Randomized Algorithms to Update Partial Singular Value Decomposition on a Hybrid CPU/GPU Cluster

Ichitaro Yamazaki, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra
{iyamazak, kurzak, luszczek, dongarra}@eecs.utk.edu
Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville, Tennessee, U.S.A.

ABSTRACT

For data analysis, a partial singular value decomposition (SVD) of the sparse matrix representing the data is a powerful tool. However, computing the SVD of a large matrix can take a significant amount of time even on a current high-performance supercomputer. Hence, there is a growing interest in a novel algorithm that can quickly compute the SVD for efficiently processing massive amounts of data that are being generated from many modern applications. To respond to this demand, in this paper, we study randomized algorithms that update the SVD as changes are made to the data, which is often more efficient than recomputing the SVD from scratch. Furthermore, in some applications, recomputing the SVD may not be possible because the original data, for which the SVD has been already computed, is no longer available. Our experimental results with the data sets for the Latent Semantic Indexing and population clustering demonstrate that these randomized algorithms can obtain the desired accuracy of the SVD with a small number of data accesses, and compared to the state-of-the-art updating algorithm, they often require much lower computational and communication costs. Our performance results on a hybrid CPU/GPU cluster show that these randomized algorithms can obtain significant speedups over the state-of-the-art updating algorithm.

1. INTRODUCTION

A partial singular value decomposition (SVD) [12] of a sparse matrix is a powerful tool for data analysis, where the data is represented as the sparse matrix. The ability of the SVD to filter out noise and extract the underlying features of the data has been demonstrated in many applications, including Latent Semantic Indexing (LSI) [8, 2], recommendation systems [8, 30], population clustering [26], and subspace tracking [18]. The SVD is also used to compute the leverage scores – statistical measurements for sampling the data in order to reduce the cost of the data analysis [14]. However, in recent years, the amount of data being generated from the observations, experiments, and simulations is growing at unprecedented paces in many areas of studies, e.g., science, engineering, medicine, finance, social media, and e-commerce [7, 9]. These phenomenon is commonly referred to as “Big Data” and the algo-

rithmic challenges to analyze the data are exacerbated by its massive volume and wide variety as well as its high veracity and velocity [22]. Though the SVD has the potential to address the variety and veracity of the modern data sets, the traditional approaches to computing the partial SVD access the data repeatedly, e.g., block Lanczos [11]. This is a significant drawback on a modern computer, where the data access has become significantly more expensive compared to arithmetic operations, both in terms of time and energy consumptions. This gap between the communication and computation costs is expected to further grow on future computers [10, 13].

To address this hardware trend, a randomized algorithm [14] has been gaining attention since compared to the traditional approaches, it often computes the SVD with fewer data accesses. Though such algorithms have the potential to efficiently compute the SVD on modern computers, there are several obstacles that need to be overcome. In particular, the randomized algorithm may require only a small number of data accesses, but each data access can be expensive due to the irregular sparsity pattern of the matrix and the power-law distribution of its nonzeros. Though several techniques to avoid such communication have been proposed [16], these techniques may not be effective for computing the SVD of the modern data because they often require a significant computational or communication overhead due to the particular sparsity structure of the matrix [37].

In this paper, we study randomized algorithms to update (rather than recompute) the partial SVD as the changes are made to the data set. This is an attractive approach because compared to recomputing it from scratch, the SVD may be updated more efficiently, while in modern applications, the existing data are being constantly updated and new data is being added. Moreover, in some applications, recomputing the SVD may not be possible because the original data, for which the SVD has been already computed, is no longer available. At the same time, in modern applications, the size of the update is significant even though it is much smaller than the massive data that has been already compressed. Therefore, an efficient updating algorithm is needed to address the large volume and high velocity of the modern data sets. Such applications with the rapidly changing data include the communication and electric grids, transportation and financial systems, personalized services on the internet, particle physics, astrophysics, and genome sequencing [7].

