An Exploration of Optimization Algorithms for High Performance Tensor Completion

Shaden Smith*, Jongsoo Park†, George Karypis*
* Department of Computer Science and Engineering, University of Minnesota
{shaden, karypis}@cs.umn.edu
† Parallel Computing Lab, Intel Corporation
jongsoo.park@intel.com

***Abstract*—Tensor completion is a powerful tool used to estimate or recover missing values in multi-way data. It has seen great success in domains such as product recommendation and healthcare. Tensor completion is most often accomplished via low-rank sparse tensor factorization, a computationally expensive non-convex optimization problem which has only recently been studied in the context of parallel computing. In this work, we study three optimization algorithms that have been successfully applied to tensor completion: alternating least squares (ALS), stochastic gradient descent (SGD), and coordinate descent (CCD++). We explore opportunities for parallelism on shared- and distributed-memory systems and address challenges such as memory- and operation-efficiency, load balance, cache locality, and communication. Among our advancements are an SGD algorithm which combines stratification with asynchronous communication, an ALS algorithm rich in level-3 BLAS routines, and a communication-efficient CCD++ algorithm. We evaluate our optimizations on a variety of real datasets using a modern supercomputer and demonstrate speedups through 1024 cores. These improvements effectively reduce time-to-solution from hours to seconds on real-world datasets. We show that after our optimizations, ALS is advantageous on parallel systems of small-to-moderate scale, while both ALS and CCD++ will provide the lowest time-to-solution on large-scale distributed systems.**

I. Introduction

Many domains rely on *multi-way* data, which are variables that interact in three or more dimensions, or *modes*. An electronic health record is an interaction between variables such as a patient, symptoms, diagnosis, medical procedures, and outcome. Similarly, how much a customer will like a product is an interaction between the customer, product, and the context in which the purchase occurred (e.g., date of purchase or location). Analyzing multi-way data can provide valuable insights about the underlying relationships of the different variables that are involved. Utilizing these insights, a doctor would be more equipped to reach a provide a successful treatment and a retailer would be able to better recommend products that meet the customer's needs and preferences.

Tensors are a natural way of representing multi-way data. Tensors are the generalization of matrices to more than two modes. *Tensor completion* is the problem of estimating or recovering missing values of a tensor. For example, discovering phenotypes in electronic health records is improved by tensor completion due to missing and noisy data [1]. Similarly, predicting how a customer will rate a product under some context can be thought of as estimating a missing value in a tensor [2].

Multi-way data analysis follows the assumption that the data of interest follows a low-rank model that can be discovered. Tensor factorization is a technique that reduces a tensor to a low-rank representation, which can then be used by applications or domain experts. Tensor completion is often accomplished by finding a low-rank tensor factorization for the known data, and if a low-rank model exists then it can be used to predict the unknown data. A subtle, but important constraint is that the factorization must only capture the non-zero (or *observed*) entries of the tensor. The remaining entries are treated as missing values, not actual zeros as is often the case in other sparse tensor and matrix operations.

Tensor completion is challenging on modern processors for several reasons. Modern architectures have lower ratios of memory bandwidth to compute capabilities, which is detrimental to tensors which have highly unstructured access patterns and three or more indices per non-zero value. Furthermore, tensors do not have uniformly distributed non-zeros and often have a combination of long, *sparse* modes (e.g., patients or customers) and short, *dense* modes (e.g., medical procedures or temporal information). Scalability in the presence of highly varied mode lengths requires attention to load balance, degree of parallelism, and communication. This challenge is largely absent in matrix completion, as the number of rows and columns is usually large.

The high performance computing community has addressed some of these challenges in recent years, with research spanning both shared-memory [3]–[7] and distributed-memory [8]–[10] systems. However, the techniques and optimizations that underlie these methods are applied to factorizations that are not suitable for tensor completion due to the treatment of missing entries.

In this work, we explore the task of high performance tensor completion with three popular optimizations algorithms: alternating least squares (ALS), stochastic gradient descent (SGD), and coordinate descent (CCD++). We address issues on shared- and distributed-memory systems such as memory and operation-efficient algorithms, cache locality, load bal-

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

ance, and communication. Our contributions include:

1) Tensor completion algorithms which scale to thousands of cores despite the presence of highly varied mode lengths.
2) An SGD algorithm which uses a hybrid of Hogwild, stratification, and asynchronous updates to maintain convergence at scale.
3) An experimental evaluation with several real-world datasets on up to 1024 cores. Our ALS and CCD++ algorithms are $153\times$ and $21.4\times$ faster than state-of-the-art parallel methods, respectively. This effectively reduces solution time from hours to seconds.
4) Publicly available MPI+OpenMP implementations that utilize compressed tensor representations in order to improve cache locality and reduce memory consumption and the number of FLOPs performed.
5) We show that depending on the underlying parallel architecture and the characteristics of the desired solution, the best performing optimization method varies.

The rest of this paper is organized as follows. Section II introduces tensor notation and provides a brief background on tensor factorization. Section III reviews optimization methods for tensor completion and existing work on their parallelization. Section IV details our own ALS, SGD, and CCD++ algorithms. Section V includes experimental methodology and results. Finally, we provide concluding remarks in Section VI.

II. Preliminaries

A. Tensor Notation & Factorization

We denote matrices using bold capital letters ($\mathbf{A}$) and tensors using bold capital calligraphic letters ($\boldsymbol{\mathcal{R}}$). A tensor occupies three or more dimensions, or *modes*. In future discussions, we focus on three-mode tensors to reduce notational clutter. However, all of the presented algorithms are general to any number of modes. A tensor has $\text{nnz}(\boldsymbol{\mathcal{R}})$ non-zero entries and is of dimension $I \times J \times K$. The entry in position (i, j, k) is written $\boldsymbol{\mathcal{R}}(i, j, k)$. A colon in the place of an index represents all members of that mode. For example, $\mathbf{A}(:, f)$ is column f of the matrix $\mathbf{A}$. A *fiber* is the result of holding all but one index constant (e.g., $\boldsymbol{\mathcal{X}}(i, j, :)$) and is the generalization of a row or column of a matrix. A *slice* is the result of holding all but two indices constant (e.g., $\boldsymbol{\mathcal{R}}(i, :, :)$) and the result is a matrix. A common operation in tensor algebra is the *Hadamard* product, denoted $\mathbf{A} * \mathbf{B}$, which performs element-wise multiplication.

The *canonical polyadic decomposition* (CPD), also known as PARAFAC/CANDECOMP [11], is a widely-used model for tensor factorization [1], [2], [12]–[15]. The CPD is popular in domains dealing with large-scale data (e.g., machine learning) due to its computationally efficient computation and because it is unique under mild assumptions [16]. The CPD models $\boldsymbol{\mathcal{R}}$ as a rank-F matrix for each mode: $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$, and $\mathbf{C} \in \mathbb{R}^{K \times F}$. Using the CPD, a tensor $\boldsymbol{\mathcal{R}}$ can be constructed as a summation of F rank-one tensors (Figure 1). Domain experts are almost always interested in a *low-rank* CPD, with a typical value for F being 10 or 50. The CPD can also be written element-wise:

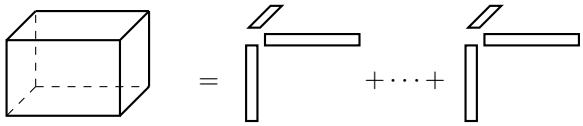


Fig. 1: The CPD as a sum of outer products.

$$\boldsymbol{\mathcal{R}}(i, j, k) = \sum_{f=1}^{F} \mathbf{A}(i, f)\mathbf{B}(j, f)\mathbf{C}(k, f). \tag{1}$$

More information on tensor factorization can be found in the survey by Kolda and Bader [17].

B. Tensor Completion with the CPD

Tensor completion under the CPD model is written as the following non-convex optimization problem:

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \quad \frac{1}{2} \sum_{\boldsymbol{\mathcal{R}}(:,:,:)} \mathcal{L}(i, j, k)^2 + \frac{\lambda}{2} \left(||\mathbf{A}||_F^2 + ||\mathbf{B}||_F^2 + ||\mathbf{C}||_F^2 \right), \tag{2}$$

where λ is a parameter for regularization and $\mathcal{L}(\cdot)$ is the loss function defined as

$$\mathcal{L}(i, j, k) = \boldsymbol{\mathcal{R}}(i, j, k) - \sum_{f=1}^{F} \mathbf{A}(i, f)\mathbf{B}(j, f)\mathbf{C}(k, f). \tag{3}$$

Note that Equation (2) is only defined over the non-zero (or *observed*) entries of $\boldsymbol{\mathcal{R}}$ and Equation (3) is derived from the element-wise formulation of the CPD in Equation (1).

