## 2PCP: Two-phase CP decomposition for billion-scale dense tensors

(Article begins on next page)

20 April 2024

# 2PCP: Two-Phase CP Decomposition for Billion-Scale Dense Tensors

Xinsheng Li[†]
Arizona State University
Tempe, AZ 85287, USA
Email: lxinshen@asu.edu

Shengyu Huang[†]
Arizona State University
Tempe, AZ 85287, USA
Email: shengyu.huang@asu.edu

K. Selçuk Candan
Arizona State University
Tempe, AZ 85287, USA
Email:candan@asu.edu

Maria Luisa Sapino
University of Torino
I-10149 Torino, Italy
marialuisa.sapino@unito.it

*Abstract*—Tensors are multi-dimensional arrays – consequently, tensor decomposition operations (CP and Tucker) are the bases for many high-dimensional data analysis tasks, from clustering, trend detection, anomaly detection, to correlation analysis in various application domains, including science and engineering[1]. One key problem with tensor decomposition is its computational complexity and space requirements. Especially, as the relevant data sets get denser, in-memory schemes for tensor decomposition become increasingly ineffective; therefore out-of-core (secondary-memory supported, potentially parallel) computing is necessitated. However, existing techniques do not consider the I/O and network data exchange costs that out-of-core execution of the tensor decomposition operation will incur. In this paper, we note that when this operation is implemented with the help of secondary-memory and/or multiple servers to tackle the memory limitations, we would need intelligent *buffer-management* and *task-scheduling* techniques which take into account the cost of bringing the relevant blocks into the buffer to minimize I/O in the system. In this paper, we introduce 2PCP, a two-phase, block-based CP decomposition system with intelligent *buffer sensitive task scheduling* and *buffer management* mechanisms. 2PCP aims to reduce I/O costs in the analysis of relatively dense tensors common in scientific and engineering applications. Experiment results compare with current state of art tensor decomposition algorithms and show that our algorithms can significantly reduce the amount of I/O and execution time while maintaining decomposition accuracy.

## I. Introduction

Tensors are multi-dimensional arrays. Thanks to the availability of various mathematical tools (such as decompositions) that support multi-aspect analysis of data, tensors are increasingly being used for representing multi-dimensional data, such as sensor streams and social networks [29], [18], [14], [17], [19]. Matrix-shaped data (i.e., 2-mode tensors) are often analyzed for their latent semantics through matrix decomposition operations, such as singular value decomposition (SVD). The corresponding analysis operation which applies to tensors with more than two modes is known as the *tensor decomposition* operation such as CANDECOMP [7] and PARAFAC [11] decompositions (known as the CP decomposition, Figure 1), and Tucker decomposition [30]. These operations form the basis for many data analysis and knowledge discovery tasks.
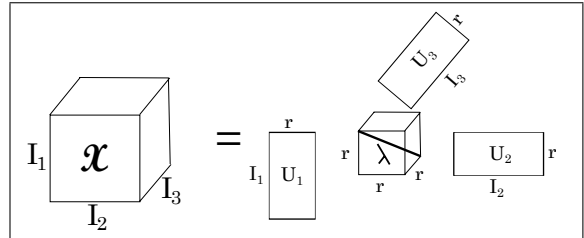


Fig. 1. CP-decomposition of a 3-mode tensor [7], [11] results in a diagonal core and three factor matrices

### A. Challenge: Memory Blow-Up

A major challenge with tensor decomposition, however, is its computational and space complexities – especially for *dense* data sets[2]. While the process is relatively faster for *sparse* tensors, decomposition is still a major bottleneck in many applications [10]: Tensor decomposition process results in dense (and hence large) intermediary data, even when the input tensor is sparse (and hence small). This is known as the *intermediate memory blow-up problem* [14] and renders purely in-memory implementations of tensor-decomposition impractical, for both CP and Tucker decompositions [23].

### B. Block-based and Parallel Tensor Decompositions

As the relevant data sets get large, existing in-memory schemes for tensor decomposition become increasingly ineffective and block-based solutions where some (possibly intermediate) data may be materialized on disks (instead of main memory) or other servers contributing to the decomposition process are necessitated. Several implementations of tensor decomposition operations on disk-resident data sets have been proposed. GridPARAFAC [22], for example, partitions the tensor into pieces, obtains decomposition for each piece (potentially in parallel), and stitches the partial decomposition results into a combined decomposition for the initial tensor through an iterative improvement process (Figure 2). TensorDB [15], [16] leverages a block-based framework to store and retrieve data, extends array operations to tensor operations, and introduces optimization schemes for in-database tensor decomposition. HaTen2 [13] focuses on sparse tensors and presents a scalable tensor decomposition suite of methods for Tucker and PARAFAC decompositions on the MapReduce framework. In all these systems, I/O costs is an inevitable problem as they need I/O to fetch data either from disk or from the network. As we experimentally verify in this paper, the I/O or communication overhead of iterative algorithms

[†] indicates student authors (with equal contributions).

[2]Such dense tensors are common in science and engineering: ensemble simulations, for example, are created by sampling the domains of the relevant input parameters, and recording simulation results for each configuration [8].
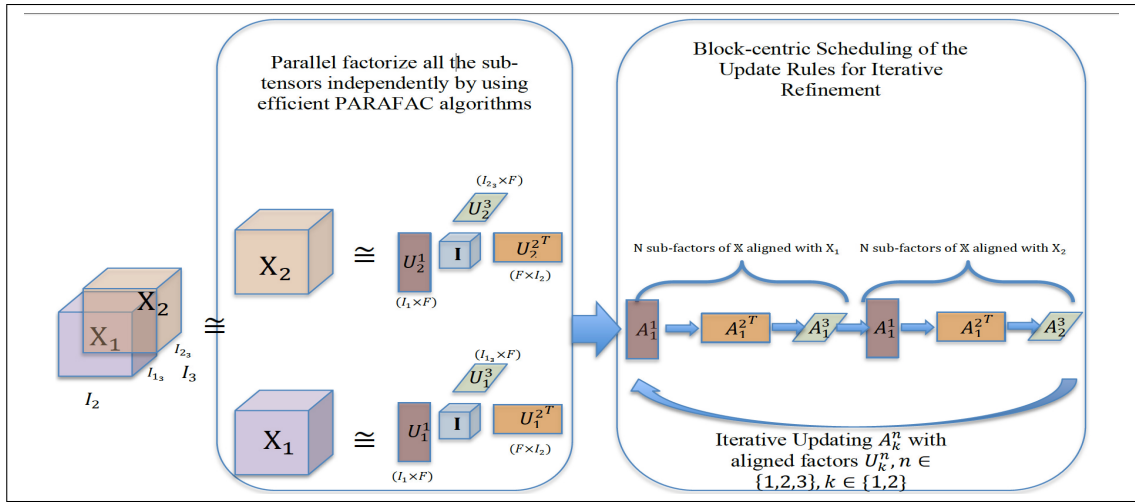
Fig. 2. Illustration of the two-phase, block-based tensor decomposition: the input tensor is partitioned into smaller blocks, each block is decomposed (potentially in parallel), and the partial decompositions are stitched together through an iterative improvement process (the notation is introduced in Section III)

(especially on a distributed platform like MapReduce) can be very expensive. In addition, naive implementations of the block-based iterative improvement algorithms can result in significant I/O, when the buffer memory is not large enough to hold the entire intermediary data. Consequently, reducing these I/O and communication costs, especially for dense tensors common in science and engineering, is a critical challenge.

### C. 2PCP and Re-Use Promoting Scheduling of Block Accesses

In this paper, we propose 2PCP, a two-phase CP tensor decomposition mechanism. As we see in Figure 2, two-phase block-based tensor decomposition can help reduce the memory-blow-up problem as the first phase requires decomposition of much smaller tensors. However, the number of the (so called factor) matrices that are produced in the first phase and the intermediary data generated while these are stitched together through an iterative process in the second phase may still be quite large. Consequently, the intermediary data may still take too much space to be fully memory-resident and may need to be brought to the memory on on-demand basis. Consequently, the 2PCP system we present in this paper complements the basic two-phase CP tensor decomposition approach with novel *data re-use promoting block scheduling* and *buffer management* mechanisms to address this difficulty:

• In its first phase, 2PCP partitions the input data to blocks (or sub-tensors), then conducts ALS on each sub-tensor (potentially in parallel using a MapReduce based platform) independently (Section IV).

• In the second phase, which is executed on a single worker machine, 2PCP leverages fine-grained block centric iterative refinement with a novel forward looking buffer replacement strategy that helps improve buffer utilization and reduce I/O:

  • In particular, we extend the conventional mode-centric approach in a way that enables more flexible, fine-grained, block-centric scheduling of updates and the corresponding data accesses (Section V).

  • Given this *fine-grained* block-centric iterative improvement scheme, we then consider alternative scheduling techniques that can maximize the utility

of the intermediary-data already in the buffer (Section VI).

  • We then propose and study alternative buffer replacement policies complementing the different scheduling techniques considered above and develop a forward-looking, predictive buffer replacement strategy that matches the proposed scheduling techniques to further push the I/O costs down (Section VII).

In the next section, we present the related work. In Section III, we present the relevant background and notations and we formalize the problem. Experiment results, reported in Section VIII, show that the proposed algorithms significantly reduce the amount of I/O as well the execution time. We conclude the paper in Section IX.

