An Out of Memory tSVD for Big-Data Factorization

HECTOR CARRILLO-CABADA\textsuperscript{1,3}, ERIK SKAU\textsuperscript{2}, GOPINATH CHENNUPATI\textsuperscript{2}, BOIAN ALEXANDROV\textsuperscript{1, (Member, IEEE)}, AND HRISTO DJIDJEV\textsuperscript{2}, (Member, IEEE)

\textsuperscript{1}Los Alamos National Laboratory, Theoretical Division (T-1) Group, Los Alamos, NM 87544, USA
\textsuperscript{2}Los Alamos National Laboratory, Information Sciences (CCS-3) Group, Los Alamos, NM 87545, USA
\textsuperscript{3}Department of Computer Science, The University of New Mexico, Albuquerque, NM 87131, USA

Corresponding authors: Hector Carrillo-Cabada (hcarrillo@lanl.gov) and Hristo Djidjev (djidjev@lanl.gov)

This work was supported in part by the U.S. Department of Energy National Nuclear Security Administration under Contract DE-AC52-06NA25396, in part by the LANL Laboratory Directed Research and Development under Grant 20190020DR, and in part by the U.S. Department of Energy National Nuclear Security Administration under Contract 89233218CNA000001 (This research used resources provided by the Los Alamos National Laboratory Institutional Computing Program).

ABSTRACT

Singular value decomposition (SVD) is a matrix factorization method widely used for dimension reduction, data analytics, information retrieval, and unsupervised learning. In general, only singular values of SVD are needed for most big-data applications. Methods such as tensor networks require an accurate computation of a substantial number of singular vectors, which can be accomplished through truncated SVD (tSVD). Additionally, many real-world datasets are too big to fit into the available memory, which mandates the development of out of memory algorithms that assume that most of the data resides on an external disk during the entire computation. These algorithms reduce communication to disk and hide part of the communication by overlapping it with communication on blocks of work. Here, building upon previous works on SVD for dense matrices, we present a method for computation of a predetermined number, $K$, of SVD singular vectors, and the corresponding $K$ singular values, of a matrix that cannot fit in the memory. Our out of memory tSVD can be used for tensor networks algorithms. We describe ways for reducing the communication during the computation of the left and right reflectors, needed to compute the singular vectors, and introduce a method for estimating the block-sizes needed to hide the communication on parallel file systems.

INDEX TERMS

tSVD, out of memory, tensor train, singular vectors, tensor networks.

I. INTRODUCTION

Large amounts of high-dimensional data is constantly generated from sensor networks, large-scale experiments, massive computer simulations, electronic communications, social-network activities and many others [1]. Most of the collected big datasets (such as imagery, spatiotemporal sensor data, etc.), are naturally high-dimensional, i.e., they are tensors [5]. High-dimensional datasets typically represent multiple concurrent latent processes (explosions, earthquakes, etc.) imprinting their signatures in the observable variables (temperature, pressure, density, etc.) in different dimensions (space/time). The different manifestations of these latent processes allow unique representation of the data. This representation requires unsupervised machine learning methods suitable for high-dimensional big-data, such as SVD [27]. SVD takes a matrix $A$, and factorizes it into a product of two matrices, $A = UH$, where $H = SV^T$, and $U$ and $V$ are orthonormal matrices of singular vectors, while $S$ is a diagonal matrix of singular values. In the truncated SVD (tSVD), only a subset of $K$ column vectors of $U$, the corresponding top $K$ singular values, and the respective row vectors of $V^T$ are calculated. Based on Eckart–Young theorem for low-rank approximation [12], tSVD provides the best low-rank (approximate) representation of the matrix $A$ with rank $K$.

The high-order generalizations of matrix factorizations, such as, Principle Component Analysis [30], Independent Component Analysis [2] and Nonnegative Matrix Factorization [23], are tensor factorizations (TF) with various constraints. The main objective of TF is to find low-rank latent (not directly observable) structures in the data,
by factorizing the multi-dimensional (tensor) dataset into factor matrices and a core tensor [18]. Well-known, classical TF are the Canonical Polyadic Decomposition (CPD) [15] and Tucker Decomposition (TD) [29]. In CPD and TD the factor matrices usually carry the wanted latent processes and structures in each tensor dimension. For tensors of a higher order, \( d (d = 10, 100, \ldots) \), the classical TFs are not feasible since the memory and amount of operations grow exponentially with the order, \( d \). In this case, the tensor networks provide a natural representation of the data and make the factorization possible [8].

One of the simplest tensor networks is Tensor Train (TT) [22], Figure 1a. The advantage of the TT compressed format over the classical TFs is the ability of TT to provide stable rank reduction, achieved through a process of consecutive tSVDs of several large matrices, Figure 1b. The optimization problem of TT often involves dense matrices with billions of entries. With tSVD, one has to extract a substantial number of singular vectors for better reconstruction [22]. Along with TT, most of the other tensor networks use a set of sequential tSVDs [7], therefore tSVD algorithms that are capable of operating on extra-large matrices are needed. Thus, the out of memory tSVD algorithms that can compute a predetermined number of singular vectors of matrices that cannot fit in the memory are important for TT decomposition of tensors that cannot fit into the available memory.

\[
X(n_1, n_2, \ldots, n_N) \approx M_1 \times G_1 \times \ldots \times G_{N-1} \times M_2
\]

(a) Tensor Train

\[
1 \times n_1 \cdots n_N = n_1 \times n_2 \cdots n_N \\
n_1 \cdots n_N = r_1 \times M_1 \approx n_1 \times n_2 \cdots n_N \\
r_1 \times n_2 \cdots n_N = r_2 \times M_2 \approx r_1 \times r_2 \times n_3 \cdots n_N
\]

(b) tSVD operations

**FIGURE 1.** Tensor Train (TT) example. a) The structure of a Tensor Train. b) The algorithm for TT-factorization of an order-\( d \) tensor consists of \( d \) iteration, each requiring the computation of a tSVD. The first iteration, illustrated on the figure, works on data of the same size as the initial tensor.