III. Related Work

The non-convexity of Equation (2) has inspired a substantial amount of research on optimization algorithms that effectively minimize the objective while being operation- and memory-efficient enough to be used in practice. Three optimization algorithms have seen particular success due to their efficiency, opportunities for parallelism, and fast convergence. These methods are summarized in Table I and described below.

We follow the convention of the matrix completion community and refer to an *epoch* as the work performed to update $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ one time using the training data. We avoid the term *iteration* in order to emphasize the varying amounts of work performed and progress made.

A. Alternating Least Squares

ALS is an alternating optimization algorithm that cyclically updates one matrix factor at a time while holding all others constant. Factor updates are based on the observation that if $\mathbf{B}$ and $\mathbf{C}$ are treated as constant, solving for a row of $\mathbf{A}$ is a convex optimization problem with a least squares solution.

Illustrated in Figure 2, computing $\mathbf{A}(i, :)$ accesses all non-zeros in $\boldsymbol{\mathcal{R}}(i, :, :)$ and also the rows $\mathbf{B}(j, :)$ and $\mathbf{C}(k, :)$ for each non-zero $\boldsymbol{\mathcal{R}}(i, j, k)$. The rows of $\mathbf{B}$ and $\mathbf{C}$ are used to compute $\mathbf{H}_i$, a $|\boldsymbol{\mathcal{R}}(i, :, :)| \times F$ matrix. If the lth non-zero in $\boldsymbol{\mathcal{R}}(i, :, :)$ has

TABLE I: Summary of optimization algorithms for matrix and tensor completion.

	Matrix			Tensor			
Algorithm	**Complexity**	**Storage**	**References**	**Complexity**	**Storage**	**Traversals**	**References**
ALS	$O(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{R}}) + IF^3)$	$O(F^2)$	[18], [19]	$O(M(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{R}}) + IF^3))$	$O(F^2)$	M	[20], [21]
SGD	$O(F \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$	$O(F)$	[22]–[27]	$O(MF \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$	$O(F)$	1	[20], [28]
CCD++	$O(F \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$	$O(I)$	[29]	$O(MF \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$	$O(I)$	MF	[21], [28]

Complexity is the number of floating-point operations performed in one epoch. **Storage** is the amount of memory required to perform the factorization, excluding matrix and tensor storage. **Traversals** is the number of times the sparsity structure must be traversed in one epoch, excluding checks for convergence. F is the rank of the factorization. M is the number of modes in the tensor. I is the length of the longest mode.

coordinate (i, j, k), then $\mathbf{H}_i(l, :) = (\mathbf{B}(j, :) * \mathbf{C}(k, :))$. Given $\mathbf{H}_i$, we can compute $\mathbf{A}(i, :)$ via

$$\mathbf{A}(i,:) \leftarrow \left(\mathbf{H}_i^T \mathbf{H}_i + \lambda \mathbf{I}\right)^{-1} \mathbf{H}_i^T \operatorname{vec}(\boldsymbol{\mathcal{R}}(i,:,:)), \qquad (4)$$

where $\operatorname{vec}(\cdot)$ rearranges the non-zero entries of its argument into a dense vector. The matrix $\left(\mathbf{H}_i^T \mathbf{H}_i + \lambda \mathbf{I}\right)$ is symmetric positive-definite and so the inversion is accomplished via a Cholesky factorization and forward/backward substitutions. ALS requires $O(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$ operations to form all of the $\mathbf{H}_i$, and $O(F^3)$ operations per row for the matrix inversions. In total, $O(F^2 \operatorname{nnz}(\boldsymbol{\mathcal{R}}) + IF^3)$ operations are performed to update a factor. After computing all rows of $\mathbf{A}$, the other factors are computed in the same manner.

Parallel ALS algorithms exploit the independence of the I least squares problems and solve them in parallel. ALS was one of the first optimization algorithms applied to large-scale matrix completion [18]. Recently, a high performance ALS algorithm for matrix completion on CPUs and GPUs was developed [19]. The algorithm exploits level-3 BLAS opportunities during the construction of $\mathbf{H}_i^T \mathbf{H}_i$.

ALS was first extended to tensor completion on shared-memory systems [20]. Shin and Kang presented a distributed-memory implementation based on the MapReduce paradigm [28]. They use a *coarse-grained* tensor decomposition which partitions each mode separately, assigning to each process a set of complete $\boldsymbol{\mathcal{R}}(i, :, :)$, $\boldsymbol{\mathcal{R}}(:, j, :)$, and $\boldsymbol{\mathcal{R}}(:, :, k)$ slices. After a process updates $\mathbf{A}(i, :)$, the new values are broadcasted to all other processes. Karlsson et al. developed a distributed-memory algorithm for MPI [21]. The distributed-memory algorithm assigns non-zeros to processes without restriction, allowing for $\operatorname{nnz}(\boldsymbol{\mathcal{R}})$ parallelism. The added parallelism comes with the cost of communicating partial computations of $\mathbf{H}_i^T \mathbf{H}_i$ and $\mathbf{H}_i^T \operatorname{vec}(\boldsymbol{\mathcal{R}}(i, :, :))$ with an all-reduce, requiring $O(IF^2)$ words communicated per process.

B. Stochastic Gradient Descent

The strategy of SGD is to take many small steps per epoch, each based on the gradient at a single non-zero. At each step, SGD selects one non-zero at random and updates the factorization based on the gradient at $\boldsymbol{\mathcal{R}}(i, j, k)$. Updates are of the form

$$\begin{aligned}
\mathbf{A}(i,:) &\leftarrow \mathbf{A}(i,:) + \eta \left[\mathcal{L}(i,j,k) \left(\mathbf{B}(j,:) * \mathbf{C}(k,:)\right) - \lambda \mathbf{A}(i,:)\right], \\
\mathbf{B}(j,:) &\leftarrow \mathbf{B}(j,:) + \eta \left[\mathcal{L}(i,j,k) \left(\mathbf{A}(i,:) * \mathbf{C}(k,:)\right) - \lambda \mathbf{B}(j,:)\right], \\
\mathbf{C}(k,:) &\leftarrow \mathbf{C}(k,:) + \eta \left[\mathcal{L}(i,j,k) \left(\mathbf{A}(i,:) * \mathbf{B}(j,:)\right) - \lambda \mathbf{C}(k,:)\right],
\end{aligned}$$

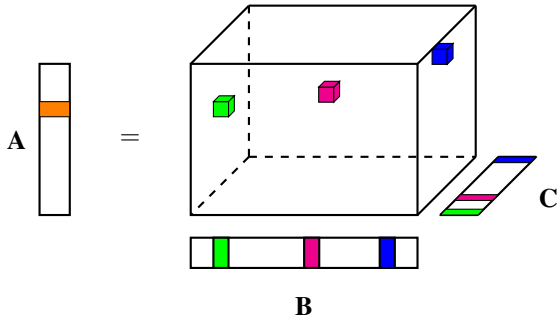


Fig. 2: Memory accesses during ALS. Computing $\mathbf{A}(i, :)$ (in orange) requires $\boldsymbol{\mathcal{R}}(i, :, :)$ and the corresponding rows of $\mathbf{B}$ and $\mathbf{C}$.

where η is a step size parameter. Each update requires $O(F)$ operations, resulting in a complexity of $O(F \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$ per epoch.

SGD is parallelized by exploiting the independence of non-zeros whose coordinates are disjoint. The standard *stratification*-based SGD method [23] partitions an M-mode tensor into a P^M grids for P processes. For example, Figure 3 shows a case with M=3 and P=3. The grid has P^{M-1} strata, each corresponding to P blocks in a diagonal line (i.e., $\{(1, t_2, ..., t_M), (2, (t_2 + 1) \bmod P, ..., (t_M + 1) \bmod P), ..., (P, (t_2 + P - 1) \bmod P, ..., (t_M + P - 1) \bmod P)\}$ for all $1 \leq t_2, ..., t_M \leq P$). Each epoch comprises processing P^{M-1} strata, which covers all the non-zeros of the tensor. Since no two non-zeros in different blocks of a given stratum share the same index in any mode, P processes can work on P blocks of a stratum in parallel.

Instead of stratification, some parallel SGD methods allow non-zeros with overlapping coordinates to be processed in parallel. Hogwild [22], a parallel algorithm for shared-memory systems, exploits the stochastic nature of SGD to have lock-free parallelism. The concept is simple: process the shuffled non-zeros in parallel without stratification or synchronization constructs. Due to the sparse nature of the input, race conditions are expected to be rare. When they do occur, the stochastic nature of the algorithm will naturally fix any errors and continue to converge. A similar idea for distributed-memory systems is *asynchronous* SGD (ASGD). All non-zeros are processed in parallel, and a few times per epoch processes combine local updates to overlapped rows via weighted averages. The communication and averaging of local

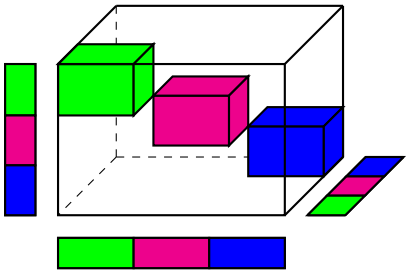


Fig. 3: Stratified SGD. Colored blocks of non-zeros can be processed in parallel without conflict.

updates are performed asynchronously [26].

