>
<td>flickr</td>
<td>0.982</td>
<td>0.008</td>
<td>0.099</td>
<td>0.020</td>
</tr>
<tr>
<td>dali</td>
<td>0.999</td>
<td>0.008</td>
<td>0.097</td>
<td>0.022</td>
</tr>
<tr>
<td>reo</td>
<td>0.988</td>
<td>0.008</td>
<td>0.501</td>
<td>0.010</td>
</tr>
<tr>
<td>fb-m</td>
<td>0.998</td>
<td>0.008</td>
<td>0.744</td>
<td>0.009</td>
</tr>
</tbody>
</table>

5.4 Partition-Based Scheduler

Table 4 shows the load balance under three scenarios and the corresponding performance speedup on parallel HiCOO-MTTKRP over random ordering. Load balance is represented by the ratio of maximum number of nonzeros assigned to a thread to the average number of nonzeros per thread, where lower numbers are favored. The target maximum number of nonzeros of a partition is set to five times of the maximum number of nonzeros per superblock. The second column shows the balance ratio under a random ordering, which might be expected to have good load balance, at the price of poor locality. Most ratios in this column are under 1.3. The third column shows the balance ratios under Lexi-ORDER reordering scheme, which increase for most of tensors, some even reach 6.72. Despite these balance ratios, HiCOO-MTTKRP on reordered tensors obtains speedup over random ordering on most of tensors, because of its better data locality. The resulting balance ratios under the partition-based scheduling appear in the fourth column. Only tensors vast, fb-m, fb-s, and nips get better balance ratios compared to reordering-only, but better performance is achieved on about half of tensors including these three. The improvement of the Reordered+Balanced scheme over Reordered-only is up to 42%. Two factors may harm the load balance in our new scheduling. One is whether we can get sufficient amount of partitions for parallelization; the other is that the cubical superblocks limits the parallelism for highly irregular tensors. Since the balanced superblock scheme does not always improve performance, our final scheduler selects between these two options.

Table 4: Load balance and its effect before and after Lexi-ORDER tensor reordering and partition-based scheduling.

<table>
<thead>
<tr>
<th>Tensors</th>
<th>Max/Avg</th>
<th>Reordered</th>
<th>Reordered</th>
<th>Performance Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>vast</td>
<td>1.25</td>
<td>1.21</td>
<td>1.14</td>
<td>1.03</td>
</tr>
<tr>
<td>null2</td>
<td>1.18</td>
<td>1.24</td>
<td>1.36</td>
<td>1.04</td>
</tr>
<tr>
<td>china</td>
<td>1.13</td>
<td>1.26</td>
<td>1.32</td>
<td>1.07</td>
</tr>
<tr>
<td>darpa</td>
<td>1.57</td>
<td>1.35</td>
<td>1.55</td>
<td>1.50</td>
</tr>
<tr>
<td>fb-m</td>
<td>1.16</td>
<td>2.07</td>
<td>1.70</td>
<td>1.21</td>
</tr>
<tr>
<td>flickr</td>
<td>1.30</td>
<td>2.16</td>
<td>1.42</td>
<td>1.23</td>
</tr>
<tr>
<td>dali</td>
<td>1.23</td>
<td>3.62</td>
<td>4.46</td>
<td>3.66</td>
</tr>
<tr>
<td>reo</td>
<td>1.28</td>
<td>5.00</td>
<td>5.90</td>
<td>0.83</td>
</tr>
</tbody>
</table>