There are four main contributions of the paper. First, we introduce three different schemes to update the partial SVD based on a randomized algorithm and analyze their communication and computation costs. Second, to study the effectiveness of the randomized algorithm, we present case-studies with two particular applications, i.e., LSI and population clustering. Our case-studies use the data sets from real applications and demonstrate that the randomized

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC '15, November 15-20, 2015, Austin, TX, USA

Copyright 2015 ACM 978-1-4503-3723-6/15/11 \$15.00

<http://dx.doi.org/10.1145/2807591.2807608>

algorithms converge quickly to obtain the desired accuracy of the SVD, e.g., three passes over the data suffice. Third, we compare the performance of the randomized algorithms with that of the state-of-the-art updating algorithm on a hybrid CPU/GPU cluster. Our performance results demonstrate that the randomized algorithms can obtain significant speedups, as much as $14.1\times$ in our experiments. Finally, we present our effort to improve the performance of the randomized algorithms based on the communication-avoiding technique [19]. Our performance results on a hybrid CPU/GPU cluster show the potential of the implementation. In the final section, we list our current efforts to improve the robustness of this communication-avoiding implementation.

The rest of the paper is organized as follows. After listing related work in Section 2, we describe the state-of-the-art updating algorithm and our randomization schemes for updating the partial SVD in Section 3. Then, in Section 4, we present our case-studies of using the randomization schemes for LSI and population clustering. Next, in Section 5, we describe our implementation of the randomized algorithm on a hybrid CPU/GPU cluster, and in Section 6, we analyze its computational and communication complexities. Finally, in Section 7, we present our performance studies on a hybrid cluster. The communication-avoiding variant of the randomized algorithm and the final remarks are listed in Sections 8 and 9, respectively.

All of our experiments were conducted on the Tsubame Computer at the Tokyo Institute of Technology¹. Each of its compute nodes consists of two six-core Intel Xeon CPUs and three NVIDIA Tesla K20Xm GPUs. We compiled our code using the GNU gcc version 4.3.4 compiler and the CUDA nvcc version 6.0 compiler with the optimization flag `-O3`, and linked it with Intel’s Math Kernel Library (MKL) version xe2013.1.046.

2. RELATED WORK

In recent years, several researchers have demonstrated the ability of the randomized algorithm to analyze the emerging large data sets [14, 23]. This paper is an extension of these studies to adapt the randomized algorithm for updating SVD and to study its performance on a hybrid CPU/GPU cluster.

Several algorithms have been proposed to update the partial SVD for the latent semantic indexing. For example, the “fold-in” algorithm [3] is an efficient algorithm, but the precision of a query result could deteriorate as more updates are incorporated into the SVD [33, 34]. The current state-of-the-art algorithm is the “updating” algorithm [41], and it has been shown to obtain the same precision as that of the recomputed partial SVD. To reduce the computational cost of the updating algorithm, the “fold-up” algorithm [33] is based on the fold-in algorithm, but periodically discards the updates and recomputes them using the updating algorithm. When the SVD is continually updated with small changes, the fold-up is an efficient approach and obtains the same accuracy as the updating algorithm. We compare the performance of our randomized algorithm with the updating algorithm, and demonstrate that the randomized algorithm can potentially replace the updating algorithm in the fold-up algorithm.

Recently, a column-wise Lanczos was used to update the SVD [34]. The experimental results in MATLAB demonstrated the efficiency of the algorithm compared to the updating algorithm. However, compared to the randomized algorithm studied in this paper, Lanczos would require more data accesses to generate the projection subspace of the same dimension. Hence, on a large scale computer, the Lanczos would likely suffer from the greater communi-

cation latency. To compute the SVD on a single compute node with multiple GPUs, we have previously compared the performance of the randomized algorithm with the block Lanczos, which often requires fewer data accesses than the column-wise Lanczos, and with a communication-avoiding variant of the block Lanczos [37]. In that study, when the randomized algorithm requires a small number of iterations, even on a single compute node with a marginal inter-GPU communication cost, the randomized algorithm was often more efficient. In addition, compared to the updating algorithm, the projection scheme proposed in [34] reduces the serial bottleneck of solving the projected problem, but it could still be significant (in terms of both time and memory). Our performance results on a hybrid CPU/GPU cluster demonstrate that our projection schemes further reduce the bottleneck, obtaining a significant speedup, while maintaining a similar accuracy.