C. Coordinate Descent

In contrast to ALS and SGD which update entire factor rows at a time, coordinate descent methods optimize only one variable at a time. CCD++ is a coordinate descent method originally developed for matrix factorization [29] and later extended to tensors [21], [28]. CCD++ updates columns of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in sequence, in effect optimizing the rank-one components of the factorization. Updates take the form:

$$\mathbf{A}(i,f) \leftarrow \frac{\alpha_i}{\lambda + \beta_i}, \tag{5}$$

where

$$\alpha_i = \sum_{\boldsymbol{\mathcal{R}}(i,:,:)} \mathcal{L}(i,j,k)\mathbf{B}(j,f)\mathbf{C}(k,f),$$

and

$$\beta_i = \sum_{\boldsymbol{\mathcal{R}}(i,:,:)} \left(\mathbf{B}(j,f)\mathbf{C}(k,f)\right)^2.$$

After updating all $\mathbf{A}(:,f)$, the columns $\mathbf{B}(:,f)$ and $\mathbf{C}(:,f)$ are updated similarly. An important optimization for CCD++ is to compute $\mathcal{L}(\cdot)$ only once each epoch and reuse it for each of the F columns [29]. This can be accomplished without additional storage by directly updating $\boldsymbol{\mathcal{R}}$ each iteration with the current residual. The resulting complexity is $O(F(\text{nnz}(\boldsymbol{\mathcal{R}})+I+J+K))$, which for most datasets is $O(F\,\text{nnz}(\boldsymbol{\mathcal{R}}))$, matching SGD.

CCD++ is parallelized in the same manner as ALS. All of the α_i's and β_i's for a mode are computed independently, again leading to a coarse-grained decomposition of $\boldsymbol{\mathcal{R}}$ [28], [29]. After updating a factor column in parallel, the new column factors are broadcasted to all processes. Karlsson et al. use a non-restrictive decomposition of the tensor non-zeros as in ALS [21]. Partial products are again aggregated with an all-reduce and new columns are broadcasted. Only the components of the current column must be communicated, and so in one epoch there are $2F$ messages per factor matrix, each of size $O(I)$.

IV. Shared- and Distributed-Memory Algorithms for High Performance Tensor Completion

A. Compressed Sparse Fiber

We will show in subsequent sections that the choice of data structure for representing a sparse tensor affects performance in ways such as memory bandwidth, number of FLOPs performed, and opportunities for parallelism. Our ALS, SGD, and CCD++ algorithms leverage the recently proposed *compressed sparse fiber* (CSF) data structure [5], illustrated in Figure 4. The CSF data structure recursively compresses the tensor mode-by-mode. CSF can be thought of as a generalization of the compressed sparse row data data structure for matrices.

The evaluation of $\mathcal{L}(\cdot)$ in Equation (3) can benefit from exploiting the CSF tensor representation. Consider the computation of $\mathcal{L}(\cdot)$ associated with two successive non-zeros:

$$\mathcal{L}(i,j,k) = \boldsymbol{\mathcal{R}}(i,j,k) - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f),$$

$$\mathcal{L}(i,j,k') = \boldsymbol{\mathcal{R}}(i,j,k') - \sum_{f=1}^{F} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k',f).$$

We can reuse partial results by storing the Hadamard (element-wise) product of $\mathbf{A}(i,:)$ and $\mathbf{B}(j,:)$ in a row vector $\mathbf{v}$:

$$\mathbf{v} \leftarrow \mathbf{A}(i,:) * \mathbf{B}(j,:)$$

$$\mathcal{L}(i,j,k) = \boldsymbol{\mathcal{R}}(i,j,k) - \sum_{f=1}^{F} \mathbf{v}(f)\mathbf{C}(k,f),$$

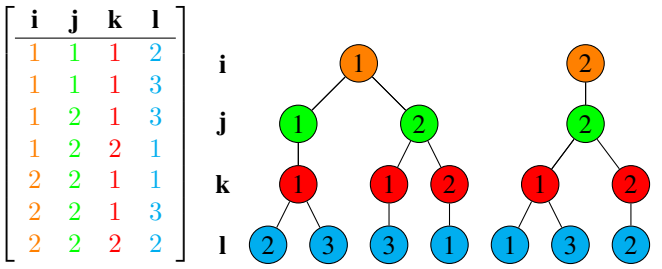
$$\mathcal{L}(i,j,k') = \boldsymbol{\mathcal{R}}(i,j,k') - \sum_{f=1}^{F} \mathbf{v}(f)\mathbf{C}(k',f).$$

This reduces the computation from $2F\,\text{nnz}(\boldsymbol{\mathcal{R}})$ multiplications to $F\,\text{nnz}(\boldsymbol{\mathcal{R}})+FP$, where P is the number of unique $\boldsymbol{\mathcal{R}}(i,j,:)$ fibers. The *matricized tensor times Khatri-Rao product* (MTTKRP) operation present in ALS uses the same technique for operation reduction [4]. As in MTTKRP, this technique can be applied recursively to tensors with more than three modes.

B. Parallel ALS

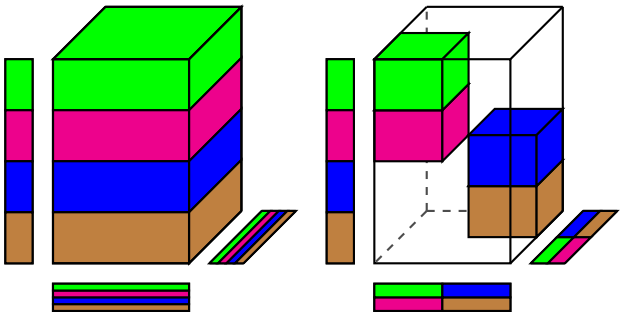
We follow the strategy of parallelizing over the rows of $\mathbf{A}$ for shared-memory parallel systems. Recall from Equation (4) that $\mathbf{H}_i$ has $|\boldsymbol{\mathcal{R}}(i,:,:)|$ rows and F columns. A major challenge when designing ALS algorithms is that if multiple rows of $\mathbf{A}$ are computed at once, the separate $\mathbf{H}_i$ matrices must somehow be represented in memory. However, the collective storage for all $\mathbf{H}_i$ requires $F\,\text{nnz}(\boldsymbol{\mathcal{R}})$ storage. Another option, and the one used by existing work [21], is to aggregate rank-1 updates. For each non-zero $\boldsymbol{\mathcal{R}}(i,j,k)$, a rank-1 update with the row vector $(\mathbf{B}(j,:) * \mathbf{C}(k,:))$ is applied to $\mathbf{H}_i^T\mathbf{H}_i$. A naive implementation must aggregate the rank-1 updates for all possible i, requiring IF^2 storage. It was observed that this storage overhead can be reduced by sorting the tensor non-zeros before processing long modes [21].

We instead store a CSF representation of $\boldsymbol{\mathcal{R}}$ for each mode. This allows us to process all of the non-zeros in $\boldsymbol{\mathcal{R}}(i,:,:)$



(a) (b)

Fig. 4: Data structures for the sparsity pattern of a four-mode tensor. (a) Coordinate format: non-zeros are represented as an uncompressed list. (b) Compressed sparse fiber (CSF) format: the sparsity structure is recursively compressed and non-zeros are stored as paths from roots to leaves.



(a) (b)

Fig. 5: Asynchronous SGD strategies for P=4 processes. (a) S=1 stratum layers. The longest mode is partitioned and other modes are updated asynchronously. (b) S=2 stratum layers. The longest mode is partitioned and S teams of P/S asynchronously update at the end of each stratum.

sequentially and thus only the $O(F^2)$ memory associated with a single row is be allocated. For multicore systems, we parallelize over the rows of $\mathbf{A}$ with a memory overhead of only $O(F^2)$ per thread. The total storage overhead with P threads is $(M-1)\,\mathrm{nnz}(\boldsymbol{\mathcal{R}})+PF^2 = O(M\,\mathrm{nnz}(\boldsymbol{\mathcal{R}}))$, which is smaller than $F\,\mathrm{nnz}(\boldsymbol{\mathcal{R}})$ and IF^2 for most problems.

The construction of $\mathbf{H}_i^T\mathbf{H}_i$ and $\mathbf{H}_i^T\,\mathrm{vec}(\boldsymbol{\mathcal{R}}(i,:,:))$ are performed together during a single pass over the sparsity structure of $\boldsymbol{\mathcal{R}}$. Interestingly, the expression $\mathbf{H}_i^T\,\mathrm{vec}(\boldsymbol{\mathcal{R}}(i,:,:))$ is equivalent to computing one transposed row of the MTTKRP operation, for which operation-efficient CSF algorithms exist [5].