## II. RELATED WORK

Tensor based representations of data and tensor decompositions (especially the two widely used decompositions CP [11] and Tucker [30]) are proven to be effective in multi-aspect data analysis for capturing high-order structures in multi-dimensional data [17], [28]. There are two widely used toolboxes: the *Tensor Toolbox for Matlab* [4] (for sparse tensors) and *N-way Toolbox for Matlab* [3] (for dense tensors).

Since tensor decomposition is a costly process for both sparse and tensors, various optimization and parallel algorithms and systems have been developed. [18] proposed a memory-efficient Tucker (MET) decomposition to address the intermediate blowup problem in Tucker decomposition by updating a subset of the modes at a time. [29] proposed MACH, a randomized algorithm (based on randomized sampling) that speedups the Tucker decomposition while providing accuracy guarantees. Recently, [21] proposed a fast approach for CP that decomposes an unfolded tensor in lower order, instead of directly factorizing the high order tensor. TensorDB [15], [16] extends a block-based array store to store and retrieve data and introduces optimization schemes for efficient CP-ALS based in-database tensor decompositions. [19] proposed a novel Personalized Tensor Decomposition (PTD) mechanism that boosts accuracy and reduces execution time in situations where the user's interest is not uniformly distributed across the whole tensor.

Parallelization of tensor decompositions have been proposed for different platforms [5], [9], [27]. In [23], authors proposed a two stage (partition and merge) scheme for implementing the CP decomposition in a parallizable manner. [31] introduced various parallelization strategies such as distributing a large tensor onto the servers in a cluster, minimizing data exchange, and limiting the memory needed for storing matrices or tensors, to speed up factor matrix update step in tensor decomposition. In [24], [25], authors propose PARCUBE, a sampling based, parallel and sparsity promoting, approximate PARAFAC decomposition scheme. [13] proposed HaTen2, a massively distributed MapReduce based implementation of PARAFAC and Tucker running on the MapReduce platform. HaTen2 focuses on sparse tensors and carefully reorders the operations and exploits the sparsity of real world tensors. Yet, HaTen2 may still result in high communication costs during iterative update process.

## III. BACKGROUND AND NOTATIONS

We now present the relevant background and notations and formalize the problem we consider in this paper.

### A. Tensors and Tensor Decompositions

Tensors are generalizations of matrices: while a matrix is essentially a 2-mode array, a tensor is an array of larger number of modes. Intuitively, the tensor model maps a relational schema with $N$ attributes to an $N$-modal array (where each potential tuple is a tensor cell).

The two most popular tensor decomposition algorithms are the Tucker [30] and the CANDECOMP/PARAFAC (CP) [11] decompositions. Intuitively, both generalize singular value matrix decomposition (SVD) to tensors.

### B. CP Decomposition

As shown in Figure 1, given a tensor $\mathcal{X}$, CP factorizes the tensor into $F$ component matrices (where $F$ is a user supplied non-zero integer value also referred to as the *rank* of the decomposition). For the simplicity of the discussion, let us consider a 3-mode tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. CP would decompose $\mathcal{X}$ into three matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, such that

$$\mathcal{X} \approx \tilde{\mathcal{X}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}] \equiv \sum_{f=1}^{F} a_f \circ b_f \circ c_f,$$

where $a_f \in \mathbb{R}^I$, $b_f \in \mathbb{R}^J$ and $c_f \in \mathbb{R}^K$. The factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are the combinations of the rank-one component vectors into matrices; e.g., $\mathbf{A} = [\ a_1\ a_2\ \cdots\ a_F\ ]$.

An alternating least squares (ALS) method is mainly used in many algorithms for tensor decomposition: at each iteration, ALS estimates one factor matrix while maintaining other matrices fixed; this process is repeated for each factor matrix associated to the modes of the input tensor until convergence condition is reached. Since tensor decomposition is an approximation algorithm, the new tensor $\tilde{\mathcal{X}}$ obtained by recomposing the factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ is often different from the input tensor, $\mathcal{X}$. The accuracy of the decomposition is often measured by considering the Frobenius norm of the difference tensor:

$$accuracy(\mathcal{X}, \tilde{\mathcal{X}}) = 1 - error(\mathcal{X}, \tilde{\mathcal{X}}) = 1 - \left( \frac{\|\tilde{\mathcal{X}} - \mathcal{X}\|}{\|\mathcal{X}\|} \right).$$

### C. Block-based CP Decomposition

As discussed previously, block-based CP decomposition techniques partition the given tensor into blocks, initially decompose each block independently, and then iteratively combine these decompositions into a final composition.

Let us consider an $N$-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\mathfrak{X} = \{\mathcal{X}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes. Without loss of generality, let us assume that $\mathcal{K}$ partitions the mode $i$ into $K_i$ equal partitions; i.e., $|\mathcal{K}| = \prod_{i=1}^{N} K_i$. Let us also assume that we are given a target decomposition rank, $F$, for the tensor $\mathcal{X}$. Let us further assume that each sub-tensor in $\mathfrak{X}$ has already been decomposed with target rank $F$ and let $\mathfrak{U}^{(i)} = \{U_{\vec{k}}^{(i)} \mid \vec{k} \in \mathcal{K}\}$ denote the set of $F$-rank sub-factors[3] corresponding to the sub-tensors in $\mathfrak{X}$ along mode $i$. In other words, for each $\mathcal{X}_{\vec{k}}$, we have

$$\mathcal{X}_{\vec{k}} \approx I \times_1 U_{\vec{k}}^{(1)} \times_2 U_{\vec{k}}^{(2)} \cdots \times_N U_{\vec{k}}^{(N)}, \qquad (1)$$

where $I$ is the $N$-mode $F \times F \times \ldots \times F$ identity tensor, where the diagonal entries are all 1s and the rest are all 0s.

Given these, [22] presents an iterative improvement algorithm for composing these initial sub-factors into the full $F$-rank factors, $A^{(i)}$ (each one along one mode), for the input tensor, $\mathcal{X}$. The outline of this block based process is as follows: Let us partition each factor $A^{(i)}$ into $K_i$ parts corresponding to the block boundaries along mode $i$:

$$A^{(i)} = [A_{(1)}^{(i)T} A_{(2)}^{(i)T} ... A_{(K_i)}^{(i)T}]^T.$$

Given this partitioning, each sub-tensor $\mathcal{X}_{\vec{k}}$, $\vec{k} = [k_1, \ldots, k_i, \ldots, k_N] \in \mathcal{K}$ can be described in terms of these sub-factors (Figure 3):

$$\mathcal{X}_{\vec{k}} \approx I \times_1 A_{(k_1)}^{(1)} \times_2 A_{(k_2)}^{(2)} \cdots \times_N A_{(k_N)}^{(N)} \qquad (2)$$

Moreover [22] shows that the current estimate of the sub-factor $A_{(k_i)}^{(i)}$ can be revised using the update rule (for more details on the update rules please see [22]):

$$A_{(k_i)}^{(i)} \longleftarrow T_{(k_i)}^{(i)} \left(S_{(k_i)}^{(i)}\right)^{-1} \qquad (3)$$

where

$$T_{(k_i)}^{(i)} = \sum_{\vec{l} \in \{[*,\ldots,*,k_i,*,\ldots,*]\}} U_{\vec{l}}^{(i)} \left(P_{\vec{l}} \oslash (U_{\vec{l}}^{(i)T} A_{(k_i)}^{(i)})\right)$$

$$S_{(k_i)}^{(i)} = \sum_{\vec{l} \in \{[*,\ldots,*,k_i,*,\ldots,*]\}} Q_{\vec{l}} \oslash \left(A_{(k_i)}^{(i)T} A_{(k_i)}^{(i)}\right)$$

such that, given $\vec{l} = [l_1, l_2, \ldots, l_N]$, we have

- $P_{\vec{l}} = \circledast_{h=1}^{N}(U_{\vec{l}}^{(h)T} A_{(l_h)}^{(h)})$ and

- $Q_{\vec{l}} = \circledast_{h=1}^{N}(A_{(l_h)}^{(h)T} A_{(l_h)}^{(h)})$.

Above, $\circledast$ denotes the Hadamart product and $\oslash$ denotes the element-wise division operation.

---

[3]If the sub-tensor is empty, then the factors are $\mathbf{0}$ matrices of the appropriate size.
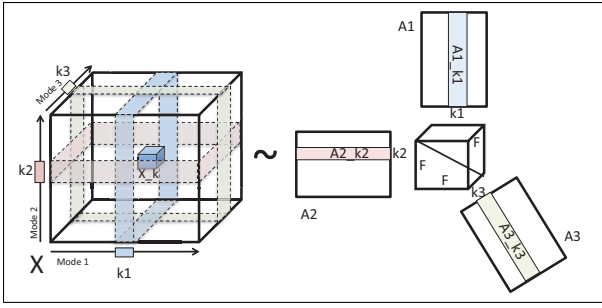
Fig. 3. Each sub-tensor (or block) can be described in terms of the corresponding sub-factors

While the precise derivation of the above update rule is not critical for our discussion (and is beyond the scope of this paper), it enables us to formulate the 2PCP two-phase, block-based tensor decomposition process.

## IV. OVERVIEW OF 2PCP

The outline of the proposed two-phase, block-based tensor decomposition algorithm, 2PCP is presented in Algorithm 1 and visualized in Figures 2 and 4.