A main computational kernel of the randomized algorithm is the sparse-matrix dense-matrix multiply (SpMM). It has been shown that for the sparse matrix whose nonzero distribution exhibits a power law, the parallel scaling of SpMM can be improved by distributing the sparse matrix in a 2D block format [5]. For our communication-avoiding variant of the randomized algorithm, another important computational kernel is the symmetric sparse-matrix sparse-matrix multiply (SpSYRK). Several ideas have been explored to improve the parallel scaling of SpSYRK [6]. Though we do not investigate these techniques in this paper, they complement our current studies, and we plan to integrate them into our implementation. We are currently investigating any extensions to these techniques that might be needed for particular shapes of our matrices – especially on a GPU.

A few software packages exist for computing partial SVD. These packages are often developed by researchers from a single discipline and optimized for their specific needs (e.g., SLEPc² from linear algebra and GraphLab³ from graph analysis). In addition, they do not provide a functionality to update SVD. Our ultimate goal is to develop a flexible interface to efficient and robust SVD solvers for a wide range of applications by combining techniques from linear algebra, randomized algorithm, and high performance computing.

3. ALGORITHMS

We assume that a rank- k approximation A_k of the m -by- n matrix A has been computed as $A_k = U_k \Sigma_k V_k^T$, where Σ_k is a k -by- k diagonal matrix whose diagonal entries approximate the k largest singular values of A , and the columns of U_k and V_k approximate the corresponding left and right singular vectors, respectively. We consider the updating problem of computing a rank- k approximation of an m -by- $\hat{n}$ matrix $\hat{A}$,

$$\hat{A} \approx \hat{U}_k \hat{\Sigma}_k \hat{V}_k, \quad (1)$$

where $\hat{A} = [A, D]$ and an m -by- d matrix D represents the new sets of the sparse columns being added to A . In many cases, D is tall-and-skinny: it has more rows than columns, i.e., $m \gg d$. This updating problem was referred to as the *updating-document* problem [3]. Two other problems, *updating-terms* and *term-weight-correction* problems, were considered, which add new rows to the matrix A and updates the nonzero values of A , respectively. These three types of updates can be all represented by sparse low-rank updates to A . Though we only focus on the updating-document problem, much of our discussion can be extended to the other two problems.

²<http://slepc.upv.es>

³<http://graphlab.org>

¹<http://tsubame.gsic.titech.ac.jp>

```

for  $j = 1, 2, \dots, s$  do
  1. Orthogonalize  $\hat{Q}$ 
  2.  $QR := \hat{Q}$ 
   Sample range of  $A$ 
    $\hat{P} := AQ$ 
  3. Orthogonalize  $\hat{P}$ 
    $PB := \hat{P}$ 
  4. Prepare to iterate (sample range of  $A^T$ )
   if  $j < s$  then
      $\hat{Q} := A^T P$ 
   end if
end for

```

Figure 1: Randomized algorithm to generate projection subspaces P and Q based on power iterations.

All the algorithms studied in this paper belong to a class of subspace projection methods which are based on the following three steps:

1. Generate a pair of $k + \ell$ orthonormal basis vectors $\hat{P}$ and $\hat{Q}$ that approximately span the ranges of the matrices $\hat{A}$ and $\hat{A}^T$, respectively,

$$\hat{A} \approx \hat{P}\hat{Q}^T,$$

where ℓ is referred to as an oversampling parameter [14] and selected to improve the performance or robustness of the algorithm.

2. Use a standard deterministic algorithm to compute SVD of the projected matrix B ,

$$B = X\hat{\Sigma}Y^T, \quad (2)$$

where $B = \hat{P}^T \hat{A} \hat{Q}$.