Accumulating rank-1 updates into $\mathbf{H}_i^T\mathbf{H}_i$ causes $O(F^2)$ data to be accessed for each non-zero and leads to memory-bound computation. We collect the Hadamard products formed during MTTKRP into a thread-local matrix of fixed size. When that matrix fills, or all $\boldsymbol{\mathcal{R}}(i,:,:)$ are processed, the thread performs one rank-k update, where k is the number of rows in the matrix. We empirically found that a matrix of size $2048\times F$ is sufficient to see significant benefits from the BLAS-3 performance. For typical values of F, this equates to storage overheads of up to a few megabytes per thread.

We follow existing work and use a coarse-grained decomposition in the distributed-memory setting [28]. Coarse-grained decompositions impose separate 1D decompositions on each tensor mode, eliminating the need to communicate and aggregate partial computations. As a result, the worst-case communication volume is reduced from $O(IF^2)$ to $O(IF)$ per process, which is the cost of exchanging updated rows of $\mathbf{A}$. Our coarse-grained decomposition uses chains-on-chains partitioning to optimally load-balance the slices of each mode [30]. After partitioning and distributing non-zeros, we can use our shared-memory ALS kernels without modification. After a factor matrix is updated, we use an all-to-all collective communication to exchange the new rows.

C. Parallel SGD

Processing non-zeros in a totally random fashion requires $O(\mathrm{nnz}(\boldsymbol{\mathcal{R}}))$ work per epoch to shuffle non-zeros and results in random access patterns to the matrix factors. We use a coarse-grained approach and randomize only one mode of the tensor, i.e., we randomize the processing order of the trees in the CSF structure (Figure 4) but sequentially access non-zeros within each tree. The coarse approach reduces the cost of shuffling to $O(I)$ and retains the cache-friendly access pattern to the matrix factors that is provided by CSF. Our shared-memory SGD algorithm uses a Hogwild approach [22] and processes the trees in parallel without synchronization constructs. Choosing which mode to randomize requires careful consideration because SGD may not converge if updates are not sufficiently stochastic. This decision is explored in Section V-C.

In a distributed-memory setting, we refer to a group of blocks in the P^M stratification grid (Figure 3) which share a coordinate as a *stratum layer*. For example, blocks of coordinate $(i,:,:)$ are in the ith layer along the first mode. We partition the grid by first selecting the longest mode and decomposing it in a 1D fashion. Each process is assigned to a unique stratum layer in the longest mode. This privatizes the largest matrix factor, meaning there will not be any communication associated with it during the factorization.

The number of strata increases exponentially with the number of modes. Since the work per epoch is constant, the average work per stratum that can be processed in parallel decreases exponentially. Moreover, higher-mode tensors are more likely to have dense modes which can be easily exceeded by P. In this case, the amount of parallelism is limited by the shortest mode (the number of blocks in a stratum equals $\min(I_1,...,I_M)$, where I_m is the length of mth mode) to preserve the property of SGD that an update for a non-zero is immediately visible to the next non-zero.

In order to address these parallelism challenges, we extend ASGD to tensors, which allows multiple processes to update the same row of a factor matrix with their local copies. ASGD can trade-off the staleness of factor matrices for increased parallelism by adjusting the number of copies and the frequency of synchronizing the copies. Our implementation is

parameterized with the number of stratum layers, S, which determines the number of strata as S^{M-1}. We can set $S = P$ (which reduces ASGD to the usual stratified SGD) for tensors with a few modes or set $S < P$ for tensors with more modes. When $S < P$, since each stratum has only S independent blocks, P/S processes need to update the same range of factor matrices simultaneously, resulting in up to P/S copies of a factor matrix row. Specifically, we partition an M-mode tensor into a $P \times S^{M-1}$ grid and assign P mode-1 layers to each process. Then, we group every P/S processes as a *team* with total S teams. This process is shown in Figure 5.

We partition each factor matrix among P processes, aligning with the grid used for the tensor partitioning. At the beginning of a stratum, each process sends the rows of factor matrices that it owns to the other processors that need them. By our construction, a factor matrix row will be sent to one team, thus limiting the number of copies to P/S. Then, each process goes through the non-zeros of the current stratum it owns, updating the corresponding rows of factor matrices. After the update, processes send the updated rows back to their owners. Finally, processes compute weighted sums of the received updated rows, where the weights are the number of non-zeros which updated the particular row. For example, for a 3-mode tensor $\boldsymbol{\mathcal{R}}$ and a given stratum, suppose process p_1 processes 2 non-zeros in a mode-2 slice $\boldsymbol{\mathcal{R}}(:, i, :)$ and p_2 processes 1 non-zeros in the same slice. At the end of the stratum, the owner of ith row computes a weighted sum of the local copies of ith row of mode-2 factor matrix from p_1 and p_2, using 2/3 and 1/3 as their weights, respectively. Since we synchronize the factor matrices every stratum, the number of synchronizations per epoch is S^{M-1}.

ASGD allows us to alleviate the limited amount of parallelism and frequent communication, the primary challenges of SGD, especially for high-mode tensors. Still, compared to ALS and CCD++, SGD has higher communication volume, which can be analyzed as follows. For each stratum, a process receives the rows of factor matrices that correspond to the non-zeros it needs to process, which equals the sum of number of non-empty slices of each mode except for the first mode (which is completely privatized). In a worst case (and not an uncommon case for highly sparse tensors), we have only a few non-zeros per slice, leading to receiving $O(F \operatorname{nnz}(\boldsymbol{\mathcal{R}}_{p,s}))$ floating point numbers for each mode for the sub-tensor $\boldsymbol{\mathcal{R}}_{p,s}$ processed by process p at stratum s. Summing over all processes and strata results in $O(MF \operatorname{nnz}(\boldsymbol{\mathcal{R}}))$ total communication volume. It is important to receive only the required factor matrix rows corresponding to non-empty slices to be processed. Otherwise, the total communication volume will be $O\left(S^{M-2} F \sum_{m=2}^{M} I_m\right)$ as analyzed by Shin and Kang [28].

D. Parallel CCD++

As discussed in Section III-C, CCD++ algorithms usually follow a similar parallelization strategy as ALS on shared-memory systems. Following Equation (5), all α_i's and β_i's are independent subproblems and can be computed in parallel. However, unlike ALS, it is not advantageous to use separate CSF representations for each mode in order to extract coarse-grained parallelism. This is due to the added $(M-1)\operatorname{nnz}(\boldsymbol{\mathcal{R}})$ operations that would be required for updating multiple residual tensors. Therefore, we restrict ourselves to a single tensor and turn to other decomposition strategies.

We leverage the the P^M-way tiling strategy developed for parallelizing MTTKRP with a single CSF [5]. An M-dimensional grid is imposed on $\boldsymbol{\mathcal{R}}$, with each dimension having P chunks. This allows P threads to partition any mode of $\boldsymbol{\mathcal{R}}$ into P independent chunks, each consisting of P^{M-1} tiles. Each mode of $\boldsymbol{\mathcal{R}}$ can thus be updated without parallel overheads such as reductions or synchronization.

Conveniently, because threads access whole layers of tiles at a time, we do not need each of the P^M tiles to have a balanced number of non-zeros. Instead, only the layers themselves need to be balanced. We use chains-on-chains partitioning to determine the layer boundaries in each mode, resulting in load-balanced parallel execution.

In a distributed-memory setting, the communication requirements of CCD++ closely follow those of MTTKRP. A minor variation comes from CCD++ being a column-major method, and thus we must exchange partial results and updated columns F times per mode instead of exchanging full rows once per mode. Unlike ALS, exchanging partial results does not cause a prohibitive amount of communication. We can therefore choose from the recently proposed fine-grained [9] and medium-grained [10] decompositions for MTTKRP. We opt for the medium-grained decomposition, which imposes an M dimensional grid over $\boldsymbol{\mathcal{R}}$. The medium-grained decomposition varies from our tiling strategy in that there are only as many cells in the grid as processes. After distributing $\boldsymbol{\mathcal{R}}$ over a grid, our shared-memory parallelization strategy can be applied by each process independently.

E. Dense Mode Replication

ALS and CCD++ parallelize over the dimensions of $\boldsymbol{\mathcal{R}}$. Many real-world tensors have modes with skewed dimensions. For example, a tensor of health records will have significantly more unique patients than unique medical procedures or doctors. Simply parallelizing over the short, dense modes is insufficient because the number of threads can easily outnumber the slices to process. Additionally, the dense modes often have non-zeros that are not uniformly distributed, leading to further load imbalance.