*Example 1:* In Figure 2, the given tensor $\boldsymbol{\mathcal{X}}$ is partitioned into two sub-tensors $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$:

- **Phase 1:** In phase 1, each sub-tensor is decomposed with a standard PARAFAC algorithm. In the example, sub-factors $\boldsymbol{U}_{(1)}^{(1)}, \boldsymbol{U}_{(1)}^{(2)}, \boldsymbol{U}_{(1)}^{(3)}$ of sub-tensor $\boldsymbol{\mathcal{X}}_1$ and sub-factors $\boldsymbol{U}_{(2)}^{(1)}, \boldsymbol{U}_{(2)}^{(2)}, \boldsymbol{U}_{(2)}^{(3)}$ of sub-tensor $\boldsymbol{\mathcal{X}}_2$ are generated in the first stage.

- **Phase 2:** In the second phase, the sub-factors $\boldsymbol{U}_{\vec{k}}^{(i)}$ of each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ are used for iteratively refining the sub-factors $\boldsymbol{A}_{(k_i)}^{(i)}$ of the input tensor $\boldsymbol{\mathcal{X}}$.

### A. Key Observations

The pseudo code presented in Algorithm 1 supports the following key observations:

**● Observation #1 (Independent/Parallel Sub-tensor Decomposition in Phase 1):** Each sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ can be decomposed (in parallel) independently from the others. This ensures that the proposed system can handle very large tensors as long as the input tensor is partitioned into several blocks in such a way that each block can be decomposed with the available memory. Moreover, this phase is easy to parallelize using, for example, the popular distributed computing framework, MapReduce, using the following `map` and `reduce` operators (assuming a three-mode input tensor):

- `map`: $\langle \boldsymbol{b}, i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$ on $\boldsymbol{b}$. Here, $\boldsymbol{b}$ is the sub-tensor id, $i$, $j$, $k$ together give the coordinate of sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$. Tuples with the same $b$ are shuffled to the same reducer in the form of $\langle key : \boldsymbol{b}, values : i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$.

- `reduce` $\langle key : \boldsymbol{b}, values : i, j, k, \boldsymbol{\mathcal{X}}(i, j, k)\rangle$: The reducer processing the key $\boldsymbol{b}$ receives the non-zero elements of sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$. It recomposes the

---

**Algorithm 1** The outline of the 2PCP block-based iterative improvement process

**Input**: original tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, and decomposition rank, $F$

**Output**: CP tensor decomposition $\boldsymbol{\mathcal{\check{X}}}$

1) Phase 1: for all $\vec{k} \in \mathcal{K}$
   - decompose $\boldsymbol{\mathcal{X}}_{\vec{k}}$ into $\boldsymbol{U}_{\vec{k}}^{(1)}, \boldsymbol{U}_{\vec{k}}^{(2)}, \ldots, \boldsymbol{U}_{\vec{k}}^{(N)}$

2) Phase 2: repeat
   a) for each mode $i = 1$ to $N$
      i) for each modal partition, $k_i = 1$ to $K_i$,
         A) update $\boldsymbol{A}_{(k_i)}^{(i)}$ using $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$, for each block $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$; more specifically,
            - compute $\boldsymbol{T}_{(k_i)}^{(i)}$, which involves the use of $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$)
            - revise $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ using $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ and $\boldsymbol{A}_{(k_i)}^{(i)}$
            - compute $\boldsymbol{S}_{(k_i)}^{(i)}$ using the above
            - update $\boldsymbol{A}_{(k_i)}^{(i)}$ using the above
            - for each $\vec{k} = [*, *, \ldots, k_i, \ldots, *, *]$
              ○ update $\boldsymbol{P}_{\vec{k}}$ and $\boldsymbol{Q}_{\vec{k}}$ using
              ○ $\boldsymbol{U}_{\vec{k}}^{(i)}$ and $\boldsymbol{A}_{(k_i)}^{(i)}$
      until stopping condition
3) Return $\boldsymbol{\mathcal{\check{X}}}$

---

sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$. Then sub-tensor $\boldsymbol{\mathcal{X}}_{\vec{k}}$ is decomposed into sub-factor $\boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}$, where $\boldsymbol{n}$ is the mode id, by using PARAFAC. Finally, reducer emits each sub-factor $\boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}$ as an independent file, with content $\langle key : \boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}, value : i, j, \boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}(i, j)\rangle$. Here, $i$, $j$ are the coordinates of sub-factor $\boldsymbol{U}_{\boldsymbol{b}}^{\boldsymbol{n}}$.

.

**● Observation #2 (In-place Iterative Refinement in Phase 2):** As shown in Algorithm 1, once Phase 1 is completed, in the second phase, each $\boldsymbol{A}_{(k_i)}^{(i)}$ can be maintained by computing and revising $\boldsymbol{T}_{(k_i)}^{(i)}$ and $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ incrementally. Note that this incremental update process presented in Algorithm 1 is logically equivalent to the one presented in [22], but includes a significant structural difference: in [22], $P$ and $Q$ are updated using a separate loop for each mode to optimize for parallelism, whereas Algorithm 1 updates $P$ and $Q$ in-place to significantly reduce the amount of disk accesses.

The total space requirement during this iterative refinement process is governed by the sizes of the $F$-rank partial factors, $\boldsymbol{A}_{(k_i)}^{(i)}$, and the corresponding, $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$, for each mode $i$ of the given tensor $\boldsymbol{\mathcal{X}}$; i.e., memory requirement $mem_{total}(\boldsymbol{\mathcal{X}})$ can be computed as

$$\sum_{i=1}^{N} K_i \times \left( \underbrace{\left(\frac{I_i}{K_i} \times F\right)}_{\boldsymbol{A}_{(k_i)}^{(i)}} + \underbrace{\left(\left(\prod_{j \neq i} K_j\right) \times \frac{I_i}{K_i} \times F\right)}_{total \ for \ \boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}} \right)$$
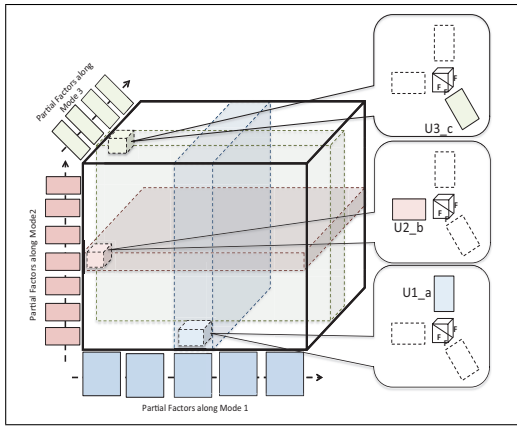
Fig. 4. The outline of the mode-centric iterative improvement algorithm proposed in [22]: the $k^{th}$ partial factor along mode $i$ is updated using the mode $i$ partial factors of the blocks aligned with the $k^{th}$ mode partition along mode $i$

or equivalently as

$$mem_{total}(\boldsymbol{\mathcal{X}}) \sim \sum_{i=1}^{N} \left( \left(1 + \prod_{j \neq i} K_j\right) \times I_i \times F \right).$$

This implies that the total memory requirement increases quickly with the number of partitions considered. Since, when large tensors are considered, the number of partitions themselves may need to be large (to ensure that each resulting block can be decomposed using the available memory), $mem_{total}(\boldsymbol{\mathcal{X}})$ can go beyond the available memory.

• **Observation #3 (On-demand Per Mode-Partition (MP) Data Access in Phase 2):** Fortunately, during this iterative refinement process we do not need all this data in the memory simultaneously. Incrementally maintaining $\boldsymbol{T}_{(k_i)}^{(i)}$ and $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}$ require bringing <u>only</u> $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$) and (the old value of) $\boldsymbol{A}_{(k_i)}^{(i)}$ to the main memory. In fact, Phase 2 can be executed by considering each mode-partition individually and bringing to the memory only the relevant partial factors. In other words, for maintaining each $\boldsymbol{A}_{(k_i)}^{(i)}$, we need

$$mem_{MP}(\boldsymbol{\mathcal{X}}, \boldsymbol{A}_{(k_i)}^{(i)}) \sim \left(1 + \prod_{j \neq i} K_j\right) \times \frac{I_i}{K_i} \times F$$

units of memory.

• **Observation #4 (Challenge - Naive Execution Increases I/O):** While the above observations show that using the proposed two phase Algorithm 1, we are able to significantly reduce the amount of memory needed to decompose large tensors. A naive implementation of this strategy, however, may require significant amount of I/O: After each mode partition is processed, to open space for the data needed to process the next mode partition, (a) all updated data need to be written back to the disk and (b) data relevant to the next mode partition need to be read from the disk into the memory, resulting in $\sum_{i=1}^{N} K_i$ data swap operations for a single iteration of the algorithm. A more reasonable alternative would be to use the memory as a
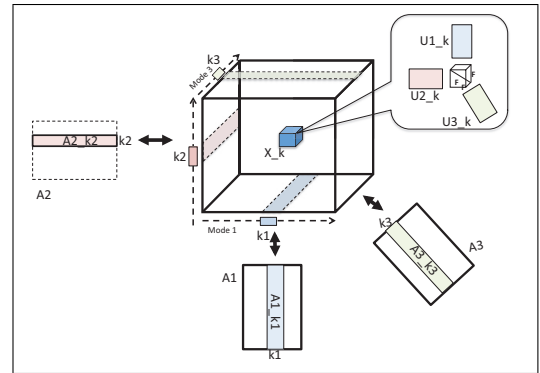


Fig. 5. For any block $\boldsymbol{\mathcal{X}}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$, its factors $\boldsymbol{U}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}^{(1)}$ through $\boldsymbol{U}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}^{(N)}$ can be used for maintaining $N$ sub-factors of $\boldsymbol{\mathcal{X}}$, one along each of the $N$ modes.

cache for mode-partition data and only swap data in and out of the buffer if the cache is full.