3. Compute the approximation to the partial SVD of $\hat{A}$,

$$\hat{A}_k \approx \hat{U}_k \hat{\Sigma}_k \hat{V}_k^T, \quad (3)$$

where $\hat{U}_k = \hat{P}X_k$ and $\hat{V}_k = \hat{Q}Y_k$. Here, $\hat{\Sigma}_k$ is the k -by- k diagonal matrix whose diagonal entries are the k largest singular values of B , and X_k and Y_k are the corresponding left and right singular vectors, respectively.

In this section, we describe algorithms that generate the basis vectors $\hat{P}$ and $\hat{Q}$.

3.1 Randomized Algorithm

To compute the partial SVD of a general sparse matrix A , our implementation of the randomized algorithm is based on the normalized power iterations [24]. Figure 1 shows the pseudocode of the algorithm. The input matrix $\hat{Q}$ represents the sampling and projection applied to the matrix A . Though there are several sampling and randomization schemes, for all the experiments in this paper, we used the Gaussian random vectors for $\hat{Q}$, for which extensive theoretical work has been established [14]. When the singular values of A decay slowly (which are typical for matrices from data-rich fields), we perform the power iterations to reduce the noise and improve the quality of the approximation. The basis vectors are orthonormalized in order to maintain the numerical stability during the iteration, and the projected matrix B is computed as a by-product of the orthogonalization process.

3.2 Updating Algorithm

To compute the projection basis for updating SVD, the updating algorithm [41] first orthogonalizes the new set of the columns D against the current left singular vectors U_k ,

$$\hat{D} := D - U_k(U_k^T D).$$

Then, the resulting vectors $\hat{D}$ are orthonormalized based on their QR factorization,

$$PR := \hat{D},$$

where P is the m -by- d orthonormal matrix (i.e., $P^T P = I$), and R is a d -by- d upper-triangular matrix.

The basis vectors are then given by

$$\hat{P} = [U_k, P] \quad \text{and} \quad \hat{Q} = \begin{bmatrix} V_k & 0 \\ 0 & I_d \end{bmatrix}, \quad (4)$$

where I_d is a d -by- d identity matrix, and the $(k + d)$ -by- $(k + d)$ projected matrix B is given by

$$\begin{aligned} B &\equiv \hat{P}^T [A_k, D] \hat{Q} \\ &= \begin{bmatrix} \Sigma_k & U_k^T D \\ 0 & R \end{bmatrix}. \end{aligned} \quad (5)$$

In practice, to reduce the prohibitively large amount of memory required to store the m -by- d dense vectors Q , the SVD is incrementally updated by adding a subset of the new columns D at a time. Though the memory requirement and the computational cost of the orthogonalization are reduced, all the columns of D are still orthogonalized against U_k . In addition, for each block of D , the corresponding SVD of the projected matrix must be computed, and the approximation U_k and V_k must be updated. As a result, the accumulated computational cost of the incremental updates could still be significant. It has also been reported that the incremental update can increase the runtime of the algorithm because the required computational kernels often obtain lower performance as they operate on smaller matrices [41, 34]. Finally, the quality of the approximation may degrade when the SVD is incrementally updated [33, 34].

3.3 Randomization Schemes to Update SVD

For updating SVD, we propose the following three schemes to generate the projection basis vectors $\hat{P}$ and $\hat{Q}$ based on the randomized algorithm, each of which performs the power iteration of Figure 1 on a different coefficient matrix:

- a) *Random-1*: perform the power iteration on the m -by- $(n + d)$ matrix,

$$\hat{A}_k = [A_k, D],$$

where $A_k = U_k \Sigma_k V_k^T$. This generates the pair of r basis vectors, $\hat{P}$ and $\hat{Q}$, that approximate the dominant singular vectors of the matrix $\hat{A}_k$. It has been shown [41] that many matrices from LSI have so-called approximate low-rank-plus-shift structures, and for this type of matrix (with a large enough k), the partial SVD of $\hat{A}_k$ accurately approximates the partial SVD of $\hat{A}$. Figure 2 shows the singular values of the SNP matrix used for population clustering studies in Section 4.2. This matrix also has this approximate low-rank-plus-shift structure, where the singular values decay quickly to a constant value. We refer to this sampling scheme as “Random-1.”