5.5 Reordering Effect on Other Formats

5.5.1 COO-MTTKRP with Reordering. We show the effect of the two reordering approaches on sequential and parallel COO-MTTKRP from PartII [22] in Figure 10. This COO-MTTKRP is implemented in C and OpenMP parallelized with or without privatization determined for different modes, using the same algorithm with Tensor Toolbox [5]. For any reordering approach, after doing a BFS-MCS, Lexi-ORDER, or random reordering on the input tensor, we still sort the tensor in the mode order of 1 \( \times \) 79 \( \times \) \( \cdots \) \( \times \) N. Observe that Lexi-ORDER improves sequential COO-MTTKRP performance by 1.00–4.29X (1.79X on average), while BFS-MCS gets 0.95–1.27X (1.11X on average). Also, Lexi-ORDER improves parallel COO-MTTKRP performance by 1.01–1.48X (1.21X on average), while BFS-MCS improves by 0.70–1.68X (1.11X on average). Note that Lexi-ORDER improves the performance of COO-MTTKRP for all tensors, both in sequential (79%) and parallel (21%). We conclude that this ordering is always helpful for COO-MTTKRP, while the improvements being less than what we saw for HiCOO-MTTKRP.

5.5.2 CSF-MTTKRP with Reordering. We show the effect of the two reordering approaches on sequential and parallel CSF-MTTKRP from...
SPLATT v1.1.1 [45] in Figure 11. CSF-MTTKRP is set to use all CSF representations (ALLMODE) for MTTKRP in all modes and with tiling option on. 2 Lexi-Order improves sequential CSF-MTTKRP performance by 0.65–2.33× (1.50× on average) and accelerates parallel CSF-MTTKRP by 0.86–1.88× (1.27× on average). BFS-MCS improves sequential CSF-MTTKRP performance by 1.00–1.86× (1.22× on average) and accelerates parallel CSF-MTTKRP by 0.59–1.36× (1.04× on average). Both ordering approaches improve the performance of CSF-MTTKRP on average. While BFS-MCS is always helpful in the sequential case, Lexi-Order is not helpful on only one tensor crime. The improvements achieved by the two reordering approaches for CSF are less than those for HiCOO and COO formats. We conclude that both reordering methods are helpful for CSF-MTTKRP, but in a lesser extend than for HiCOO and COO based MTTKRP.

5.5.3 Format Comparison. Figure 12 compares the parallel performance of COO-, CSF-, and HiCOO-MTTKRP all using Lexi-Order. The performance is shown by the speedup of each implementation to reordered COO-MTTKRP for every tensor. HiCOO achieves the best performance for most third-order tensors and is better than COO and CSF with tiling for most cases. In these experiments, we found that SPLATT without tiling option is even faster than with tiling especially on fourth-order tensors. After reordering, HiCOO is 0.74–7.21× (3.08× on average) faster than COO and 0.25–28.84× (5.29× on average) faster than CSF-tiling. Compared to CSF with no tiling, HiCOO obtains 0.19 – 5.30× (1.40× on average) speedup. On flickr, deli, nell1, and enron data, HiCOO behaves worse than both the CSF settings because of the tensors’ hypersparseness property. Although the reordering methods have improved the nonzero locality on them, due to their severe load imbalance in Table 4, their performance is still not comparable to CSF format. Overall, as also stated in [24], HiCOO works the best on short tensor modes and with reasonable sparsity of tensor blocks; while CSF is the complimentary for hypersparse tensors.

5.6 Reordering Methods Comparison

As seen above, Lexi-Order improves performance more than BFS-MCS in most cases. Compared to the reordering method used in

---

2Our tests using one CSF representation (0ONEMODE) show quite similar results of reordering effect to the ALLMODE setting. Same for using tiling or no tiling options.

3HiCOO shows more performance advantage with more CPU cores as illustrated in [24].
5.8 Effect of the Number of Iterations in Lexi-Order

Since Lexi-Order improves the ordering iteratively, we evaluate the effect of the number of iterations on HiCOO-MTTKRP performance. The results appear in Figure 15 (a), which is normalized to the runtime of 10 iterations. MTTKRP on most tensors does not vary a lot by setting different number of iterations, except cast, nell2, uber, and nips. We use 5 iterations to get good MTTKRP performance similar to that of 10 iterations, with about half of the overhead (shown in Figure 15 (b)). But 3 or fewer iterations will get an acceptable performance when users care much about the pre-processing time.