### B. Problem Statement: Re-Use Promoting Data Access and Buffer Management During Iterative Refinement Phase

As we have seen above, in the second phase of 2PCP, the overall efficiency of the process depends highly on the effectiveness of the buffer utilization. Therefore, the key problem that needs to be addressed to improve the efficiency of 2PCP is to improve the utilization of the buffer through *re-use promoting data access and buffer management* during the iterative improvement process.

In the rest of the paper, we introduce how 2PCP addresses this problem: we first (a) present an alternative fine-grained block-centric iterative refinement scheme, next (b) we consider alternative block-scheduling techniques leveraging this block-centric update process, and then (c) we investigate corresponding buffer replacement strategies to help improve the buffer utilization and reduce I/O costs.

### V. BLOCK-CENTRIC SCHEDULING OF ITERATIVE IMPROVEMENT PROCESS

As we have seen in the previous section (Algorithm 1 and visualized in Figure 4), the conventional way to perform the iterative refinement process of the block-based CP involves considering each mode, $i$, separately. In this <u>*mode-centric*</u> scheme, for the $k_i^{th}$ partition of mode $i$, we then maintain $\boldsymbol{A}_{(k_i)}^{(i)}$ using, for all $1 \leq j \leq N$, the current estimates for $\boldsymbol{A}_{(k_j)}^{(j)}$ and the decompositions in $\boldsymbol{\mathfrak{U}}^{(j)}$; i.e., the $F$-rank sub-factors of the sub-tensors in $\boldsymbol{\mathfrak{X}}$ along different modes.

As we experimentally validate in Section VIII, however, this conventional scheme may result in a significant amount of I/O (for swapping data in and out of memory) if the available memory is not sufficient to buffer all the data. Intuitively, this is because the order in which the update rules are applied does not lend itself into significant amount of data sharing. In this section, we present an alternative <u>*block-centric*</u> way to implement the process of iterative refinement; as we see later in the paper, this alternative process lends itself to better data sharing and memory utilization, thereby helping reduce the overall I/O costs.
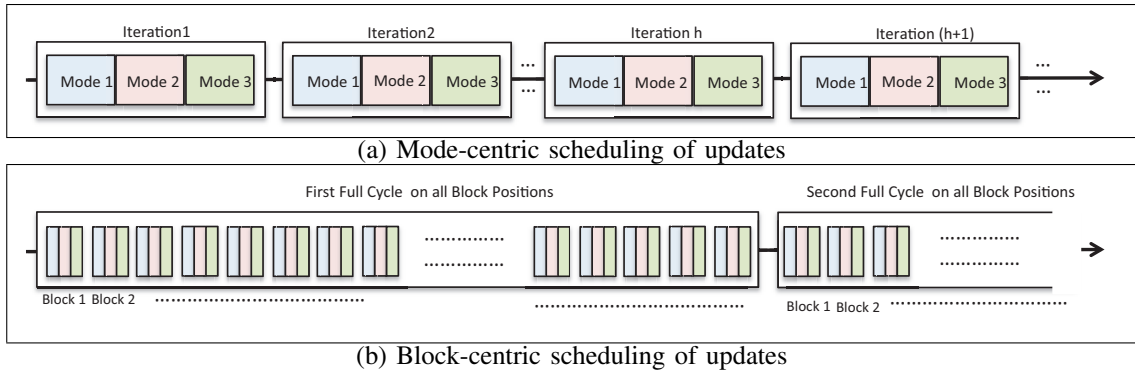
(a) Mode-centric scheduling of updates



(b) Block-centric scheduling of updates

Fig. 6. (a) Mode-centric vs. (b) block-centric scheduling of updates

---

**Algorithm 2** The outline of the fine-grained decomposition algorithm, used by 2PCP, which schedules update rules in a block-centric manner

**Input**: original tensor $\boldsymbol{\mathcal{X}}$, partitioning pattern $\mathcal{K}$, and decomposition rank, $F$

**Output**: CP tensor decomposition $\mathring{\boldsymbol{\mathcal{X}}}$

1)    for all $\vec{k} \in \mathcal{K}$
       •  decompose $\boldsymbol{\mathcal{X}}_{\vec{k}}$ into $\boldsymbol{U}_{\vec{k}}^{(1)}, \boldsymbol{U}_{\vec{k}}^{(2)}, \ldots, \boldsymbol{U}_{\vec{k}}^{(N)}$

2)    repeat for each $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$
      a)   for each mode $i = 1$ to $N$
         i)   update $\boldsymbol{A}_{(k_i)}^{(i)}$ using $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$, for each block $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$; more specifically,
             •  compute $\boldsymbol{T}_{(k_i)}^{(i)}$, which involves the use of $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ (i.e. the mode-$i$ factors of $\boldsymbol{\mathcal{X}}_{[*,\ldots,*,k_i,*,\ldots,*]}$)
             •  revise $\boldsymbol{P}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ using $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ and $\boldsymbol{A}_{(k_i)}^{(i)}$
             •  compute $\boldsymbol{S}_{(k_i)}^{(i)}$ using the above
             •  update $\boldsymbol{A}_{(k_i)}^{(i)}$ using the above
         ii)  for all $\vec{l} = [*, \ldots, *, k_i, *, \ldots, *] \in \mathcal{K}$
             •  update $\boldsymbol{P}_{\vec{l}}$ and $\boldsymbol{Q}_{\vec{l}}$ using
                ◦  $\boldsymbol{U}_{\vec{l}}^{(i)}$ and $\boldsymbol{A}_{(k_i)}^{(i)}$
    until stopping condition
3)    Return $\mathring{\boldsymbol{\mathcal{X}}}$

---

### A. Block-centric Scheduling of the Update Rules for Iterative Refinement

The key observation that forms the basis of this alternative iterative refinement process is visualized in Figure 5: here, we see that for any block $\boldsymbol{\mathcal{X}}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}$, its factors $\boldsymbol{U}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}^{(1)}$ through $\boldsymbol{U}_{[k_1,\ldots,*,k_i,*,\ldots,k_N]}^{(N)}$ can be used for maintaining $N$ sub-factors of $\boldsymbol{\mathcal{X}}$, one along each of the $N$ modes. Therefore an alternative way to implement the iterative refinement process is to schedule the update rules in a *block-centeric manner* as opposed to the *mode-centeric* manner of the conventional scheme.

The outline of the proposed block-centric iterative refinement process is visualized in Algorithm 2. While the core update-rule is identical to that of the conventional, mode-centric decomposition process [22] (detailed in Algorithm 1) and while the two algorithms have the *same time and space complexities*, these two algorithms differ significantly in the

ways the update-rules are scheduled:

- as visualized in Figure 6(a), the mode-centric Algorithm 1 considers each mode one at a time, and for each mode it schedules the update rule for all the partitions of that mode; whereas

- as visualized in Figure 6 (b), the block-centric Algorithm 2, considers the individual block positions one at a time, and for each block index, $\vec{k}$, it schedules update rules for all $N$ modes together.

One way to see the key difference between these two algorithms is to consider their outer repeat-loops: as also visualized in Figure 6, in the mode-centric Algorithm 1, for each cycle of the repeat rule, each sub-factor, $\boldsymbol{A}_{(k_i)}^{(i)}$, along each mode, $i$, is updated exactly once. In the block-centric Algorithm 2, however, when all the block indexes, $\vec{k} \in \mathcal{K}$, are considered once, the sub-factor $\boldsymbol{A}_{(k_i)}^{(i)}$ along mode $i$ is updated once for each block along the partition $k_i$; i.e., $\boldsymbol{A}_{(k_i)}^{(i)}$ has been updated $\prod_{j \neq i} K_j$ times.

*Definition 1 (Block-Centric Update Schedule):* Let us consider an $N$-mode tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\boldsymbol{\mathcal{X}} = \{\boldsymbol{\mathcal{X}}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes. An update schedule, $S = \langle u_1, u_2, \ldots \rangle$, is a sequence, such that each $u_j$ belongs to the set, $\mathcal{K}$, of block positions.

In other words, a block-centric update schedule drives the order in which Algorithm 2 applies its updates through the order in which the blocks of the tensor are considered. In this paper, we consider tensor-filling, cyclic update schedules.

*Definition 2 (Tensor-Filling Schedules):* Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ be an $N$-mode tensor partitioned into a grid of blocks and $\mathcal{K}$ be the indexes of the resulting blocks. A schedule, $S$, is said to be tensor-filling if $S$ is of the form $C : C : C : \ldots : C'$ (i.e., $S$ can be thought as a repeated concatenation of a cycle sequence, $C$), such that

- the length of the cycle sequence $C$ is equal to $|\mathcal{K}|$,
- there exists a one-to-one mapping between $u_j \in C$, and $\vec{k}_i \in \mathcal{K}$,
- the last (potentially partial) sequence, $C'$, is a prefix of the cycle sequence $C$.

Intuitively, a tensor-filling schedule consists of cycle sequences, each traversing all the blocks of the given tensor – possibly except the last cycle, which may be partial if the termination condition is satisfied before the cycle sequence is completed. A tensor-filling cycle sequence would ensure that all sub-factors of $\mathcal{X}$ are updated using all the data available from the decompositions of its blocks.