Here, we propose an out of memory tSVD algorithm suitable for tensor network algorithms. Our approach relies on the existing works in [3], [17], [21], that use a two-phase approach to construct the tSVD suitable for dense matrices. The first phase reduces the input matrix, \( A \) into a product of \( Q_L M Q_R \), where \( Q_L \) and \( Q_R \) are orthonormal and \( M \) is band diagonal. In the second phase, the band diagonal matrix, \( M \) is reduced to a product of \( \tilde{U} S \tilde{V} \) where \( S \) is a diagonal matrix with singular values, \( \tilde{U} \) and \( \tilde{V} \) are orthonormal. Finally, the singular vectors are computed by taking the products \( Q_L \tilde{U} = U \) and \( \tilde{V} Q_R = V^T \).

**A. OUR CONTRIBUTIONS**

We implement an out of memory tSVD algorithm that computes both the singular values and the singular vectors, which can be used in out of memory tensor network algorithms. Our main contributions are as follows.

- We implement an algorithm to compute the left and right reflectors of the reduction to band diagonal phase that are needed to compute the singular vectors in out-of-core fashion as opposed to just computing the singular values as in [17].
- We develop caching strategies to reduce the number of input/output (I/O) operations specific to the access patterns of the left and right reflectors, thereby increasing the efficiency of the implementation.
- Our implementation uses a task based model in a shared memory machine. We analyze the scalability of the implementation with respect to the number of threads.
- We analyze the effects of parallel I/O on the overlap of communication and computation.

**II. BACKGROUND**

Approaches for singular value decomposition typically go through either a two phase or three phase approach [3], [17], [21]. The two phase approach implemented in LAPACK [3] reduces a general band matrix to bidiagonal form in two phases by repeatedly applying Householder transformations [16] to annihilate elements either to the right of the diagonal or below the subdiagonal. This method generates, for an input matrix \( A \), a bidiagonal matrix \( B \), left and right reflectors \( Q_L \) and \( Q_R \) such that \( A = Q_L B Q_R \). Singular values are obtained by reducing \( B \) to diagonal form and computing reflectors \( \tilde{U} \) and \( \tilde{V} \) such that \( A = Q_L \tilde{U} S \tilde{V} Q_R = \tilde{U} S \tilde{V}^T \). While the LAPACK implementation does compute the singular vectors of \( A \), it is not capable of efficient out of memory computation as it requires \( O(N^3) \) word movements [17].

Out of memory algorithms have been developed for one sided factorizations [9], [11], [28], including QR factorization. [25] performs an out of memory SVD on tall-and-skinny matrices by first performing a out of memory QR such that \( A = QR \). This is followed by an in memory SVD on the \( R \) matrix, which has the same singular values as \( A \). Another out of memory attempt in [17] uses a three stage algorithm to compute singular values. The first stage is a general out of memory dense matrix reduction to band diagonal form using a tile algorithm. The second reduces the band to bi-diagonal form and the third reduces the bi-diagonal to diagonal form, which are the resultant singular values. The two stage reduction to bidiagonal form has been used for SVD in [13], [20], [21]. Demchik et al. [10] design an out-of-memory randomized SVD algorithm for both dense and sparse matrices that uses a randomized matrix approximation technique to reduce the size of the input matrix before
computing an SVD. Heavner et al. [14] address the related problem of a rank-revealing factorization as an alternative to tSVD for matrices too large to fit in the RAM. They describe an algorithm for a UTV factorization (introduced in [24]) based on randomization.

The authors in [17] suggest three strategies to either reduce communication with disk or overlap communication with computation, in the first stage, which consists of intertwined QR and LQ phases. The first of these strategies is to cache computation, in the first stage, which consists of intertwined QR and LQ phases. The second strategy involves the fact that we can store the reflectors, \( q \), which we use in the update of the trailing matrix in memory since these are read once per update. The last strategy overlaps communication with computation by reading the next tile, writing the previous tile and computing on the current one at the same time. The authors derive the formula \( b = \frac{3 \alpha}{2} \beta \) for the optimal tile size to hide the cost of communication where \( b \) is the tile size, \( \alpha \) is the computational performance of the machine, and \( \beta \) is the speed of the external storage.

The reduction to band phase is followed by a reduction to diagonal phase, which uses the bulge chasing technique found in [4]. The work in [21] uses the same two step process without any of the strategies from above. Their work uses a task based model reducing of a band matrix to bi-diagonal form. Nevertheless, none of the attempts in the literature calculate the singular vectors, which we address in this paper and the consequent memory related problems in our out of memory tSVD implementation.