The issue of dense modes was first addressed in [20] in shared-memory ALS for cases when $I < P$. Non-zeros are instead divided among threads and each thread computes a local set of $\mathbf{H}_i^T \mathbf{H}_i$ and $\mathbf{H}_i^T \operatorname{vec}(\boldsymbol{\mathcal{R}}(i, :, :))$. A parallel reduction is then used to combine all partial results before the Cholesky factorization. We adopt this solution in our own ALS implementation and parallelize directly over non-zeros when the mode is dense. We use the tiling mechanism used by CCD++ to load balance non-zeros. The tensor is still stored using CSF and thus the optimizations to achieve BLAS-3 performance can still be employed.

CCD++ uses a similar mechanism for handling dense modes. CCD++ only constructs one representation of $\mathcal{R}$ which is already tiled for parallelism. There is no advantage to tiling the dense modes because they will not be partitioned, and so we use a P^{M-d}-way tiling on a tensor with d dense modes. Each thread then forms local α_i and β_i and we aggregate them with a parallel reduction when the mode is dense.

V. Experimental Methodology and Results

A. Experimental Setup

We use the Cori supercomputer at NERSC. Each compute node has 128 GB of memory and is equipped with two sockets of 16-core Intel Xeon E5-2698 v3 that has 40 MB last-level cache. The compute nodes are interconnected via Cray Aries with Dragonfly topology. Our ALS, SGD, and CCD++ implementations are made part of the open source tensor factorization library, SPLATT [31]. We use double-precision floating-point numbers and 64-bit integers. We use the Intel compiler version 16.0.0 with the `-xCORE-AVX2` option for new instructions available in the Haswell generation of Xeon processors, Cray MPI version 7.3.1, and Intel MKL version 11.3.0 for LAPACK routines used in ALS. Compute jobs are scheduled with Slurm version 16.05.03. We run one MPI rank per socket (two ranks per node) and one OpenMP thread per core for SGD and CCD++, and one MPI rank per node for ALS. We use the *bold driver* heuristic [32] for a dynamic step size parameter in SGD with an initial value of 10^{-3}.

We follow the recommender systems community and use *root-mean-square error* (RMSE) as a measure of factorization quality. RMSE was the metric used by the Netflix Prize [33], where the first algorithm to improve the baseline RMSE by 10% was awarded one million dollars. RMSE is defined as

$$\text{RMSE} = \sqrt{\frac{\sum_{\mathcal{R}(:,:,:)} \mathcal{L}(i,j,k)^2}{\text{nnz}(\mathcal{R})}}.$$

Datasets are split into 80% *training*, 10% *validation*, and 10% *test* sets. The training set is used to compute the factorization. RMSE is computed each epoch using the validation set, and convergence is detected when the RMSE does not improve for twenty epochs. The final factorization quality is determined by the test set.

B. Datasets

Table II summarizes the tensors that we use for evaluation. The reported non-zeros and memory requirements reflect only that of the training data, because that is the portion of the computation that we focus on in this work. We work with real-world datasets coming from a variety of domains. Netflix and Yahoo! are *(user, item, month)* product rating tuples with values ranging 1-5 and 1-100, respectively. Amazon is formed from *(user, item, word)* tuples taken from product reviews. Outpatient is a six-mode tensor of *(patient, institution, physician, diagnoses, procedure, day)* tuples formed from outpatient Medicare claims. We selected non-zeros from the original data in order to have three-, four-, five-, and six-mode versions of this dataset with the same number of non-zeros. The Amazon and Outpatient datasets can be found online[1].

TABLE II: Summary of training datasets.

Dataset	NNZ	Dimensions	Mem. (GB)
Netflix [33]	80M	480K, 17K, 73	2.4
Outpatient [34]	87M	1.6M, 6K, 13K, 6K, 1K, 192K	4.5
Yahoo! [35]	210M	1M, 625K, 133	6.3
Amazon [36]	1.4B	4.8M, 1.7M, 1.8M	41.5

K, **M**, and **B** stand for thousand, million, and billion, respectively. **NNZ** is the number of non-zeros in the training dataset. **Mem.** is the memory required to store the tensor as a list of *(coordinate, value)* tuples, measured in gigabytes.

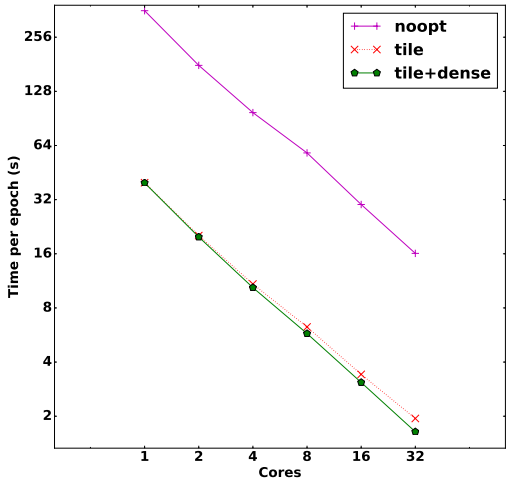


Fig. 6: Effects of ALS optimizations during a rank-10 factorization of the Yahoo! tensor. **noopt** is a baseline ALS implementation with a CSF data structure. **tile** delays rank-1 updates to use the BLAS-3 *dsyrk* routine. **tile+dense** includes **tile** and also dense mode replication.

C. Intra-Method Evaluation

1) ALS: Figure 6 shows the effects of BLAS-3 routines and dense mode replication while factoring the Yahoo! tensor with F=10. With neither optimization, each ALS epoch takes 359 seconds on average and we achieve a $22.3\times$ speedup on 32 cores. BLAS-3 routines improve the runtime by $9\times$, but speedup is reduced to $20.5\times$. Finally, by replicating the dense mode across cores and using a parallel reduction on partial products, we achieve $24.2\times$ speedup with 32 cores and a resulting $219\times$ improvement over the original serial implementation.

2) CCD++: Figure 7 shows the effects of dense mode replication on CCD++. Dense mode replication will not affect serial runtime, and so we show speedup as we scale the number of threads. Speedup is improved from $4.6\times$ to $16.2\times$ due to improved load balance from mode-replication and also from temporal locality. Note that super-linear speedup is observed, reaching $2.3\times$ with two cores. The $p\times p$ tiling of $\mathcal{R}$ for p

[1] http://cs.umn.edu/~shaden/sc16/

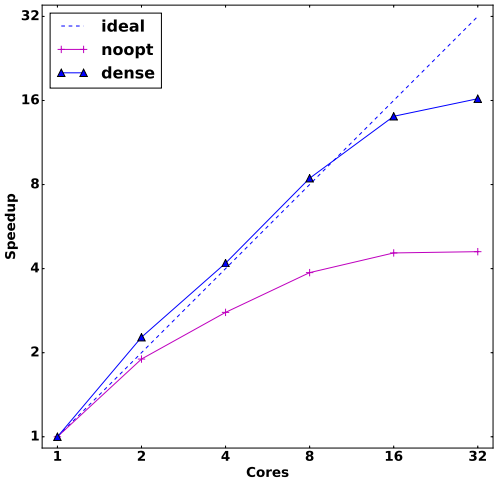


Fig. 7: Effects of CCD++ optimizations during a rank-10 factorization of the Yahoo! tensor. **noopt** is a baseline CCD++ implementation with the CSF data structure. **dense** uses dense mode replication.

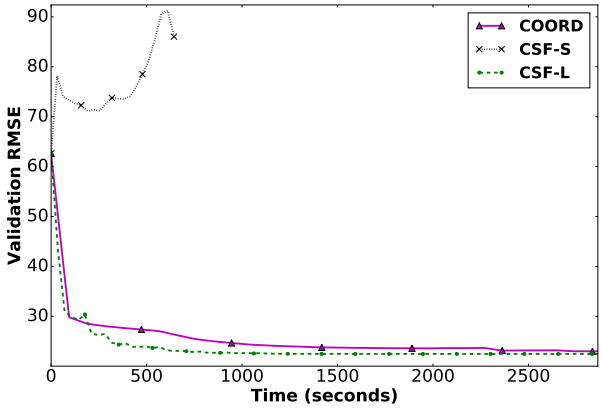


Fig. 8: Effects of randomization strategy on SGD convergence rate during a serial rank-40 factorization of the Yahoo! tensor. **COORD** uses complete randomization on a coordinate form tensor. **CSF-S** randomizes over the shortest mode. **CSF-L** randomizes over the longest mode.