## B. Virtual Iterations

In the mode-centric Algorithm 1, the stopping condition is checked once for each iteration of the outer repeat-loop; in other words, each sub-factor, $\boldsymbol{A}_{(k_i)}^{(i)}$, along each mode, $i$, is updated once. On the other hand, the outer-cycle of the block-centered Algorithm 2 is not necessarily aligned with individual iterations of the outer repeat-loop Algorithm 1. In fact, as visualized in Figure 6, each cycle of Algorithm 2 potentially involves many more updates than a single iteration of Algorithm 1. This naturally raises the question of when to check for the termination condition. While the termination condition can be checked at the end of one full cycle, this might result in redundant updates if the termination condition is reached early in a cycle.

Therefore, instead of using cycle boundaries as positions for termination condition evaluation, we introduce *virtual iterations*, which are equal in length to the length of the iterations of the mode-centric update process.

*Definition 3 (Virtual Iteration):* Given an $N$-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, partitioned into a set (or grid) of sub-tensors $\mathfrak{X} = \{\mathcal{X}_{\vec{k}} \mid \vec{k} \in \mathcal{K}\}$ where $\mathcal{K}$ is the set of sub-tensor indexes, partitioning each mode $i$ into $K_i$ equal partitions, the length of each virtual iteration is

$$length\_virtual\_iteration(\mathcal{K}) = \sum_{i=1}^{N} K_i$$

updates of the sub-factors of $\mathfrak{X}$.

Given a tensor-filling, cyclic update schedule $S$ with cycle $C$, the update schedule $C$ is split into $\frac{\prod_{i=1}^{N} K_i}{\sum_{i=1}^{N} K_i}$ virtual iterations, each of length $length\_virtual\_iteration(\mathcal{K})$.

As visualized in Figure 7, we check for termination once for each virtual iteration.

## VI. I/O REDUCING UPDATE SCHEDULES

Intuitively, when the available buffer is not sufficient to hold the entire data needed to support the decomposition process, the efficiency of the decomposition will necessarily depend on the effectiveness of the utilization of the buffer. In this section, we consider alternative block-centric update scheduling techniques, which set the order in which blocks are considered in a way to boost data reuse and reduce I/O needed to obtain tensor decomposition.

In Section IV-A, we had formalized the amount of data needed for each iteration of the mode-centric iterative improvement algorithm. We now formalize the amount of data needed for each step of the block-centric update process.

*Definition 4 (Unit of Data Access):* As we see in Algorithm 2, for each $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$, we need to bring
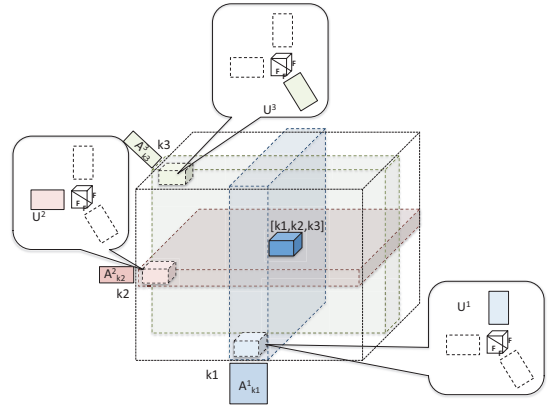


Fig. 8. For a given block position $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$, we need to bring into the memory, for each mode $i = 1$ to $N$, the data unit $data(\vec{k}, i)$ consisting of the sub-factor, $\boldsymbol{A}_{(k_i)}^{(i)}$ and $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$.

into the memory, for each mode $i = 1$ to $N$, (a) the sub-factor, $\boldsymbol{A}_{(k_i)}^{(i)}$, of the corresponding mode partition, and (b) the $i^{th}$ mode factors of all blocks corresponding to the mode partition $k_i$; i.e., $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$. Therefore, for a given a block position, $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$,

$$data(\vec{k}, i) = \left\{ \boldsymbol{A}_{(k_i)}^{(i)}; \ \boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)} \right\}, i \in \{1 \ldots N\}$$

are the $N$ *units of data* needed for implementing the update corresponding to this block position.

This is visualized in Figure 8. Therefore the data in the buffer can be organized in terms of mode-partition pairs,

$$\langle i, k_i \rangle = \left\{ \boldsymbol{A}_{(k_i)}^{(i)}; \ \boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)} \right\},$$

of size (assuming 8-byte double precision representation)

$$\left( \underbrace{\left( \frac{I_i}{K_i} \times F \right)}_{\boldsymbol{A}_{(k_i)}^{(i)}} + \underbrace{\left( \left( \prod_{j \neq i} K_j \right) \times \left( \frac{I_i}{K_i} \times F \right) \right)}_{\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}} \right) \times 8 \quad \text{bytes}$$

as also discussed in Section IV-A. Once brought into memory, such a pair can be used for implementing the factor matrix revision updates at any block position $\vec{k}$ for which the $i^{th}$ mode partition is equal to $k_i$.

## A. Re-Use Promoting Schedules

As seen above, the update process corresponding to two distinct block positions, $\vec{k} = [k_1, \ldots, k_N] \in \mathcal{K}$ and $\vec{l} = [l_1, \ldots, l_N] \in \mathcal{K}$, can share the same $\boldsymbol{U}_{[*,\ldots,*,k_i,*,\ldots,*]}^{(i)}$ matrices, if $k_i = l_i$ for some mode $i$, $1 \leq i \leq N$ (i.e., they are along the same mode partition on mode $i$). The larger the number of common mode partitions between $\vec{k} = [k_1, \ldots, k_N]$ and $\vec{l} = [l_1, \ldots, l_N]$, the larger will be the sharing of $\boldsymbol{U}$ matrices. This leads us to our primary desideratum.
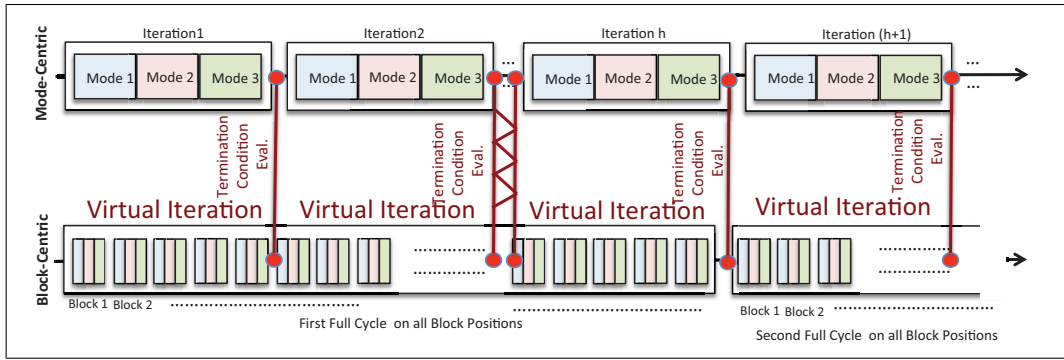
Fig. 7. Virtual iterations are equal in length to the length of the iterations of the mode-centric update process and the block-centric process checks for termination once for each virtual iteration.

*Desideratum 1 (Reuse-Promoting Schedules):* The cycle sequence, $C = \langle u_1, u_2, \ldots, u_m \rangle$ of the update schedule $S$ should be such that, the closer $u_a$ and $u_b$ are to each other (i.e., the smaller $|a - b|$ is), the larger the intersection between the corresponding block partitions, $\vec{k_a}, \vec{k_b} \in \mathcal{K}$. ◇

As we experimentally validate in Section VIII, the more reuse-promoting a schedule is, the lower is the number of accesses to the disk. We next consider alternative update schedules, with different traversal patterns of the tensor blocks.

### B. Fiber-Order Update Schedules

The first, straightforward, alternative is to traverse the tensor blocks one fiber at a time as shown in Figure 9(a). This strategy is very simple to implement in the form of a set of nested loops, with one loop for each mode.

Fiber-order update schedules support significant amount of data re-use. To see why, consider two consecutive block positions $\vec{k_h} = [k_{h,1}, \ldots, k_{h,N}]$ and $\vec{k}_{h+1} = [k_{h+1,1}, \ldots, k_{h+1,N}]$, visited as the schedule traverses along a single fiber. It is easy to see that, in most cases, the only difference between these two positions will be in their $N^{th}$ position; i.e., $\forall_{i<N} (k_{h,i} = k_{h+1,i})$ and $(k_{h+1,i} = k_{h,i} + 1)$. Consequently, $\forall_{i<N} data(\vec{k_h}, i) = data(\vec{k}_{h+1}, i)$. Therefore (assuming that $N$ data units fit into memory), once all the data needed for the block position, $\vec{k_h}$, are brought into memory, the only new data unit that may need to be fetched from the disk for the block position, $\vec{k}_{h+1}$, is $data(\vec{k}_{h+1}, N)$.

### C. Fractal-based Update Schedules

We, however, note that we can have even better data reuse by using fractal-structured block traversals, instead of relying on this simple fiber-order scheduling strategy. A fractal curve, thus, is a curve that looks similar when one zooms-in or zooms-out in the space that contains it. Fractals that are space-filling, such as $Z$-order curve [20] and Hilbert curve [12] are known to show good clustering properties in the sense that these curves tend to completely traverse a *neighborhood* of the space before moving to another neighborhood.