![Figure 15: The performance and overhead of different numbers of iterations.](image)

5.9 CPD Application

Figure 16 shows the Lexi-Order effect on the parallel HiCOO-CPD algorithm using alternating least squares [19]. Similar to the numbers in Figure 9 (b), CPD after reordering achieves 6.5–10.44× speedup (1.81× on average). Reordering helps us to enhance the performance of a whole tensor application.

![Figure 16: CPD performance on reordered tensors.](image)

6 RELATED WORK

Plenty recent research studied the optimization of tensor algorithms [6, 8–10, 12, 13, 16, 19, 20, 23, 25, 27, 29, 38, 40]. Our work emphasizes on reordering schemes to get better nonzero structure. Various reordering methods have been considered for matrix algebra [2, 17, 28, 35, 36, 48]. For tensor operations, Smith et al. [44] proposed two reordering methods based on partitioning: the first is the partitioning of a graph that models the interactions between slices; the second is the partitioning of a hypergraph that models the memory access of the MTTKRP operation. A tensor reordering is induced such that the vertices in the same partition are set to consecutive labels. Hypergraphs have also been used to efficiently parallelize MTTKRP in distributed memory systems [18]. These two hypergraphs are constructed differently from our BFS-MCS, they use fibers and tasks as vertices respectively, while our hypergraph takes tensor indices as vertices. Besides, we propose matLexiOrder, an alternative doubly lexical ordering method for sparse matrices, and design Lexi-Order for sparse tensors based on it.

7 CONCLUSION

Motivated by an interest in further improving the HiCOO implementation, we investigated the problem of reordering a tensor to improve block density for tensor computations. Inspired by algorithms for sparse matrices, we proposed two heuristics, BFS-MCS and Lexi-Order. BFS-MCS is based on the maximum cardinality search principle; Lexi-Order is based on matLexiOrder, which we propose here as an alternative to existing doubly lexical ordering method for sparse matrices. matLexiOrder has near linear runtime per tensor dimension, rendering the overall tensor reordering method Lexi-Order practical. Lastly, we improve the superblock scheduling strategy of HiCOO [24] with a partition-based scheme, which eases the side effect of increased load imbalance that can occur after reordering, thereby improving MTTKRP performance for several sample inputs.

Overall, BFS-MCS is asymptotically faster while Lexi-Order is more effective. Lexi-Order obtains performance speedup on HiCOO-MTTKRP of 2.12× on average for both sequential and multicore parallel implementations. For COO and CSF-MTTKRP, the average sequential and parallel speedups are 1.79×, 1.21× and 1.50×, 1.27×, respectively. Users can control a parameter of Lexi-Order to reduce the reordering overhead without much performance drop and also tune the HiCOO parameters for the reordered tensors to pursue the highest MTTKRP performance. As future work, we plan to pursue the automatic performance tuning for different sparse tensor formats, HiCOO parameters, the tradeoff of reordering methods and parallel strategy parameters, and improve the runtime of Lexi-Order. Our proposed heuristics set a basis for improving MTTKRP performance through tensor reordering. Furthermore, we anticipate the development of additional heuristics for the general tensor reordering problem. Additionally, we expect reordered tensors will prove useful for any tensor operation in which the memory access behavior depends on the locality of tensor indices, as in MTTKRP. Examples include Tensor-times-matrix and tensor-times-vector operations in Tucker decompositions and higher-order power methods, respectively. Our approaches will be especially useful as local improvements for intra-node data locality in large HPC environment.

ACKNOWLEDGMENTS

This research was partially funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation". Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830. This research was also funded in part under Defense Advanced Research Projects Agency (DARPA) contract FA8750-18-2-0108, under the DARPA MTO Software Defined Hardware program, and the Laboratory Directed Research and Development program at Sandia National Laboratories under contract DE-NA-0003525. Disclaimer: The views, opinions, and/or findings contained in this document are those solely of the author(s) and should not be interpreted as representing the official views or policies of any of its funding sources.