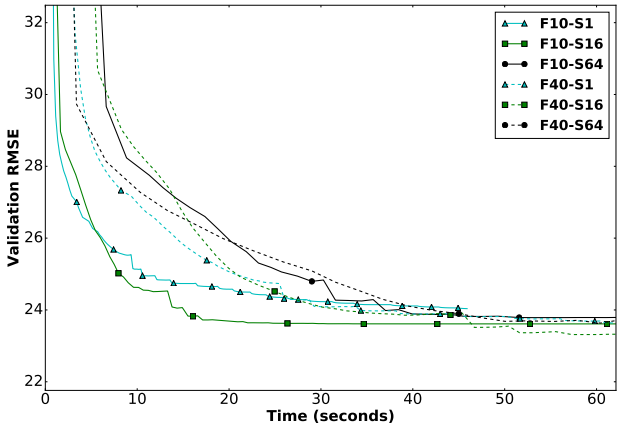


Fig. 9: Convergence rates for SGD parallelization strategies using 32 compute nodes on the Yahoo! dataset. **F** denotes the rank of the factorization. **S** denotes the number of stratum layers, scaling from fully asynchronous (S1) to fully stratified (S64).

threads results in a smaller portion of the factor matrices accessed at a time, improving temporal locality. Interestingly, super-linear speedup was also observed in the original CCD++ evaluation for matrices [29].

Despite the improved speedup, CCD++ sees little improvement after 16 cores (one full socket) due to NUMA effects and the memory-bound nature of the algorithm. Compared to ALS, CCD++ performs a factor of F fewer FLOPs on every non-zero that is accessed. Additionally, CCD++ being a column-oriented method requires MF passes per epoch over the sparsity structure of $\mathcal{R}$, compared to M times for ALS and one time for SGD.

3) SGD: Figure 8 shows the effects of coarse-grained randomization on SGD convergence. We use a full random traversal of the tensor as a baseline and compare against two CSF configurations. The first configuration, CSF-S, sorts the modes in non-decreasing order with the smallest mode placed at the top of the data structure. CSF-S is the default mode ordering used by SPLATT due to it typically resulting in the highest level of compression [4]. CSF-L sorts the modes in non-increasing order, with the longest mode placed at the top of the CSF structure. CSF-L effectively trades additional storage for increased randomization and higher degrees of parallelism. CSF-S has the fastest per-epoch runtime but fails to converge. CSF-L exhibits similar per-epoch convergence compared to the baseline, but the computational savings afforded by the CSF data structure results in a faster time-to-solution. We therefore use CSF-L as the default in the remaining experiments.

Figure 9 shows the effects of stratification on convergence. We evaluate three algorithms across two factorizations: fully asynchronous, fully stratified, and a hybrid algorithm with 16 stratum layers, totaling 256 strata. The hybrid configuration outperforms both baselines for the rank-10 factorization and converges to a higher quality solution in less time. All three algorithms are competitive for rank-40, but the hybrid

D. Comparison Against the State-of-the-Art

In Figure 10 we compare our ALS and CCD++ algorithms against the state-of-the-art MPI implementations [21]. We scale from 1 to 1024 cores on the Yahoo! tensor with F=10. We use one MPI rank per node for opt-ALS due to the high speedup it achieves on 32 cores and also due to the high communication volume that comes with a coarse-grained decomposition. Throughout the comparison we refer to the existing implementations as "base-ALS" and "base-CCD++" and our own as "opt-ALS" and "opt-CCD++". In later discussions

when we reference an ALS or CCD++ without a prefix, we are referring to our own implementations.

On one core, opt-ALS is $7\times$ faster than base-ALS due to the BLAS-3 performance. opt-ALS then scales to achieve $295\times$ speedup at 1024 cores, compared to the $13.5\times$ speedup of base-ALS. The improvements in speedup are due to the coarse-grained decomposition used by opt-ALS which reduces the communication volume from $O(IF^2)$ to $O(IF)$ words. The difference in communication requirements is observed in the ratio of communication to computation: base-ALS spends 95% of the total runtime communicating, compared to opt-ALS which spends 60% of its runtime communicating. opt-ALS is $153\times$ faster than base-ALS when both use 1024 cores.

Serial opt-CCD++ is $2.2\times$ faster than base-CCD++ due to the operation reduction and improved cache locality resulting from the CSF data structure. The locality improvements are also present in the ALS results, but due to ALS being a compute-bound algorithm they are not observed except for very small values of F. On 1024 cores, opt-CCD++ and base-CCD++ achieve $685\times$ and $74.2\times$ speedup, respectively. The disparity in speedups is attributed to the large amount of communication performed by base-CCD++, in which 69% of the total runtime is spent in MPI communication calls. In comparison, opt-CCD++ spends 25% of the total runtime communicating. The medium-grained decomposition used by opt-CCD++ results in a smaller communication volume than the arbitrary decomposition used by base-CCD++. Communication volume is further reduced by utilizing *sparse* communication and only sends an updated value to the processes that require it.

We also note that with the configuration using one MPI rank per socket, opt-CCD++ achieves a $30\times$ speedup on 32 cores compared to $16.2\times$ with a pure OpenMP configuration.

E. Communication Volume

Figure 11 shows the average communication volume per epoch. CCD++ consistently has a lower communication volume than SGD and ALS due to its medium-grained decomposition. The communication volume for SGD increases until four nodes (eight ranks) are used, and then sharply decreases. The communication volume of SGD scales with the number of strata used, which we limit to 64. Recall that SGD uses S^{M-1} strata for an M-mode tensor. Therefore, we see volume increase until we reach the maximum number of strata. From that point communication is limited to within strata, and we see communication volume decrease. The communication volume of ALS increases until eight nodes and then stays near constant. ALS uses a coarse-grained decomposition which in the worst case requires each communication of entire factors per epoch.

F. Strong Scaling

Figure 12 shows strong scaling results. For SGD, we again limit the number of strata to 64 which provides a good trade-off between convergence rate and parallelism. As discussed in the evaluation of communication volume, SGD primarily

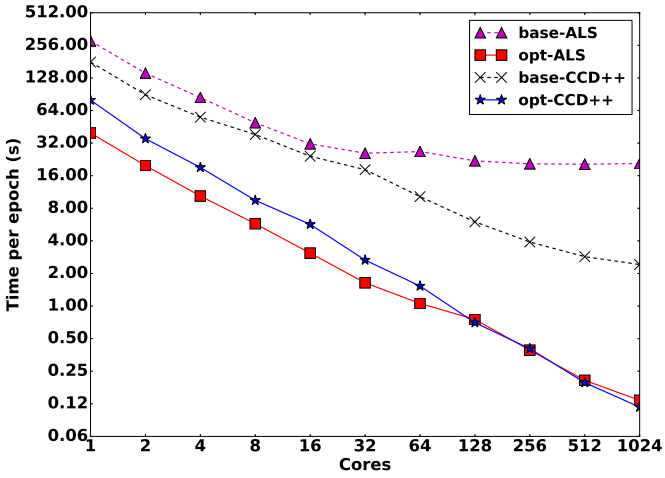


Fig. 10: Comparison of the presented ALS and CCD++ algorithms (prefixed **opt**) against the state-of-the-art MPI implementations (prefixed **base**) on a rank-10 factorization of the Yahoo! tensor.

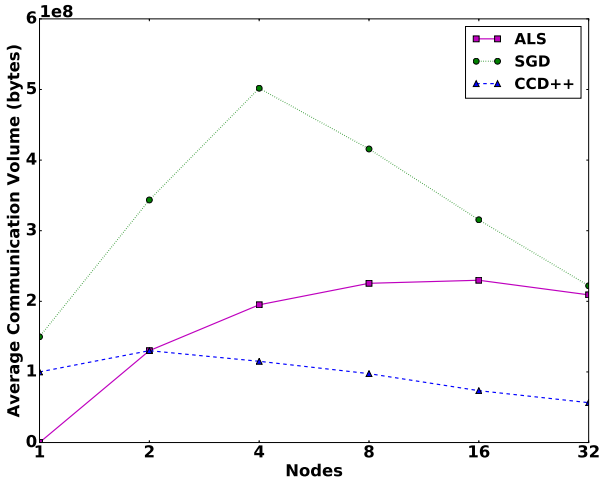
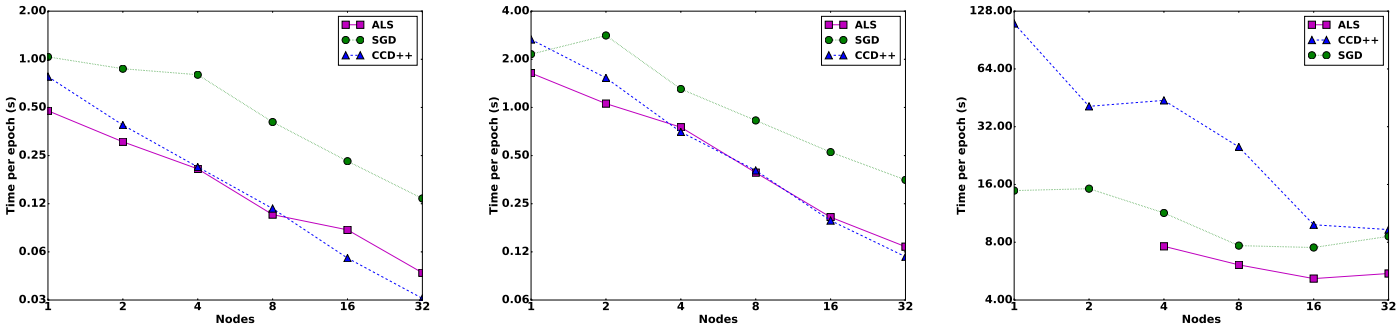


Fig. 11: Average communication volume per node on the Yahoo! dataset. CCD++ and SGD use two MPI ranks per node and ALS uses one.

introduces overhead until the maximum number of strata is reached, and after that point we begin to overcome the communication overheads. SGD only scales to eight nodes on the Amazon tensor. This is because Amazon is significantly more sparse than Netflix and Yahoo! and as a result SGD performs less work per stratum, resulting in high communication costs.