Our intuition is that, if two block locations $\vec{k_a} = [k_{a,1}, \ldots, k_{a,N}]$ and $\vec{k_b} = [k_{b,1}, \ldots, k_{b,N}]$, share significant amount of data, then traversals based on space-filling fractal curves are likely to visit them close to each other. Based on



(a) fiber-order schedule



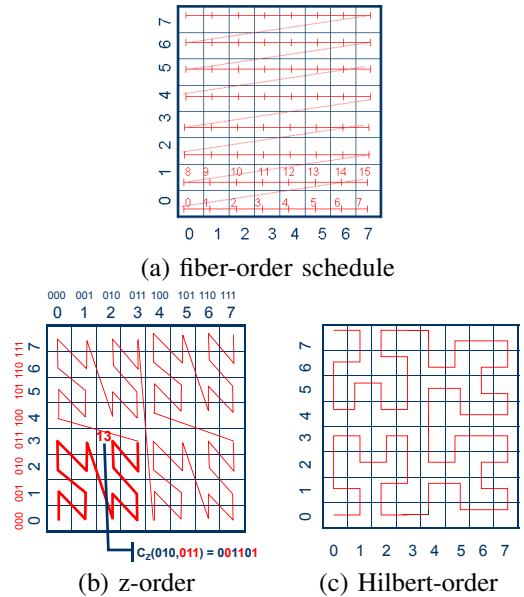(b) z-order



(c) Hilbert-order

Fig. 9. Alternative update schedules for the blocks of a 2-mode tensor (the numbers along a given mode denote the block indexes along that mode)

this intuition, we propose two update scheduling techniques, based on *Z-order* and *Hilbert-order traversal* of tensor blocks.

*1) Z-Order Update Schedules:* Z-order (or Morton-order) curve [20] is a fractal-based curve that fills the space more effectively than the fiber-order traversal described above. In particular, as mentioned above its fractal nature ensures that it clusters nearby block positions (i.e., nearby block positions are visited close to each other during traversal).

Let us consider an $N$-mode tensor where each mode $i$ is partitioned into $2^m$ partitions for some $m > 0$ and let $\vec{k} = [k_1, k_2, \ldots, k_N]$ be a block position. Then, the Z-value corresponding to $\vec{k}$ is an integer, $zvalue(\vec{k})$, defined as follows: $\forall_{1 \leq i \leq N} \forall_{1 \leq j \leq m}$

$$zvalue(\vec{k}).\texttt{base2}((m-j)N + i) = k_i.\texttt{base2}(j),$$

where, given an integer $a$ ($0 \leq a \leq 2^m - 1$), $a.\texttt{base2}(j) \in \{0, 1\}$ denotes the value of the $j^{th}$ least significant bit of $a$.

Figure 9(b) provides an example. In this example, the block position $[2, 3]$, has the corresponding $Z$-order value, $001101_2$ ($= 13_{10}$), which can be obtained by shuffling the bits of

the inputs, $010_2$ ($= 2_{10}$) and $011_2$ ($= 3_{10}$) as specified by the above formula. This example also shows that the Z-order traversal of the space is self-similar (i.e., composed of "Z"s at multiple scales) and clustered: block-positions that are closer to each other on the grid tend to be visited closer to each other also in traversal sequence. We argue that this property of the Z-order traversal should help provide a more re-use promoting schedule than the fiber-order traversal.

*2) Hilbert-Order Update Schedules:* A second look at Figure 9(b), however, points to a potential weakness of the Z-order traversal: the Z-order traversal of the blocks contains a few relatively large jumps and, at these points, the Z-order based schedule may result in a large amount of data to be brought into the memory from the disk. Therefore, the Hilbert (or Peano-Hilbert) curve [12], which tends to have smaller jumps, may provide a higher degree of reuse-promotion. Figure 9(c) shows a sample Hilbert traversal of blocks assuming a 2 mode tensor. As we see here, Hilbert traversal relies on "U" shaped curve-segments (as opposed to the "Z" shaped curve-segments of the Z-order traversal) and this helps better preserve the adjacency property (i.e., avoiding discontinuity - which would require undesirable jumps).

As we experimentally verify in Section VIII, Hilbert-order based traversal indeed provides lower I/O costs than Z-order based scheduling of the updates. However, one difficulty with the Hilbert-order traversal is that, unlike the Z-order traversal, there does not exist an efficient way to map from the block positions to the positions on the Hilbert curve (and vice versa). Existing algorithms are not practical for tensors with large numbers (10s or 100s) of modes as they may require large amounts of memory. Therefore, in those cases, $Z$-order traversal, which (as described above) has very efficient mapping implementations, may be preferred over Hilbert-order traversals of block positions.

## VII. Update Schedule Aware Buffer Replacement

So far, we focused on the problem of selecting an update schedule for tensor decomposition refinement such that the total amount of data that need to be brought from disk to the buffer is minimized. In this section, we argue that the I/O needed to perform the iterative refinement can be further reduced by using buffer replacement policies that complement the traversal order, driving the update scheduling process. In particular, we argue that the common *least-recently used (LRU)* buffer replacement strategy[4], which relies on the *temporal-locality* principle (i.e., assumes that data brought to the memory recently is likely to be used also in the near future) and drops the data which have been used furthest in the past, is not likely to be effective.

### A. Fiber-Order Schedules and MRU

Let us first consider the fiber-order strategy described in Section VI-B and visualized in Figure 9(a). It is easy to see that data brought to the memory for a given block of a fiber will not be accessed again during the traversal of that fiber. This implies that temporal locality principle does not hold during fiber-order traversal. In contrast, due to the looping characteristic of the

---

[4]For example, SciDB[1] array database underlying TensorDB [15], [16] implements LRU-based buffer management.
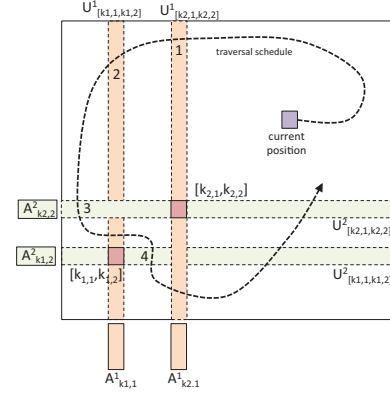


Fig. 10. Forward-looking, schedule-aware buffer replacement: Let us assume that the buffer currently contains the 4 shown data units, 2 for each of $[k_{1,1}, k_{1,2}]$ and $[k_{2,1}, k_{2,2}]$. Since, according to the current traversal plan, $data([k_{1,1}, k_{1,2}], 2) = \{\boldsymbol{A}^{(2)}_{(k_{1,2})}; \ \boldsymbol{U}^{(2)}_{[k_{1,1}, k_{1,2}]}\}$ is the last data unit to be needed, it will be the one selected for replacement

traversal, a *temporal-alocality* principle (which states that data brought to the memory for a block is <u>not</u> likely to be used in the near future) holds and this implies that a *most-recently used (MRU)* buffer replacement strategy may be more suitable for fiber-order schedules.

### B. Forward-Looking Buffer Replacement

More importantly, though, the definition of the unit of data access (Definition 4 in Section VI), along with the proposed update scheduling techniques, enable more precise *forward-looking, and traversal order aware, predictive buffer replacement strategies* as opposed to the *backward-looking* strategies, such as LRU and MRU. In particular, as we have seen in the previous section, the data in the buffer are organized in the form of mode-partition pairs, $\langle i, k_i \rangle = \left\{ \boldsymbol{A}^{(i)}_{(k_i)}; \ \boldsymbol{U}^{(i)}_{[*, \ldots, *, k_i, *, \ldots, *]} \right\}$. Once brought into memory, a $\langle i, k_i \rangle$ pair is used for updates at any block position $\vec{k}$ for which the $i^{th}$ mode partition is equal to $k_i$. Consequently, given the current position in an update schedule, if we can compute for each mode-partition pair $\langle i, k_i \rangle$ in the buffer, how far in the future the traversal will cross that mode-partition pair, then we can select for replacement the pair that will be crossed furthest in the future (Figure 10).

While such forward-looking replacement policies are difficult to implement when the data accesses are irregular and unpredictable, thanks to the regular natures of fiber-, Z-, and Hilbert-order traversals, it is possible to compute in advance precisely how far in the future (i.e., how *urgently*) a given data unit that is brought into the buffer will be needed again. This enables us to maintain the data units in an order of *urgency* and, if needed, replace the least urgent data unit. As we see in the next section, this forward-looking, traversal-order aware replacement policy significantly reduces the I/O cost of the decomposition process.

## VIII. Experimental Evaluation

In this section, we report experiments that aim to assess the effectiveness of the proposed disk and buffer sensitive update scheduling and buffer management techniques underlying

| Tensor size | 2PCP (sec.) | HaTen2 (sec.) |
|---|---|---|
| $500 \times 500 \times 500$ (0.025B non-zeros) | 92.9 | 2380.2 |
| $1000 \times 1000 \times 1000$ (0.2B non-zeros) | 441.5 | 11764.9 |
| $1500 \times 1500 \times 1500$ (0.7B non-zeros) | 1513.9 | FAILS |

TABLE I. COMPARISON OF EXECUTION TIMES ON BILLION-SCALE DENSE TENSORS (DENSITY 0.2; TARGET RANK 10; RESULTS REPORTED HERE USE A $2 \times 2 \times 2$ PARTITIONING STRATEGY FOR 2PCP; DUE TO THE LARGE EXECUTION TIME OF HATEN2, WE ONLY REPORT EXECUTION TIME FOR 1 ITERATION)
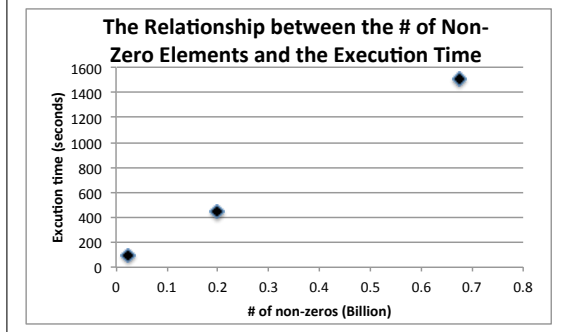


Fig. 11. 2PCP scales well as the tensor size grows (data from Table I)

2PCP. In particular, we aim to observe and report the amount of I/O (i.e., data swaps) necessitated by different update schedules under different partitioning strategies, memory availabilities, and buffer replacement strategies.