III. ALGORITHM

Our method consists of two phases. The first phase reduces a dense matrix \( A \) with tile size \( b \) to band form using the same tile scheme as in [17]. This is followed, in the second phase, with a sparse tSVD on the band matrix. Using this approach, the first phase reduces communication when compared to the single-phase bi-diagonal reduction approach implemented in LAPACK. Unlike [17], we also compute the left and right reflectors needed to compute the singular vectors. After the first phase, the entire matrix consists of zeros except for a diagonal band of elements of width \( b \). In total, there are approximately \( N \times b \) non-zero elements. With a small enough \( b \), the entire matrix can fit in memory as a sparse matrix. We create an in-memory sparse matrix by reading the band tiles from disk and writing the non-zero entries to memory. This allows the use of sparse tSVD solvers that operate in-memory. We note that in the rest of the paper we refer to columns of tiles and rows of tiles as simply columns and rows respectively.

A. FIRST STAGE: REDUCE TO BAND DIAGONAL

Algorithm 1 shows the procedure for transforming an input matrix, \( A \), into a band diagonal form. All of the functions presented are available in LAPACK. As in [17], we store every matrix in column-major tile format. The tile size, \( b \), can be chosen to minimize the runtime of the algorithm. For an \( N \times N \) matrix with tile size \( b \), the total number of tiles is \( u^2 \), where \( u = \frac{N}{b} \). We first update the dense matrix \( A \) in a sequence of intertwined QR and LQ phases which zero out portions of the matrix, while including updates to matrices \( Q_L \) and \( Q_R \), which are defined as the left and right reflectors that transform \( A \) into a band diagonal form. Matrices \( Q_L \) and \( Q_R \) are initialized as identity matrices.

The first of the QR functions is DGEQRT. We can use this to transform diagonal tile, \( A[i, i] \), into upper triangular form and to obtain \( \tilde{Q} \), which we use to update the tiles to the right of \( A[i, i] \). The upper triangular tile is obtained by setting \( A[i, i] = \tilde{Q}^T A[i, i] \) so all other tiles to the right are set to \( A[i, j] = \tilde{Q}^T A[i, j] \). We do not compute \( \tilde{Q}^T A[i, j] \) for tiles to the left since these tiles are zero from the previous QR phases. Since our goal is to compute the left reflector \( Q_L \), we compute \( Q_L[j, i] = Q_L[j, i] \tilde{Q} \) for \( j = 1 \) to \( u \).

Similarly, the algorithm computes QR factorization on pairs of tiles using DTPQRT, which takes as input the diagonal tile \( A[i, i] \) and a tile below it, \( A[i, j] \), and returns an upper triangular matrix, where \( A[i, j] \) is upper triangular and \( A[i, i] \) is zeroed out. This is followed by updates to the tiles in the same row as \( A[i, i] \) and \( A[i, j] \) as shown in Figures 2a and 12b.

Tiles of \( Q_L \) in columns \( i \) and \( j \) are updated as shown in Figures 3a and 3b. LQ factorization follows a similar pattern to produce lower triangular tiles rather than the upper triangular tiles, Figure 2c and 3c. The QR and LQ phases are intertwined to produce the band shape.

B. SECOND STAGE: SPARSE SVD

In the second stage, we convert the band matrix into a sparse matrix with density approximately equal to \( \frac{b}{N} \) and number of non-zero elements approximately \( Nb + N - \frac{b^2}{2} - \frac{b}{2} \). In this stage, we assume that \( Nb \) is small enough to fit in memory. This returns the top \( K \) singular values and vectors in \( \tilde{U}, \tilde{S} \), and \( \tilde{V}^T \) such that

\[
B \approx \tilde{U} \tilde{S} \tilde{V}^T.
\]

Therefore,

\[
A \approx Q_L \tilde{U} \tilde{S} \tilde{V}^T Q_R = U S V^T,
\]

where \( U = Q_L \tilde{U} \) and \( V^T = \tilde{V}^T Q_R \), both of which require a single matrix multiplication to compute. Since \( Q_L \) and \( Q_R \) can both exceed the main memory capacity, the algorithm computes the product by reading in tiles of the matrix \( Q \) and multiplying by tiles of in-memory matrix \( \tilde{U} \) or \( \tilde{V} \).