When the $k + d$ vectors $\hat{Q}$ in Eq. (4) are used as the starting vectors, after one iteration, Random-1 generates the same projection basis vectors, $\hat{P}$ and $\hat{Q}$, as the updating algorithm. Hence, Random-1 provides a more general framework, and we used it to

Figure 2: Singular values of the 116565-by-499 SNP matrix from Section 4.2.

study the effectiveness of the random sampling to compress the desired information into a small projection subspace (e.g., in our experiments, $r = k + \ell$ and $\ell \ll d$).

b) Random-2: apply the power iterations to the deflated matrix,

$$(I - U_k U_k^T)D,$$

and generate r left basis vectors P . We use the projection basis vectors $\hat{P}$ and $\hat{Q}$ defined by Eq. (4). The same projection subspace was used in [34], whereby instead of the block-wise power iteration, the column-wise Lanczos iteration was used to generate P . It has been shown that $\hat{Q}$ is the optimal right subspace for the left subspace $[U_d, P]$, and U_k is used to approximate U_d . We refer to this as “Random-2.”

c) Random-3: apply the power iterations to the same deflated matrix as Random-2, but let the right basis vectors $\hat{Q}$ be the $\hat{n}$ -by- $(k + r)$ matrix,

$$\hat{Q} = \begin{bmatrix} V_k & 0 \\ 0 & Q \end{bmatrix},$$

where Q contains the right basis vectors generated by the sampling. Since the right projection subspace of Random-2 is of dimension $k + d$, the cost of computing the SVD of B is significantly reduced using the basis vectors Q in the right projection subspace (i.e., $d \gg r$). This is called “Random-3.”⁴

4. CASE STUDIES

In Section 6, we analyze the computational and communication complexities of the proposed randomization schemes. We then study their performance on a hybrid CPU/GPU cluster in Section 7. In this section, prior to these studies, we demonstrate the potential of the randomized algorithm to compress the desired information into a small projection subspace with a small number of data accesses. To this end, we focus on a powerful statistical analysis tool, the principal component analysis (PCA) [4]. In PCA, a multi-dimensional dataset is projected onto a low-dimensional subspace given by the partial SVD such that related items are close to each other in the projected subspace. In the following two subsections, we examine two particular real-world applications of PCA, Latent Semantic Indexing (LSI) and population clustering.

⁴Though we use the Lanczos basis vectors P and Q in this paper, Random-2 and Random-3 can alternatively use the k dominant Ritz vectors of $(I - U_k U_k^T)D$ to generate the projection subspace.

Figure 4: Average 11-point interpolated precision for 5735-by-1033 MEDLINE matrix with 30 queries ($k = 50$).

4.1 Latent Semantic Indexing

For information retrieval by text mining [29], a variant of PCA, Latent Semantic Indexing (LSI) [8], has been shown to effectively address the ambiguity caused by the synonymy or polysemy, which are difficult to address using a traditional lexical-matching [21]. In order to study the effectiveness of the proposed randomized algorithm to update the SVD, in this section, we use the updated SVD for LSI. The test matrices are the term-document matrices generated using the Text to Matrix Generator (TMG)⁵ and the TREC dataset⁶, and are preprocessed using the `1xn.bpx` weighing scheme [20]. These are the standard test matrices and were used in the previous studies [41, 34].