### A. Experiments with Strong Configuration

In these experiments, we considered billion-scale (with $\sim 1$ billion non-zero entries) dense tensors and compared the execution time performance of 2PCP to that of another billion-scale tensor decomposition platform, HaTen2 [13]. Note that unlike 2PCP designed for scientific applications, HaTen2 is designed for handling sparse tensors, commonly found in social media applications.

**Hardware.** For these experiments, we used EC2 platform with R3.xlarge configurations: we deployed 2PCP and HaTen2 each on 8 Intel Xeon E5-2670 v2 (Ivy Bridge) Processors (4 CPUs, each with 30.5GB and 80 GB SSD storage).

**Software.** Distributed version of 2PCP was implemented in Java 7, over Hadoop 0.20.2. The binary code for HaTen2 was obtained from http://datalab.snu.ac.kr/haten2.

In Table I, we compare the execution time performance of 2PCP to that of HaTen2 [13] for dense tensors of different sizes. As we mentioned earlier, HaTen2 is designed for handling sparse tensors, commonly found in social media applications, whereas 2PCP (motivated by scientific and engineering applications) does not make this assumption. The results reported in Table I confirm this: as we see in this table and Figure 11, while 2PCP scales well as the tensor size grows, HaTen2 requires significantly more time and memory and soon fails to run with the available resources.

It is important to note that this execution time gain does not come with any loss in accuracy. In fact, 2PCP provides significantly higher accuracy than that of HaTen2. For example, for the case with $0.025B$ non-zeros, the fit measure (described in Section III-B) for 2PCP is 0.077 whereas HaTen2's fit for the same configuration is only 0.0011.

| # Part. | Phase I<br>BD (per block) | Phase II | | Total | |
|---|---|---|---|---|---|
| | | LRU | FOR | LRU | FOR |
| Naive CP | >12 hours | N/A | N/A | N/A | N/A |
| $2 \times 2 \times 2$ | 79.1 | 10.6 | 9.6 | 89.7 | 88.7 |
| $4 \times 4 \times 4$ | 9.8 | 64.3 | 54.5 | 74.1 | **64.4** |

TABLE II. EXECUTION TIMES (IN MINUTES)

| Parameter | Alternative values |
|---|---|
| # partitions | $2 \times 2 \times 2$; $4 \times 4 \times 4$; $8 \times 8 \times 8$ |
| Buffer size (portion of the total space requirement) | 1/3; 1/2; 2/3 |
| # (virtual) iterations | 100; 200 |
| Schedules | Mode-centric (MC); Fiber-order (FO); Z-order (ZO); Hilbert-order (HO) |
| Replacement | LRU; MRU; Forward (FOR) |

TABLE III. PARAMETER SETTINGS (UNLESS OTHERWISE SPECIFIED)

### B. Experiments with Weak Configuration

In addition to the configuration considered above, we also ran experiments with a weaker configuration, consisting of quad-core Intel(R) Core(TM)i5-2400 CPU @ 3.10GHz machines with 8.00GB RAM. Since in this configuration the main-memory is much smaller, 2PCP is implemented on top TensorDB (obtained from https://github.com/mkim48/TensorDB and installed on SciDB 12.12 [1], using Python and C++) to enable out-of-core CP-ALS computations in Phase 1.

In Table II, we compare Naive CP-ALS against 2PCP with LRU and the forward-looking (FOR) strategies (both under the proposed Z-order update scheduling scheme) for different partition scenarios, for a $1000 \times 1000 \times 1000$ of high density (0.49). The target rank was set to 100. Here, the execution time for the first phase includes the time for obtaining and decomposing each block. The second phase was ran until convergence. The table also includes times for the conventional CP-decomposition (i.e., default TensorDB with no partitioning [15], [16]).

The first thing to note in this table is that, compared to conventional CP tensor decomposition (without partitioning), the fine-grained decomposition strategies that operate in a block-centric manner, significantly improve the execution time of tensor decomposition process. Secondly, we see that, as expected, the forward-looking buffer replacement (FOR) outperforms the LRU buffer replacement strategy. The fastest execution time was obtained using the $4 \times 4 \times 4$ strategy, where the forward strategy (FOR) completed in 64.4 minutes, against 74.1 minutes for the LRU strategy, a $\sim 15\%$ gain. These results are especially significant when considering that (a) this tensor cannot be decomposed using a fully in-memory Matlab based strategy and (b) a naive secondary-storage supported CP tensor decomposition using TensorDB runs more than 12 hours for the same configuration.

### C. Parameter Analysis (Stand-Alone Configuration)

In the next set of experiments we study the impact of various parameters on the data swap and accuracy performance of 2PCP. These experiments are running on a stand alone version of 2PCP developed to support system-independent evaluation of the algorithms underlying 2PCP: in order to count data swaps precisely, this version is implemented and run using Matlab 7.11.0 (2010b) and Tensor Toolbox Version 2.5 [4].

(a) Buffer size is $\frac{1}{3}$ of total data req.    (b) Buffer size is $\frac{1}{2}$ of total data req.    (c) Buffer size is $\frac{2}{3}$ of total data req.
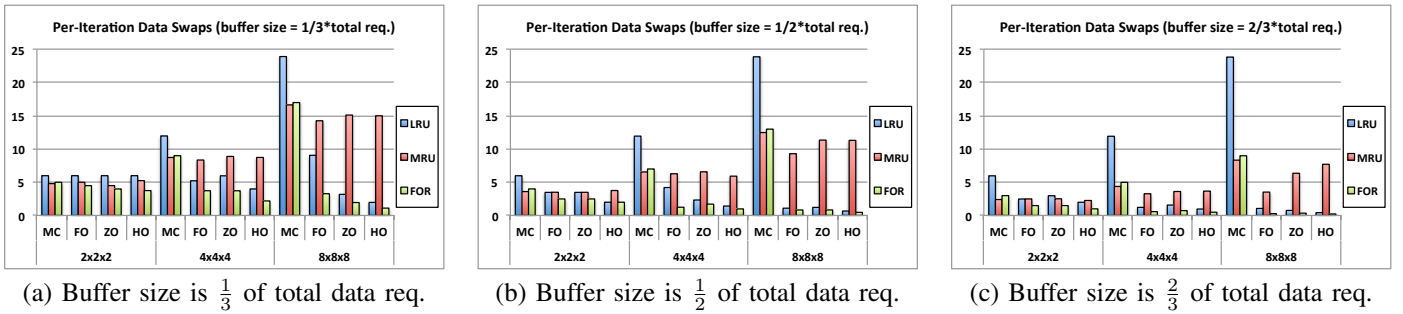
Fig. 12. Per-(virtual)iteration number of data swaps for different configurations (since the per-iteration number of swaps is not a function of the data, but the number of partitions and the *size of the buffer relative to the total space requirement*, we have the same result for all data sets)

**Evaluation Criteria - Data Swaps.** Since (a) the wall-clock execution times depend on the particular hardware/software setting (including the disk page read/write times and data compression/decompression costs – based on whether the data is compressed on the disk) and (b) since the block-based decomposition process is I/O-bound[5] we observe and report the amount of I/O (i.e., data swaps) between the disk and memory buffer for different scenarios, as is common in the buffer/cache management literature.

**Evaluation Criteria - Accuracy.** We use the measure reported in Section III-B to assess decomposition accuracy. Each experiment is ran 10 times and we report median results.

**Data.** We used four real datasets with different characteristics: *Epinions* [32], *Ciao* [32], *Enron* [26], and Face [2]. The first two of these are comparable in terms of their sizes and semantics: they are represented in the form of $170 \times 1000 \times 18$ (density $2.4 \times 10^{-4}$) and $167 \times 967 \times 18$ (density $2.2 \times 10^{-4}$) tensors, respectively, and both have the schema $\langle user, item, category \rangle$. The *Enron* email data set, however, is larger ($5632 \times 184 \times 184$, density $1.8 \times 10^{-4}$) and has a different schema, $\langle time, from, to \rangle$. The *Face* data set, a benchmark for research of face recognition, is a **dense** $480 \times 640 \times 100$ tensor with schema $\langle x\text{-}coord, y\text{-}coord, image \rangle$ and density 1.0. For these experiments, we set the stopping condition to an accuracy improvement of less than $10^{-2}$ per iteration; but, we also set a maximum number of (virtual) iterations to help observe how quickly iterative improvement converges under different strategies. In these experiments, we set the target decomposition rank to 100.

**Parameter Settings.** The number of data swaps necessary for iterative improvement is not a function of the absolute data size, but the number of partitions and the *size of the buffer relative to the total space requirement* (see Section IV-A). Therefore, as we report in Table III, in these experiments, we primarily vary the number of partitions of the tensors and the size of the memory buffer, relative to the total space requirement for the decomposition process.