IV. IMPLEMENTATION

A. TILE CACHING

In this section, we describe caching strategies, which reduce communication cost. The authors of [17] cache the most accessed tiles in the lower right hand side of the matrix to reduce communication. We adopt a similar strategy for caching tiles of matrices \( Q_L \) and \( Q_R \), where we store the leftmost columns and topmost rows respectively for each \( Q \). Since these matrices access tiles differently than matrix \( A \),
Algorithm 1 The Band Reduction Algorithm. L and R Indicate Which Side \( \tilde{Q} \) Is Applied at

```plaintext
1: for i = 1 to u do 
2: \( \tilde{Q} \leftarrow \text{DGEQRT}(A[i, i]) \) \>
\( \triangleright \) QR of \( A[i, i] \)
3: for j = i + 1 to u do 
4: \( \text{DGEMQRT}(\tilde{Q}^T, A[i, j], L) \) \>
\( \triangleright \) Update \( A[i, j] = \tilde{Q}^T A[i, j] \)
5: for j = 1 to u do 
6: \( \text{DGEMQRT}(\tilde{Q}, Q_L[i, i], R) \) \>
\( \triangleright \) Update \( Q_L[i, i] = Q_L[i, i] \tilde{Q} \)
7: for j = i + 1 to u do 
8: \( \tilde{Q} \leftarrow \text{DTPQRT}(A[i, i], A[j, i]) \) \>
\( \triangleright \) QR of \( A[i, i] \)
9: for k = i + 1 to u do 
10: \( \text{DTPMQRT}(\tilde{Q}^T, A[i, k], A[j, k], L) \) \>
\( \triangleright \) Update \( [A[i, k], A[j, k]] = \tilde{Q}^T [A[i, k], A[j, k]] \)
11: for k = 1 to u do 
12: \( \text{DTPMQRT}(\tilde{Q}, Q_L[k, i], Q_L[k, j], R) \) \>
\( \triangleright \) Update \( [Q_L[k, i], Q_L[k, j]] = [Q_L[k, i], Q_L[k, j]] \tilde{Q} \)
13: if i < u then 
14: \( \tilde{Q} \leftarrow \text{DGELQT}(A[i, i + 1]) \) \>
\( \triangleright \) LQ of \( A[i, i + 1] \)
15: for j = i + 1 to u do 
16: \( \text{DGEMLQT}(\tilde{Q}^T, A[i, j + 1], R) \) \>
\( \triangleright \) Update \( A[j, i + 1] = A[j, i + 1] \tilde{Q}^T \)
17: for j = 1 to u do 
18: \( \text{DGEMLQT}(\tilde{Q}, Q_R[i, j + 1], L) \) \>
\( \triangleright \) Update \( [Q_R[i, j + 1], A[i, j + 1]] = [Q_R[i, j + 1], A[i, j + 1]] \tilde{Q}^T \)
19: for j = i + 2 to u do 
20: \( \tilde{Q} \leftarrow \text{DTPLQT}(A[i, i + 1], A[i, j]) \) \>
\( \triangleright \) LQ of \( A[i, i + 1] A[i, j] \)
21: for k = i + 1 to u do 
22: \( \text{DTPMLQT}(\tilde{Q}^T, A[k, j + 1], A[k, j], R) \) \>
\( \triangleright \) Update \( [A[k, i + 1] A[k, j]] = [A[k, i + 1] A[k, j]] \tilde{Q}^T \)
23: for k = 1 to u do 
24: \( \text{DTPMLQT}(\tilde{Q}, Q_R[i + 1, k], Q_R[j, k], L) \) \>
\( \triangleright \) Update \( [Q_R[i + 1, k]] = \tilde{Q} [Q_R[i + 1, k]] \)
```

![Diagram](image_url)

**FIGURE 2.** Various steps of the reduction to band algorithm. Dark blue tiles to the right of the decomposed tiles are updated. Dark green tiles are zeroed out after the decomposition.

Each requires a different caching pattern. \( Q_L \) accesses all tiles an equal number of times throughout the decomposition, \( u \) times in total, and accesses tiles column by column, as shown in Figure 3. \( Q_R \) is similar except tiles are accessed row by row and the topmost row is never accessed because the LQ factorization is not applied to tile \( A[1, 1] \) but rather \( A[1, 2] \). For a square matrix, \( Q_L \) is of dimension \( N \times N \), or with \( u \times u \) tiles. Each tile is read \( u \) times and written \( u \) times for a total of \( 2u^3 \) memory calls. If we can store \( p \) columns of this matrix, we will only have to access the disk \( 2u(u - p) \) times during the first QR sweep as shown in figure 4. After the first update phase, column 1 is never accessed again.
which means the memory used to store column one can store column \( p + 1 \). In the next iteration, the external memory is accessed \( 2u(u-p-1) \) times as shown in Figure 4b. In total, if \( p \) is at least 1, the disk is accessed

\[
\sum_{i=0}^{u-p} 2u(u-p-i) + 2u^2 \text{ times.}
\]

where \( 2u^2 \) is required to read the tiles into memory and write them back out to disk. Note that caching the first column provides the most gains as this caches the one column that is used in the update of the rest of the tiles. Similarly, we can store \( p \) rows of \( Q_R \) and only access the disk

\[
\sum_{i=1}^{u-p} 2u(u-p-i) + 2u(u-1) \text{ times.}
\]

We can further reduce reads and writes to disk by caching the top row of the band matrix during the QR phase and the leftmost column during the LQ phase. As seen in Figure 2a and 2b, the topmost row is used in the update of all tiles below the row. Figure 5 shows that the top row is read \( u \) times and written \( u \) times for a single QR sweep, whereas tiles below are only read and written once. We can cache the tiles in this row during the update using DGEMQRT and DGEMQLT (updates with single tiles) and have these in memory during the two tile updates using DTPMQRT and DTPMLQT. Since DTPMQRT and DTPMLQT require two tiles, one of which is used multiple times in a single update of the trailing matrix, caching the one that is reused reduces the I/O by half. Lastly, since the two tile updates make up the bulk of the operations, this translates to approximately 50% reduction in I/O during the entire SVD. In order to simplify things, and since the LQ and QR phases are broadly speaking identical, the next sections will refer only to the QR phase.

**B. DEPENDENCIES**

We implement our model using task-based parallelism. In our experiments, this showed better performance than a single multi-threaded task model. We ensure that the algorithm is executed in a valid order tying tasks to dependencies. These include data dependencies, ensuring that two tasks never access the same data, and algorithmic dependencies where
one task must always occur before the other. For example, the matrix multiplication updates must always come after a QR factorization task. Furthermore, since every task in a QR sweep uses the top row, as shown in Figure 2, no two tasks can update the same column.