Figure 3 compares the average 11-point interpolated precisions [20] after adding different numbers of documents from the CRANFIELD matrix. Specifically, we first computed the rank- k approximation of the first 700 documents by 20 power iterations of the randomized algorithm. Then, the table shows the average precision after new columns are added (e.g., under the column labeled “1000,” 300 documents were added). To recompute the partial SVD of $\hat{A}$, we performed 20 power iterations, while the randomized algorithm used the oversampling parameter set to be $\ell = k$ (i.e., $r = 2k$), and performed two iterations that access the matrix three times. Since the basis vectors $\hat{P}$ and $\hat{Q}$ approximate the ranges of $\hat{A}$ and $\hat{A}^T$, respectively, the randomized algorithm accesses the matrix at least twice. Then, they access the matrix one more time to compute the projected matrix B . We let the incremental update algorithm (Update-inc) add $k + \ell$ columns at a time such that it requires about the same amount of memory as the randomized algorithm. All the results were computed using one GPU (see Sections 5 for the description of our implementation). We have verified the precisions of recomputing SVD by comparing them against the precisions obtained using the dense SVD of the matrix $\hat{A}$ in MATLAB. We see that with only three data passes, the randomized algorithm obtained similar precisions as those of the updating algorithm. Figure 4 shows the similar results for the MEDLINE matrix. In some cases,

⁵<http://scgroup20.ceid.upatras.gr:8000/tmg>

⁶<http://ir.dcs.gla.ac.uk/resources>

Method	Total number of documents, $n + d$							
	700	800	900	1000	1100	1200	1300	1400
Recompute	35.6	40.8	40.2	45.3	44.9	44.0	43.6	41.7
Update	35.6	40.8	40.4	45.6	44.9	44.1	43.9	42.2
Update-inc	35.6	40.8	40.4	45.6	44.9	43.9	44.0	42.6
Random-1	35.6	40.0	40.4	45.6	44.5	44.1	43.6	41.8
Random-2	35.6	36.7	40.4	45.3	44.4	43.8	43.7	42.0
Random-3	35.6	36.7	40.4	45.3	44.4	43.6	43.4	41.7

(a) $k = 200$.

Method	Total number of documents, $n + d$							
	700	800	900	1000	1100	1200	1300	1400
Recompute	26.7	30.9	32.0	32.5	32.7	31.3	30.8	29.8
Update	26.7	29.8	30.1	30.7	31.5	30.7	30.4	29.7
Update-inc	26.7	29.8	30.1	30.6	30.9	30.1	29.8	29.5
Random-1	26.7	29.0	29.9	31.9	31.9	30.9	29.5	28.6
Random-2	26.7	29.6	29.6	30.0	31.0	30.1	30.0	29.7
Random-3	26.7	29.6	28.2	28.2	27.9	27.4	26.8	25.8

(b) $k = 50$.

Figure 3: Average 11-point interpolated precision for 6916-by-1400 CRANFIELD matrix with 225 queries, $n = 700$.

	Update-inc	Random-1
1	The World Is Not Enough	Mission to Mars
2	Mrs. Doubtfire	The World Is Not Enough
3	Mission: Impossible	Armageddon
4	Die Another Day	Crimson Tide
5	The 6th Day	Mission: Impossible
6	Mission to Mars	Die Another Day
7	The Mummy	Entrapment
8	Die Hard 2: Die Harder	Patriot Games
9	Charlie's Angels	Die Hard 2: Die Harder
10	The Santa Clause	Men of Honor

Figure 5: Query results for “Tomorrow Never Dies” on 3 GPUs.

the updating and randomized algorithms obtained higher precisions than recomputing the SVD, while the precisions of the incremental update slightly deteriorated at the end. Such phenomena were also reported in the previous studies [33, 41].

We have also conducted the same experiments using the Netflix matrix⁷, which stores the user scores of the movies, where the matrix columns and rows represent movies and users, respectively (i.e., 2,649,429 users and about 5,654 rankings per movie). We computed the rank-30 approximation of 5,000 movies by 20 power iterations of the randomized algorithm, then ran two iterations of Random-1 to update SVD with 5,000 new movies with the oversampling parameter set to be $\ell = 30$. For comparison, we incrementally updated the SVD by adding 60 movies at a time. Figure 5 shows the results for the query of “Tomorrow Never Dies.” We see that with only two iterations, Random-1 returned reasonable recommendations for the query. Though there are a couple of movies that seem to be irrelevant using the updating algorithm, these are popular movies, and it is reasonable to imagine that the users who liked “Tomorrow Never Dies” also liked these two movies.