ALS is unable to process the Amazon tensor on less than four nodes due to it needing to store three copies of $\mathcal{R}$ during factorization. Neither ALS nor CCD++ are consistently faster per epoch at 32 nodes. CCD++ begins slower on all datasets but out-scales ALS on all but Amazon. The large size of



(a) Netflix (b) Yahoo! (c) Amazon

Fig. 12: Strong scaling the optimized ALS, SGD, and CCD++ algorithms. Each node has 32 cores.

Amazon is more taxing on memory bandwidth for CCD++ due to it traversing the sparsity structure MF times per epoch.

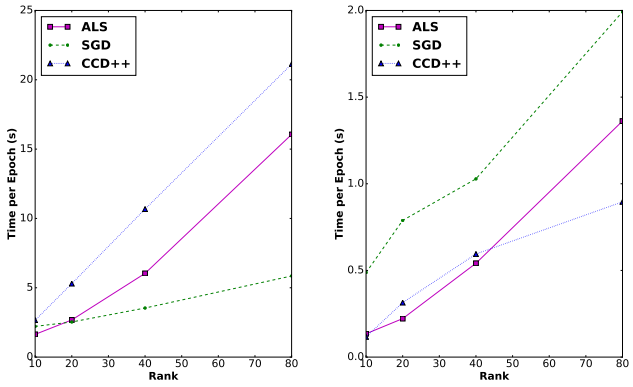
G. Rank Scaling

Figure 13 shows the effects varying the rank of the factorization. We scale from rank 10 to 80 on the Yahoo! dataset. CCD++ and SGD both have $O(F\,\mathrm{nnz}(\boldsymbol{\mathcal{R}}))$ complexity, so we expect the runtime to increase by $8\times$ as we scale F. ALS, on the other hand, has complexity $O(F^2\,\mathrm{nnz}(\boldsymbol{\mathcal{R}}) + IF^3)$. The $F^2\,\mathrm{nnz}(\boldsymbol{\mathcal{R}})$ term will dominate in most scenarios because users are interested in low rank factorizations and because $I \ll \mathrm{nnz}(\boldsymbol{\mathcal{R}})$ for most tensors. Under this assumption, we expect the runtime of ALS to increase by a factor of $80^2/10^2 = 64$.

CCD++ sees the expected linear increase in runtime on both 32 and 1024 cores: $7.9\times$ and $7.6\times$, respectively. SGD scales sub-linearly and only sees $2.4\times$ and $4.2\times$ increases on 32 and 1024 cores, respectively. The sub-linear effects are due to the way SGD accesses the matrix factors. SGD only ever accesses entire rows of the factors, leading to spatial locality and vectorized inner loops. We do not see the same effects for CCD++ because it accesses the factors in a strided manner that is dependent on the sparsity pattern. Additionally, CCD++ must traverse the sparsity pattern of the tensor F times for each mode, compared to once for SGD. Surprisingly, ALS only sees $9.8\times$ and $10.1\times$ increase in runtime at 32 and 1024 cores, respectively. While the work does increase quadratically, all of the quadratic functions are performed by BLAS-3 routines on dense matrices. The work that depends on the sparsity pattern of $\boldsymbol{\mathcal{R}}$ is an MTTKRP operation, which has the same spatial locality as SGD and a complexity of $O(F\,\mathrm{nnz}(\boldsymbol{\mathcal{R}}))$. The BLAS-3 routines will eventually out-scale the cost of the MTTKRP operation, but factorizations of such a high rank are unlikely to be useful to a domain expert.

H. Mode Scaling

Figure 14 shows the scalability of our algorithms as we increase the number of tensor modes while keeping the number of non-zeros constant. ALS sees a roughly linear increase in runtime, matching the computational complexity in Table I. The runtime of the sixth mode increases super-linearly, which we attribute to the sixth mode being longer than most others and ALS having a $O(IF^3)$ complexity component. CCD++ exhibits severe slowdown as the number of modes is increased. This is due to CCD++ doing MF passes over $\boldsymbol{\mathcal{R}}$ per epoch and performing only $O(\mathrm{nnz}(\boldsymbol{\mathcal{R}}))$ work per pass. The memory-bound nature of CCD++ is exaggerated as the number of modes increases. SGD has a nearly constant runtime due to it only performing one pass over $\boldsymbol{\mathcal{R}}$ per epoch, regardless of the number of modes. Additionally, higher-order tensors such as Outpatient have several dense modes which will exhibit high temporal locality, leaving the system's memory bandwidth free for streaming through the single representation of $\boldsymbol{\mathcal{R}}$. SGD appears to be an attractive choice for higher-order tensors. We cautiously recommend it, however, because situations may arise in which many stratum layers are required to maintain convergence, negatively impacting performance.



(a) 32 cores (1 node) (b) 1024 cores (32 nodes)

Fig. 13: Effects of increasing factorization rank on the Yahoo! dataset.

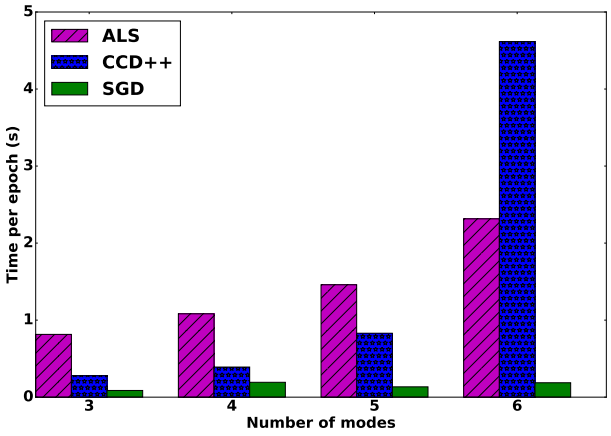


Fig. 14: Average time per epoch with 16 nodes while scaling the number of modes using the Outpatient dataset.

I. Convergence

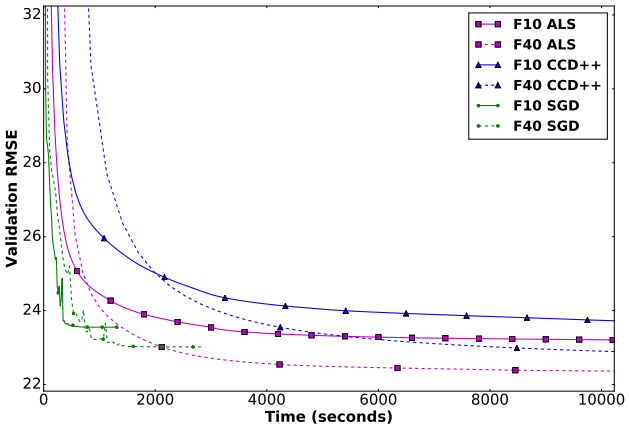
Finally, we evaluate the time-to-solution of ALS, CCD++, and SGD. Figure 15 shows convergence of our optimized algorithms using 1, 32, and 1024 cores. SGD is the most successful algorithm in a serial setting. For F=10, SGD converges within 1500 seconds and achieves a quality that takes ALS over twice as long to reach. SGD sees a similar advantage when F=40, but ALS ultimately reaches a higher quality solution shortly after SGD converges.

On a single node, ALS outperforms SGD and CCD++ for both F=10 and F=40. Since ALS has the fastest per-epoch times in addition to making the most progress per epoch, it is the recommended algorithm for small-to-moderate node counts. CCD++ is also competitive in the multi-core environment and has the added benefit of having a smaller memory footprint due to only storing a single copy of $\mathcal{R}$. Trends continue as we move to a large-scale distributed system and ALS narrowly bests CCD++ for F=40. While ALS still converges faster than SGD and CCD++, CCD++ is more scalable in distribute-memory environments due to its lower communication volume.