In our model, every QR decomposition (blue nodes in figure 6) creates update tasks which must be performed in order so that overlap of computation and communication can occur. Once an update occurs (red nodes in figure 6), the next update task can proceed (nodes to the right of red nodes). In the standard algorithm, update tasks in the same row are independent. However, since each red node is also reading data for the next task, these tasks must be performed sequentially within a row. That is, update of tile $A[i, k]$ depends on tile $A[i, k - 1]$. Each red node also allows tasks in the same column to initiate. This is because tasks in the same column use the same topmost tile. In practice, each update task contains subtasks for reading the next tile, updating the current tile, and writing the previous one. The I/O tasks are submitted to an I/O thread pool.

![FIGURE 6. Task dependency graph for the first QR sweep of a $N \times N$ matrix with $u = 4$. Blue nodes represent QR factorization task while the other nodes represent updates to tiles.](image)

Furthermore, a lock must be used for tiles to the left and right of the tile that is being modified, because those tiles are being read or written in parallel with the computation. Since the topmost row is shared between tasks, this necessitates that if a task is operating on three tiles in the same row, no other task can operate on tiles that are on a different row but in the same column as the these tiles. Figure 7a shows how these tasks would look like.

In order to increase the parallelism, in our model we assume that the top row and the leftmost column are held in memory, so these are fast reads and writes compared to disk I/O. These tiles are the common tiles shared by all update tasks in a QR/LQ sweep and should be released quickly. To do this, each I/O subtask only reads and writes tiles that are not in the top row. During the compute subtask, the topmost tile necessary for the update is read, computed on, and written. That is, all the subtasks are combined into a single task for the shared tiles only. After this, its lock is released even when the I/O subtasks might not have yet completed. Figure 7b shows that this method increases the number of compute tasks that can be performed in parallel. We use this scheme for the final implementation.

We use the tasks construct in OpenMP to build a DAG representing dependencies between tasks. OpenMP then schedules each task on a thread according to these dependencies.

The DAG does not contain sequential sections. There are sections (e.g., for example, the update of rows), which must be performed sequentially. However, other threads can, and do operate sequentially in their own rows.

We experimentally validated the correctness and accuracy of the algorithm by running on multiple instances of matrices with known singular values. Our tests established the following:

- $|A - USV^\top|_f < \epsilon$,  
- $|UU^\top - I|_f < \epsilon$,  
- $|V^\top V - I|_f < \epsilon$,

where the last two are to ensure that the singular vectors are orthonormal. We set $\epsilon$ to $10^{-5}$.

C. I/O OVERLAP

Each task from the section above can include a read, write, or both. All tasks send their reads and writes to a collection of I/O threads that perform the communication. This allows the threads to overlap the computation of a tile with the read and write of adjacent tiles. The model for I/O overlap is the same as in [17] and so we can use their estimate $\frac{3.2\alpha}{\beta}$.

We use the parallel file system to show improvement in the speed of the algorithm. Specifically, we show that this reduces the optimal tile size needed to overlap communication with computation to

$$\frac{3.2\alpha}{T_{IO}\beta}.$$

Here $\beta$ is the memory bandwidth per IO thread and $T_{IO}$ is the number of threads devoted to IO. In the cases where I/O is the bottleneck, sacrificing computational threads for I/O threads is resulting in better performance.

D. SECOND STAGE: SPARSE SVD

To compute the truncated SVD of the band diagonal matrix, we employ ARPACK [19] through the C++ Armadillo

![FIGURE 7. a) Shows how the subtasks would be distributed without combining subtasks of the shared (top row) tiles and b) shows how they would be distributed with combined subtasks.](image)
library [26]. This uses implicitly restarted Lanczos iterations to efficiently construct the Krylov subspaces of our sparse band matrix. From this, it computes the tSVD, providing the K largest singular values and vectors.

E. MEMORY REQUIREMENTS
The first phase of the algorithm requires enough memory to store the shared tiles in A (top row in the case of the QR phase) and the shared tiles in either QT or QT. Per thread, each one needs space to store matrix Ū, the factors needed to update the rest of the tiles (given by LAPACK as V and T), and the placeholder tiles for the tiles that are read, written, and computed on. V and T are used to compute the product of Householder reflectors H1H2...Hb = H = I − VTVT which are utilized for QR decomposition [3]. Ū and the factors require three tiles. There are three placeholder tiles, one for each subtask. In total, this translates to 8 * b2(6Tf + 2u) minimum required memory for the entire process. Here, Tf is the number of threads.

For example, an SVD on a 128GB matrix, 36 threads, and b = 1024, the memory needed for the first phase is no more than 4GB. For the second phase, the memory used by ARPACK is no more than 8 * (2NK + O(K2)) [19], where K is the number of requested singular values and 8 comes from the number of bytes in a double precision number. The memory used to store the sparse matrix itself is less than ((8 * 3)*(Nb + N − b2/2 − b3/6)), where (Nb + N − b2/2 − b3/6)) is the number of non-zero elements in the band matrix and 3 comes from the usage of three vectors — the row index, column index, and data vectors. For the same matrix as above and 128 singular values, this amounts to approximately 3.3GB.

F. TILE SIZE SELECTION
In this section, we develop a heuristics to select an effective tile size that would result in good performance in the first phase of the algorithm. We first look to reduce the thread idle time. Let us look at the i-th iteration of Algorithm 1. The QR factorization of tile A[i, i] (in line 2 of Algorithm 1) has to be completed before all other steps, since all these steps depend on the value of Ū. After that, however, multiple independent tasks can be done in parallel. Specifically, tile A[i + 1, i] can be factorized, and independently of that factorization all tiles A[i, k] for i + 1 ≤ k ≤ u can be updated sequentially with a single thread. Hence, a good tile size would result in having the time to update the i-th row with one thread to be roughly equal to the time to factorize tiles A[i + 1, i] to A[i + Tt, i] as this would allow all threads to work in parallel updating their own row.