4.2 Population Clustering

PCA has been successfully used to extract the underlying genetic structure of human populations [25, 27, 28]. To study the potential of the randomized algorithm, we used it to update the SVD, when a new population is added to the population dataset from the HapMap project⁸. Figure 6 shows the correlation coefficient of the resulting population cluster, which is computed using the k -mean algorithm of MATLAB in the low-dimensional subspace given by the dominant left singular vectors. We randomly filled in the missing data with either -1 , 0 , or 1 with the probabilities based on the available information for the SNP. We let Random-1 iterate twice, and with only the three data passes, Random-1 improved the clustering results, potentially reducing the number of times the SVD must be

⁷<http://netflixprize.com>

⁸<http://hapmap.ncbi.nlm.nih.gov>

	JPT+MEX	+ ASW	+ GIH	+ CEU
Recompute	1.00	1.00	1.00	0.97
No update	1.00	0.81	0.84	0.67
Update-inc	1.00	1.00	0.89	0.70
Random-1	1.00	0.95	0.92	0.86

Figure 6: Average correlation coefficients of population clustering based on the five dominant singular vectors, where 83 African ancestry in south west USA (ASW), 88 Gujarati Indian in Houston (GIH), and 165 European ancestry in Utah (CEU) were incrementally added to the 116,565 SNP matrix of 86 Japanese in Tokyo and 77 Mexican ancestry in Los Angeles, USA (JPT and MEX). Random-1 iterated twice with $\ell = k$.

recomputed.

5. HYBRID CPU/GPU IMPLEMENTATION

Since the computational cost of the randomized algorithm is dominated by the cost of generating the projection basis vectors, $\hat{P}$ and $\hat{Q}$, we accelerate this step using GPUs, while the SVD of the projected matrix B is redundantly computed by each MPI process on CPU. On a hybrid CPU/GPU cluster with multiple GPUs on each node, each MPI process could manage multiple GPUs on the node to combine or avoid the MPI communication to the GPUs on the same node. However in our previous studies [38], the overhead for each MPI process to manage multiple GPUs (e.g., sequentially launching GPU kernels on multiple GPUs) outweighed the benefit of avoiding the intra-node communication, for which many MPI implementations are optimized. Hence, for this paper, we use one MPI process to manage a single GPU.

The two main computational kernels of the randomized algorithms are the sparse-matrix dense-matrix multiply (SpMM) and the orthogonalization. In the following subsections, we describe our implementations of these two kernels on a hybrid CPU/GPU cluster.

5.1 Sparse Matrix Matrix Multiply

To update SVD by adding new columns D on a hybrid CPU/GPU cluster, our first implementation distributes both D and D^T among the GPUs in 1D block row formats. The basis vectors $\hat{P}$ and $\hat{Q}$ are then distributed in the same formats. Figure 7(a) illustrates the matrix and vector distributions for performing SpMM with D^T . Then, to perform SpMM, each GPU first exchanges the required non-local vector elements with its neighboring GPUs. This is done by first copying the required local elements from the GPU to the CPU, then performing the point-to-point communication among the neighbors using the non-blocking MPI (i.e., `MPI_Isend` and `MPI_Irecv`), and finally copying the non-local vector elements back to the GPU. Then, each GPU computes the local part of the next basis vectors

(a) 1DBR with neighborhood-collective before local SpMM. (b) 1DBC with all-reduce after local SpMM.

Figure 7: Illustration of matrix and vector distributions for SpMM with D^T . The submatrices distributed to the same GPU are colored in the same color. In Figure 7(a) or 7(b), the sparse matrix D^T is distributed either in 1D block row or block column (1DBR or 1DBC in short), respectively.