*1) Amount of I/O (Data Swaps):* In Figure 12, we see per-(virtual)iteration data swaps for different scheduling and replacement algorithms, and different buffer sizes. Since the number of per-iteration swaps is not a function of the data,

---

[5]We observed that, on the average, swapping a block takes $\sim 3$ times more than the time needed to perform the in-memory operations on the block.

but the number of partitions and the *size of the buffer relative to the total space requirement*, we have the same result for all data sets. To see the impact of scheduling on convergence, the scheduling process was run without any bound on iterations. As we see in this figure 12, for all configurations the conventional mode-centric (MC) update plans result in the highest amount of I/O. In contrast, the proposed block-centric schedules require significantly lesser I/O, especially when combined with the forward-looking, schedule aware buffer replacement strategies. The worst strategy is the mode-centric (MC) schedules with LRU buffer replacement, with up to $\sim 24$ swaps per iteration, for $8 \times 8 \times 8$ partitions, independent of the buffer availability; while MRU based replacement brings the number of swaps down, MC is overall the worst strategy. In contrast, block-centric Hilbert-order schedules (HO) with forward-looking (FOR) buffer replacement has as low as $\sim 1.1$ swaps per iteration for $8 \times 8 \times 8$ partitions with $\frac{1}{3}$ buffer availability and $\sim 0.22$ swaps per iteration for the same partition with $\frac{2}{3}$ buffer availability.

To give context, let us consider a $100K \times 100K \times 100K$ tensor partitioned into $8 \times 8 \times 8$ blocks. Let us also assume that our target rank is 100. The best case for MC is on the average $\sim 8.32$ swaps per iteration with MRU, corresponding to (assuming *double-precision* number representation)

$$8.32 \times \left( \left( \frac{10^5}{8} \times 100 \right) + \left( (8 \times 8) \times \left( \frac{10^5}{8} \times 100 \right) \right) \times 8 \right)$$

$\simeq 6GB$ data exchange per iteration (Section VI). In contrast, for the same configuration, Hilbert-order (HO) schedule with forward-looking replacement (FOR) requires only $\sim 0.22$ swaps per iteration, corresponding to only $\sim 160MB$ data exchange per iteration.

*2) Accuracy Results:* In Figure 13, we see the accuracy results for different data sets, scheduling algorithms, and partition configurations The charts in the figure plot the *relative accuracy difference* between the block-centric algorithms (fiber-order, FO, Z-order, ZO, and Hilbert-order, HO) and conventional mode-centric scheduling (MO). Positive relative difference indicates cases where buffer-centric approach outperforms mode-centric approach.

As we see here, except for a few instances (specifically Enron data set, $2 \times 2 \times 2$ partitions), the accuracies of the block-centric algorithms (especially Hilbert-order, HO) do match or exceed the accuracies of the mode-centric algorithm. As expected, the variability is higher for the sparse data sets as the accuracy of the block-based iterative improvement strategy
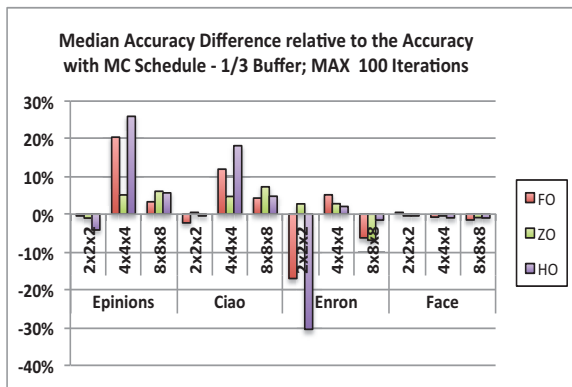
(a) Max. 100 iterations; buffer size is $\frac{1}{3}$ of total data req.
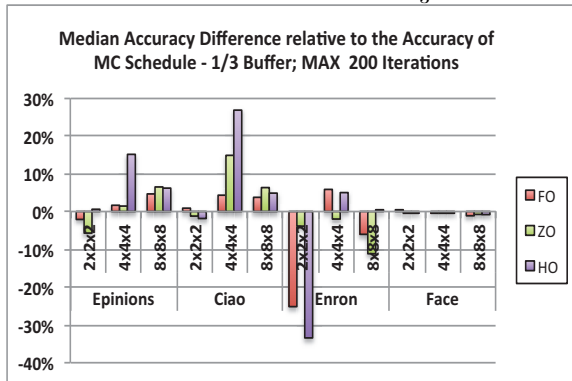


(b) Max. 200 iterations; buffer size is $\frac{1}{3}$ of total data req.

Fig. 13. Relative accuracy difference: positive values indicate cases where buffer-centric approach outperforms mode-centric approach.

depends highly on the densities of the blocks and, on sparse data sets, densities of the blocks can vary significantly. For the dense Face data set, accuracies for the mode- and block-centric algorithms are virtually identical.

## IX. CONCLUSIONS

In-memory implementations of tensor decomposition operations do not scale well. In this paper, we introduced 2PCP, a two-phase CP decomposition system with novel *re-use promoting data access scheduling* and *buffer replacement* mechanisms for efficient implementations of out-of-core and/ore parallel tensor decompositions. We first extended a block-based iterative improvement scheme in a way that enables fine-grained scheduling decisions for data accesses. Given this *fine-grained* block-centric scheme, we then considered alternative update scheduling techniques that maximizes the utility of the intermediary-data already in the buffer. We then proposed alternative buffer replacement policies and a forward-looking buffer replacement strategy that matches the proposed scheduling techniques to further bring the I/O costs down.

## REFERENCES

[1] http://www.scidb.org.

[2] "The extended yale face database b", http://vision.ucsd.edu/~leekc/ExtYaleDatabase/ExtYaleB.html.

[3] C. A. Andersson and R. Bro. The n-way toolbox for matlab. Chemometrics and Intelligent Laboratory Systems, 52(1):1-4, National Labs, 2000.

[4] B. W. Bader, T. G. Kolda, *et al.* MATLAB Tensor Toolbox Version 2.5, Available online, January 2012. URL: http://www.sandia.gov/~tgkolda/TensorToolbox.

[5] A. Beutel, A. Kumar, E. E. Papalexakis, P. P. Talukdar, C. Faloutsos, and E. P. Xing. Flexifact: Scalable flexible factorization of coupled tensors on Hadoop. SDM, 2014.

[6] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD*, 963–968, 2010.

[7] J. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 1970.

[8] K. S. Candan. Scalable Retrieval and Analysis of Simulation and Observation Data Sets. 7th International Conference on Similarity Search and Applications, SISAP 2014.

[9] J. H. Choi and S. V. Dfacto: Distributed factorization of tensors. Advances in Neural Information Processing Systems 27, pages 1296-1304, 2014.

[10] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *Proc. VLDB Endow.*, 2(2):1481–1492, Aug. 2009.

[11] R. A. Harshman, Foundations of the PARAFAC procedure: Model and conditions for an explanatory multi-mode factor analysis. UCLA Working Papers in Phonetics, 16:1-84, 1970.

[12] D. Hilbert. Ueber stetige abbildung einer linie auf ein flachenstuck. Mathematische Annalen, 38:459-460, 1891.

[13] I. Jeon, E. Papalexakis, U. Kang, and C. Faloutsos. HaTen2: Billion-scale tensor decompositions. ICDE'15, 1047-1058, 2015.

[14] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times algorithms and discoveries. KDD, 2012

[15] M. Kim and K.S. Candan. Efficient Static and Dynamic In-Database Tensor Decompositions on Chunk-Based Array Stores. CIKM, 2014.

[16] M. Kim and K.S. Candan. TensorDB: In-Database Tensor Manipulation with Tensor-Relational Query Plans. CIKM Demos, 2014.

[17] T. G. Kolda and B.W. Bader. The tophits model for higher-order web link analysis. Workshop on Link Analysis, Counterterrorism and Security, 2006

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

[19] X. Li, S. Huang, K.S. Candan, M.L. Sapino. Focusing Decomposition Accuracy by Personalizing Tensor Decomposition (PTD). CIKM, 2014.

[20] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical Report, Ottawa, Canada: IBM Ltd., 1966.

[21] A. H. Phan et al. CANDECOMP/PARAFAC decomposition of high-order tensors through tensor reshaping. *TSP*, 2013.

[22] A. H. Phan and A. Cichocki, PARAFAC algorithms for large-scale problems, Neurocomputing, 74(11), 2011.

[23] A. H. Phan and A. Cichocki. Block decomposition for very large-scale nonnegative tensor factorization. CAMSAP, Workshop, 2009.

[24] E. Papalexakis, C. Faloutsos, N. Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. ECML/PKDD. 2012.

[25] E. Papalexakis, C. Faloutsos, N.Sidiropoulos. Parcube: Sparse parallelizable CANDECOMP-PARAFAC tensor decompositions. TKDD 10(1): 3. 2015.

[26] C. E. Priebe, J. M. Conroy, D. J. Marchette, and Y. Park. Enron data set, 2006. http://cis.jhu.edu/ parky/Enron/enron.html

[27] K. Shin and U. Kang. Distributed methods for high-dimensional and large-scale tensor factorization. ICDM, pages 989–994, 2014.

[28] J. T. Sun, H. J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: a novel approach to personalized web search. WWW, 2005

[29] C. E. Tsourakakis, Mach: Fast randomized tensor decompositions. Arxiv preprint arXiv:0909.4969, 2009

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

[31] Q. Zhang, M. Berry, B. Lamb, and T. Samuel. A parallel nonnegative tensor factorization algorithm for mining global climate data. ICCS, 2009.

[32] http://www.public.asu.edu/~jtang20/datasetcode/ truststudy.htm