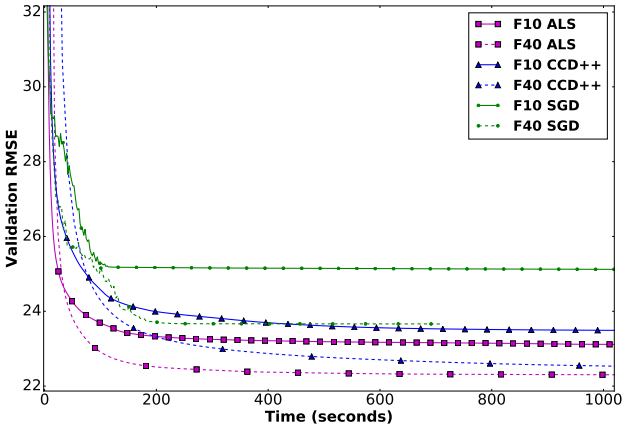
VI. Conclusions

We explored the design and implementation of three optimization algorithms for tensor completion: ALS, SGD, and CCD++. We focused on modern architectures with shared- and distributed-memory parallelism. We addressed issues such as memory- and operation-efficiency, cache locality, load balance, and communication. Following our improvements, we achieve speedups up through 1024 cores and are up to 153× faster than state-of-the-art parallel methods.

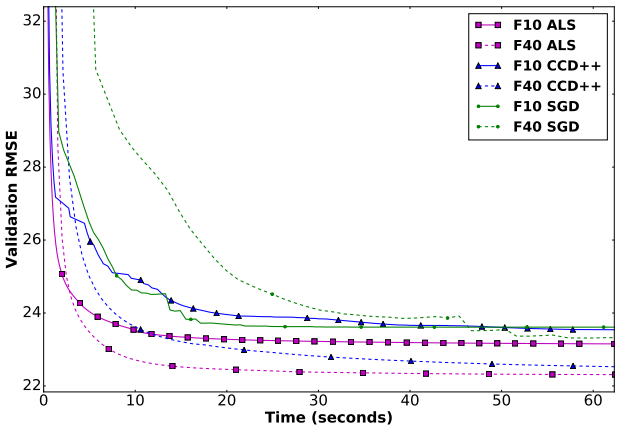
When comparing algorithms for tensor completion, time-to-solution is the most important detail for end users. We compared convergence rates in three configurations: serial, a multi-core system, and a large-scale distributed system and showed that no algorithm performs best in all three environments. SGD is most competitive in a serial environment, ALS is recommended for shared-memory systems, and both ALS



(a) Serial



(b) 32 cores (1 node)



(c) 1024 cores (32 nodes)

Fig. 15: Convergence rates for parallel methods on the Yahoo! dataset with factorization ranks 10 (F10) and 40 (F40). Ticks are placed every five epochs.

and CCD++ are competitive on distributed systems. Using our developments, the time-to-solution on large datasets is reduced from hours to seconds.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for insightful feedback, Mikhail Smelyanskiy for valuable discussions, and Karlsson et al. for sharing source code used for evaluation. This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1265–1274.

[2] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver, "Tfmap: optimizing map for top-n context-aware recommendation," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 155–164.

[3] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–6.

[4] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *International Parallel & Distributed Processing Symposium (IPDPS'15)*, 2015.

[5] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 7.

[6] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 76.

[7] O. Kaya and B. Uçar, "High-performance parallel algorithms for the tucker decomposition of higher order sparse tensors," Ph.D. dissertation, Inria-Research Centre Grenoble–Rhône-Alpes, 2015.

[8] J. H. Choi and S. Vishwanathan, "DFacTo: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.

[9] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[10] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS'16)*, 2016.

[11] H. A. Kiers, "Towards a standardized notation and terminology in multiway analysis," *Journal of chemometrics*, vol. 14, no. 3, pp. 105–122, 2000.

[12] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[13] D. Nion and N. D. Sidiropoulos, "Tensor algebra and multidimensional harmonic retrieval in signal processing for mimo radar," *Signal Processing, IEEE Transactions on*, vol. 58, no. 11, pp. 5693–5705, 2010.

[14] Q. Zhang, M. W. Berry, B. T. Lamb, and T. Samuel, "A parallel nonnegative tensor factorization algorithm for mining global climate data," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 405–415.

[15] J. C. Ho, J. Ghosh, and J. Sun, "Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 115–124.

[16] J. M. ten Berge and N. D. Sidiropoulos, "On uniqueness in candecomp/parafac," *Psychometrika*, vol. 67, no. 3, pp. 399–409, 2002.

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

[18] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*. Springer, 2008, pp. 337–348.

[19] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 667–676.

[20] W. Shao, "Tensor completion," Master's thesis, Universität des Saarlandes Saarbrücken, 2012.

[21] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the cp format," *Parallel Computing*, 2015.

[22] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[23] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 69–77.

[24] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 249–256.

[25] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 655–664.

[26] F. Petroni and L. Querzoni, "GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning." in *Proceedings of the 8th ACM Conference on Recommender systems*. ACM, 2014, pp. 241–248.

[27] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[28] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *Data Mining (ICDM), 2014 IEEE International Conference on*, Dec 2014, pp. 989–994.

[29] H.-F. Yu, C.-J. Hsieh, I. Dhillon *et al.*, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 765–774.

[30] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 974–996, 2004.

[31] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/~splatt/.

[32] R. Battiti, "Accelerated backpropagation learning: Two optimization methods," *Complex systems*, vol. 3, no. 4, pp. 331–342, 1989.

[33] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[34] Center for Medicare and Medicaid Services. (2010) CMS data entrepreneurs synthetic public use file (DE-SynPUF). [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/Downloadable-Public-Use-Files/SynPUFs/index.html

[35] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11." in *KDD Cup*, 2012, pp. 8–18.

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

APPENDIX

Artifact Description: An Exploration of Optimization Algorithms for High Performance Tensor Completion

A. Abstract

The artifact contains all components of the tensor completion algorithms presented in the Supercomputing 2016 paper *An Exploration of Optimization Algorithms for High Performance Tensor Completion*.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Tensor completion via ALS, SGD, and CCD++.
- **Program:** C/C++ code.
- **Compilation:** C and C++ compiler; we used icc/icpc version 16.0.0.
- **Data set:** Tensors formed from public and proprietary datasets.
- **Hardware:** Cori supercomputer at NERSC.
- **Output:** Runtime, factorization quality, and (optionally) factorization result.
- **Experiment workflow:** Git clone source code, download datasets, and run the factorization.
- **Experiment customization:** Yes, the software is general purpose.
- **Publicly available?:** Yes.

2) How delivered: The artifact is implemented as part of SPLATT, an open source library released under the MIT license. Source code and instructions can be found online[2] and also on Github[3].

3) Hardware dependencies: SPLATT has been tested on a variety of x86 machines. A fast interconnect for MPI is recommended.

4) Software dependencies: SPLATT requires CMake version $\geq$ 2.8.0 and a BLAS/LAPACK implementation. The performance of ALS is especially dependent on an optimized BLAS/LAPACK implementation. MPI and OpenMP were both used for experiments, but are not required for compilation or correctness testing.

5) Datasets: Several datasets used for evaluation can be found on our website[4].

C. Installation

Follow the build instructions in `README.md`, found in the SPLATT distribution.

D. Experiment workflow

1) Download and compile the source code:

```
$ git clone https://github.com/ShadenSmith/splatt.git
$ cd splatt
$ git checkout sc16
$ ./configure && make
$ cd build/Linux_x86-64/bin
```

Users can simplify use of the Intel compiler and MKL by passing the flag '`--intel`' during configuration. MPI support requires '`--with-mpi`' during configuration.

2) Download the public datasets:

```
$ url=http://cs.umn.edu/~shaden/sc16/datasets
$ wget ${url}/outpatient3_{train,validate,test}.tns
```

A complete listing of available datasets can be found on our website.

3) Run the code:

```
$ ./splatt complete --alg=ALG --rank=10 \
    outpatient3_train.tns \
    outpatient3_validate.tns \
    outpatient3_test.tns
```

where `ALG` is either `als`, `sgd`, or `ccd`. If users wish to use MPI, run the SPLATT executable with `mpirun` or other suitable program.

E. Evaluation and expected result

SPLATT will output quality (RMSE) and runtime during factorization. Varying the algorithm, dataset, and number of cores will allow users to verify that convergence rates and scalability suitably match the artifact.

F. Experiment customization

The artifact is implemented as part of a general software package for tensor factorization. Therefore, experiments can be easily customized to run with different datasets and parameters by changing command line parameters. Users can run:

```
$ ./splatt --help
```

for detailed runtime options. File format information can be found in `README.md`.

G. Notes

Official releases of SPLATT can be found on our EDU website. Development releases can be found on our Github page. All questions, feedback, or issues are welcome.

[2] http://cs.umn.edu/~splatt/

[3] https://github.com/ShadenSmith/splatt

[4] http://cs.umn.edu/~shaden/sc16/