Let us denote the times for factorization and the update rate of a tile (essentially just a matrix multiplication) by tf(b) and tm(b), respectively. Since u = N/b, then the above condition can be written as

$$tf(b) \approx \frac{N/b \times tm(b)}{Tt - 1},$$

or, ignoring the −1 in the denominator,

$$b \approx \frac{N \times \ln(b)}{Tt \times \beta}.$$

If only doing a single iteration of the algorithm, we heuristically find that \(\frac{\ln(b)}{Tt \times \beta}\) can be approximated by γ where \(\frac{1}{2} < \gamma < 1\). Since in further iterations of the algorithm the number of the tiles that we need to update decreases from u per row in subsequent iterations as the trailing matrix gets smaller. In practice, we find that setting \(\gamma = \frac{1}{4}\) gives us good results for the full algorithm.

Next, we look at the communication time. This is given by [17] as

$$b \approx \frac{N \times \frac{3.2a}{4Tf}}{\beta},$$

For matrices most of the matrix fits in memory we can choose

$$b \approx \frac{N}{4Tf}.$$

V. EXPERIMENTAL RESULTS
We ran most of our experiments on a two 18 core Intel Broadwell E5-2695V4 processors running at 2.1 GHz clock speed, 45MB shared cache, 128GB RAM, and peak performance of 1.2 TF/s. This node is connected to a parallel Lustre file system. We ran an additional experiment on an Intel Broadwell E5-2650V4 with 128 GB of RAM and 24 hyper-threads.

A. I/O OVERLAP
We ran our algorithm on three matrix sizes with N = 80000, N = 100000, and N = 120000 with 128 singular values and 500 × 500 tile size with varying number of computation and I/O threads. Together with the reflectors, these matrices are too large to fit in memory.

Figure 8 compares the average time each thread spent in I/O, computing, and idling for 1-6 I/O threads. Since I/O time and computation time are overlapped, we measure I/O time as the time a thread spends waiting for an I/O request to complete after completing its computation.

Despite removing computation threads to increase I/O threads, computation time does not noticeably increase. We do not achieve complete overlap, however, we get massive gains to I/O speed. We can see that, from one to two I/O threads, the non-overlapped I/O time is reduced by more than a half. Parallel I/O reduces the optimal tile size needed to overlap computation with communication with up to four I/O threads by increasing the I/O bandwidth. With more than four I/O threads, we get more varied results. As shown in figure 8(a), having more I/O threads reduces communication time consistently, but for (b) and (c) we get either no noticeable reduction or, in the case of (b), an increase in communication time. Overall, however, for up to four I/O
FIGURE 8. Time breakdown for an out of memory SVD with different number of I/O threads for matrices of size a) $N = 80,000$, b) $N = 100,000$, and c) $N = 120,000$ with tile size $b = 500$.

FIGURE 9. Execution time of the two phases and the total running time for matrices with (a) $N = 20160$, (b) $N = 45360$, (c) $N = 83160$, (d) $N = 110880$, for various tile sizes. Matrices in (a) and (b) fit in memory while (c) and (d) are larger than memory.

threads, this gives us more options in terms of choosing a tile size that reduces global execution time. The equation given in [17] for the optimal tile size to overlap computation with communication then becomes $b \geq \frac{32\alpha}{T_{\text{IO}}\beta}$ on parallel file systems, where $\beta$ is the communication bandwidth for serial I/O.
### B. TILE SELECTION

We analyze the effects of tile size on execution time of our code. We run our algorithm on square matrices of sizes $N = 20160$ and $N = 45360$ that fit in memory, and each run with one I/O thread. We also run our code on matrices of size $N = 83160$ and $N = 110880$ that do not fit in memory, and run with two I/O threads to lessen non-overlapped communication. The elements of the matrices are generated from random uniform distribution, and we compute the top 128 singular vectors and values. Note that, since the matrices are uniform random, there exists one very large singular value, followed by tightly clustered singular values. Since the implicitly restarted Lanczos method used by LAPACK tends to under-perform for these matrices, the execution time of the second phase of our method is an upper bound of what can be expected for most real world data.

Figure 9 shows that, as expected, the second phase performs best for small tile sizes and the execution time increases as the tile size increases. We observe that, for all the matrices, relatively small tile sizes perform the worst for the first phase of the algorithm. The first phase time then stabilizes but continues to increase for large tile sizes. The reason that larger tile sizes are less effective is because they limit the amount of parallelism. If $u < T_r$ then at most $u$ threads can work in parallel.

We compared the execution time of runs using a tile size using selected using our tile selection method and the tile with the minimum idle time in Table 1. Tile selection is modified so that only sizes $b$ which are factors of $N$ are chosen. We set $b$ to the factor of $N$ which is closest to $\max\left\{\frac{N}{N}, \frac{3.20}{p}\right\}$. We find that, as a fraction of the total time spent idling in phase 1, the estimated tile size is either close to the best tile size or the fraction spent idling is close the best idling time, with the exception of $N = 20160$. With smaller $N$, we notice the same issue. However, the rule holds for all larger matrices that we tried.

One final observation is that as the matrix size increases, the running time of the algorithm becomes increasingly dominated by the first phase running time. In Table 2 we show that for most tile sizes which are optimal for phase 1 their global execution time is close to the optimal global execution time. This is because the execution time for the first phase grows faster than the execution time for the second phase when we fix the number of singular values, $K$.

### C. STRONG SCALING

We analyze the effect of number of threads on the performance of the algorithm. We run the tSVD on a $32768 \times 32768$ matrix with 1, 2, 4, 8, 16, and 32 computation threads, 1 I/O thread, and tile sizes 128, 256, 512, 1024. Figure 10 shows that, when the tile size is 256, which is near the proposed optimal for 32 threads, we achieve 96% scaling efficiency with 32 threads; however, with all other sizes this reduces to approximately 50%. We also note that, with 4 threads, we achieve higher single thread performance than using only a single thread. This happens because we choose the internal LAPACK tile size at run time and these can have effect on the execution time.

---

### TABLE 1. Tile selection and idle time.

| $N$   | CPU  | Compute Threads | Tile Selection | Percent Idle for Selected Tile | Optimal Tile | Optimal Idle Percentage |
|-------|------|-----------------|----------------|-------------------------------|--------------|------------------------|
| 20160 | E5-2695V4 | 35              | 144            | 26.6%                         | 252          | 14.8%                  |
| 45360 | E5-2695V4 | 35              | 315            | 8.1%                          | 315          | 8.1%                   |
| 50400 | E5-2695V4 | 35              | 360            | 7.3%                          | 360          | 7.3%                   |
| 83160 | E5-2695V4 | 34              | 594            | 4.5%                          | 462          | 2.4%                   |
| 110880| E5-2695V4 | 34              | 880            | 6.0%                          | 880          | 6.0%                   |
| 50400 | E5-2650V4 | 23              | 525            | 6.4%                          | 360          | 4.8%                   |

### TABLE 2. Optimal tile size for phase 1 and global optimal.

| $N$    | CPU  | Optimal Tile Size for Phase 1 | Phase 1 Optimal Time | Global Optimal Tile Size | Phase 1 Time with Global Optimal Tile Size |
|--------|------|-------------------------------|-----------------------|--------------------------|------------------------------------------|
| 20160  | E5-2695V4 | 320                          | 3.61                   | 192                      | 3.03                                     |
| 45360  | E5-2695V4 | 336                          | 22.73                  | 336                      | 22.73                                    |
| 50400  | E5-2695V4 | 420                          | 29.65                  | 420                      | 29.65                                    |
| 83160  | E5-2695V4 | 1540                         | 134.64                 | 1188                     | 134.07                                   |
| 110880 | E5-2695V4 | 2640                         | 320.88                 | 1848                     | 277.04                                   |
| 50400  | E5-2650V4 | 840                          | 49.86                  | 480                      | 47.86                                    |
We introduced novel strategies to cache tiles of inputs matrix, computes the top $K$ singular values along with the corresponding singular vectors of a large matrix. In our approach, we used the well-known two-phased approach for SVD, where in the first phase the input matrix is transformed into band diagonal from which, in the second phase, to a diagonal matrix. We introduced novel strategies to cache tiles of inputs matrix, reflectors ($Q_L$ and $Q_R$), thereby reduce communication costs. We presented an optimal range of tile sizes for our algorithm for which the idle time is minimized. We derived a simple heuristic to estimate the optimal tile size. In the future, we will extend our framework to distributed (similar to [6]) out of memory TSVD. Finally, we show the effect of parallel I/O on non-overlapped communication time when using a parallel file system.

VI. CONCLUSION

We implemented an out of memory tSVD algorithm that computes the top $K$ singular values along with the corresponding singular vectors of a large matrix. In our approach, we used the well-known two-phased approach for SVD, where in the first phase the input matrix is transformed into band diagonal from which, in the second phase, to a diagonal matrix. We introduced novel strategies to cache tiles of inputs matrix, reflectors ($Q_L$ and $Q_R$), thereby reduce communication costs. We presented an optimal range of tile sizes for our algorithm for which the idle time is minimized. We derived a simple heuristic to estimate the optimal tile size. In the future, we will extend our framework to distributed (similar to [6]) out of memory TSVD. Finally, we show the effect of parallel I/O on non-overlapped communication time when using a parallel file system.

REFERENCES

[1] B. Akhtar, G. B. Saathoff, H. R. Arabnia, R. Hill, A. Staniforth, and P. S. Bayerl, Application of Big Data for National Security: A Practitioner’s Guide to Emerging Technologies, London, U.K.: Butterworth, 2015.

[2] S.-I. Amari, A. Cichocki, and H. H. Yang, “A new learning algorithm for blind signal separation,” in Proc. Adv. Neural Inf. Process. Syst., 1996, pp. 757–763.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKeny, and D. Sorensen, LAPACK Users' Guide, vol. 9. Philadelphia, PA, USA: SIAM, 1999.

[4] P. Bientinesi, F. D. Igual, D. Kressner, and E. S. Quintana-Ortí, “Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures,” in Proc. Int. Conf. Parallel Process. Appl. Math. Berlin, Germany: Springer, 2009, pp. 385–397.

[5] M. Boussè, O. Debals, N. Vervliet, F. Van Eeghem, and L. De Lathauwer, “Non-overlapped communication time when using a parallel file system,” in Proc. Int. Conf. Parallel Process. Appl. Math. Berlin, Germany: Springer, 2009, pp. 385–397.

[6] G. Chennupati, R. Vangara, E. Skau, H. Djidjev, and B. Alexandrov, “Distributed non-negative matrix factorization with determination of the number of latent features,” J. Supercomput., Feb. 2020.

[7] A. Cichocki, “Tensor networks for dimensionality reduction, big data and deep learning,” in Advances in Data Analysis With Computational Intelligence Methods. Cham, Switzerland: Springer, 2018, pp. 3–49.

[8] A. Cichocki, N. Lee, I. V. Oseledets, A.-H. Phan, Q. Zhao, and D. Mandic, “Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges PART 1,” 2016, arXiv:1609.00893. [Online]. Available: http://arxiv.org/abs/1609.00893

[9] E. D’Azevedo and J. Dongarra, “Design and implementation of the parallel out-of-core ScalAPACK LU, QR, and Cholesky factorization routines,” Concurrency-Pract. Exper., vol. 12, no. 15, pp. 1481–1493, Dec. 2000.

[10] V. Demchik, M. Bačák, and S. Bordag, “Out-of-core singular value decomposition,” 2019, arXiv:1907.06470. [Online]. Available: http://arxiv.org/abs/1907.06470

[11] J. J. Dongarra, S. Hammarling, and D. W. Walker, “Key concepts for parallel out-of-core LU factorization,” Comput. Math. Appl., vol. 35, no. 7, pp. 13–31, Apr. 1998.

[12] C. Eckart and G. Young, “The approximation of one matrix by another of lower rank,” Psychometrika, vol. 1, no. 3, pp. 211–218, Sep. 1936.

[13] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra, “A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction,” in Proc. IEEE 29th Int. Parallel Distrib. Process. Symp., May 2012, pp. 25–35.

[14] N. Heavner, P.-G. Martinsson, and G. Quintana-Ortí, “Computing rank-revealing factorizations of matrices stored out-of-core,” 2020, arXiv:2002.06960. [Online]. Available: http://arxiv.org/abs/2002.06960

[15] F. L. Hitchcock, “The expression of a tensor or a polyadic as a sum of products,” J. Math. Phys., vol. 6, nos. 1–4, pp. 164–189, Apr. 1927.

[16] A. S. Householder, “Unitary triangularization of a nonsymmetric matrix,” J. ACM, vol. 5, no. 4, pp. 339–342, Oct. 1958.

[17] K. Kabir, A. Haidar, S. Tomov, A. Bouteiller, and J. Dongarra, “A framework for out of memory SVD algorithms,” in Proc. Int. Supercomput. Conf., Switzerland: Springer, 2017, pp. 158–178.

[18] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” SIAM Rev., vol. 51, no. 3, pp. 455–500, Aug. 2009.

[19] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems With Implicitly Restarted Arnoldi Methods, vol. 6. Philadelphia, PA, USA: SIAM, 1998.

[20] H. Ltaief, P. Luszczek, and J. Dongarra, “Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction,” in Proc. Int. Conf. Parallel Process. Appl. Math., vol. 7203, Sep. 2011, pp. 661–670.

[21] H. Ltaief, P. Luszczek, and J. Dongarra, “High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures,” ACM Trans. Math. Softw., vol. 39, no. 3, pp. 1–22, Apr. 2013.

[22] I. V. Oseledets, “Tensor-train decomposition,” SIAM J. Sci. Comput., vol. 33, no. 5, pp. 2295–2317, Jan. 2011.

[23] E. Rabani and S. Toledo, “Out-of-core SVD and QR decompositions,” in Proc. 26th Int. Parallel Distrib. Process. Symp., May 2012, pp. 25–35.

[24] A. Chennupati, R. Vangara, E. Skau, H. Djidjev, and B. Alexandrov, “Distributed non-negative matrix factorization with determination of the number of latent features,” J. Supercomput., Feb. 2020.

[25] H. Carrillo-Cabada et al.: Out of Memory TSVD for Big-Data Factorization

FIGURE 10. Strong scaling with four different tile sizes.
ERIK SKAU received the B.Sc. degree in applied mathematics and physics, and the M.Sc. and Ph.D. degrees in applied mathematics from North Carolina State University, Raleigh, NC, USA. He is currently a Postdoctoral Research Associate with the Los Alamos National Laboratory (LANL), Information Sciences (CCS-3) Group. His research and technical interests include signal processing, matrix and tensor factorization, and machine learning.

GOPINATH CHENNUPATI received the Ph.D. degree from the University of Limerick, Ireland. He is currently a Computer Scientist with the Los Alamos National Laboratory (LANL), Information Sciences (CCS-3) Group. He works on high performance computing (HPC), performance modeling, natural language processing (NLP), deep/machine learning, high performance linear algebra, and so on.

BOIAN ALEXANDROV (Member, IEEE) received the M.Sc. degree in theoretical physics, and the first Ph.D. degree in nuclear engineering and the second Ph.D. degree in computational biophysics. He is currently a Scientist with the Los Alamos National Laboratory, Theoretical Division (T-1) Group. He is specialized in big data analytics, matrix and tensor factorization, unsupervised learning, and latent feature extraction.

HRISTO DJIDJEV (Member, IEEE) received the M.Sc. degree in applied mathematics and the Ph.D. degree in computer science from Sofia University, Bulgaria. He is currently a Computer Scientist with the Los Alamos National Laboratory (LANL), Information Sciences (CCS-3) Group. Before joining LANL as a Scientist, he worked as an Assistant Professor with Rice University and a Senior Lecturer with Warwick University. He is also a Research Adjunct Professor with Carleton University, Ottawa, ON, Canada.