Figure 8: Number of vector elements needed for SpMM by a GPU, relative to the total number of elements in the global vector. The solid and dashed lines show the results with and without hypergraph partitioning, respectively. PaToH ran out of CPU memory partitioning among more than 12 GPUs.

using the CuSPARSE SpMM in the compressed sparse row (CSR) format. This was an efficient communication scheme in our previous studies to develop a linear solver [38], where the coefficient matrix arising from a scientific or engineering simulation is often sparse and structured, e.g., with three-dimensional embedding. Unfortunately, sparse matrices originating from the modern data sets such as social networks and/or commercial applications have irregular sparsity structures, and have wide ranges of nonzero counts per row. In fact, they often exhibit power-law distributions of nonzeros as they result from scale-free graphs. As a result, this point-to-point communication with all the neighbors at once could lead to a prohibitively large communication buffer (see Figure 8).

To reduce the size of the communication buffer required for SpMM, our second implementation is based on the point-to-point blocking MPI (i.e., MPI_Send and MPI_Recv), hence reusing the communication buffer. Unfortunately, this was not scalable. To alleviate the problem, we have tested the following collective communication schemes:

	Send/Recv	Bcast	Allgatherv
3 GPUs, 2 neighbors/GPU			
Time (ms)	7.5 (7.1)	6.6	2.5
Buffer (MB)	0.8	0.8	2.4
12 GPUs, 11 neighbors/GPU			
Time (ms)	229.5 (257.5)	84.1	36.0
Buffer (MB)	0.2	0.2	2.4

(a) $(n, d) = (5000, 5000)$ with $k = 30$.

	Send/Recv	Bcast	Allgatherv
3 GPUs, 2 neighbors/GPU			
Time (ms)	9.9 (10.0)	9.1	2.8
Buffer (MB)	1.6	1.6	4.8
12 GPUs, 11 neighbors/GPU			
Time (ms)	488.1 (552.9)	190.3	85.3
Buffer (MB)	0.4	0.4	4.8

(b) $(n, d) = (10000, 3000)$ with $k = 60$.

Figure 9: Communication statistics of SpMM with D for Netflix matrix. For Send/Recv, the times in parentheses are without hypergraph partitioning.

- **MPI_Bcast:** Each MPI process broadcasts its local elements. The buffer is reused to receive each message, which is then unpacked into the internal storage used by the solver. Even though this may perform unnecessary data exchanges, the communication time can be greatly reduced when the nonzeros of the matrix follows the power-law distribution.
- **MPI_Allgatherv:** With both broadcast and all-gatherv, each process sends its local vector elements, which are needed by at least one of its neighbors, to all the processes. Though this all-to-all approach requires the buffer to store the receiving messages from all the processes at once, it could obtain a significant speedup over the broadcast.
- **MPI_Alltoallv:** All MPI processes exchange the messages of different sizes with their neighbors at once. Hence, this approach may be more effective than the all-gatherv, when there are significant differences in the vector elements needed by the MPI processes. This is not the case for the matrix with their nonzeros following the power-law distribution (see Figure 8(a)). In addition, this requires storing the messages sent to a different neighbor separately.

Figure 9 shows the performance of SpMM with D using the above communication schemes for updating the Netflix matrix (see Sections 4 and 7 for our experimental setups). To update the SVD by adding the new columns D to A , we distributed the rows of D in the same format as A , while the rows of D^T were evenly distributed among the GPUs. For the point-to-point communication, we tested two partitioning schemes. In the first scheme, we used PaToH⁹ to distribute the rows of A and A^T based on a hypergraph algorithm [15]¹⁰. In the second scheme, we simply partitioned A and A^T such that each GPU has similar numbers of rows. For the collective communication, we used the second scheme which obtains the load balance. Due to the power-law distribution, using the hypergraph algorithm seems to have negligible effects on the communication cost. This is especially true on the hybrid cluster because we often do not require a large number of GPUs to obtain a desired performance due to the high computing power of the GPU. We clearly see that MPI_Allgatherv requires much less time.

⁹<http://bmi.osu.edu/umit/software.html#patoh>

¹⁰Our implementation using a graph partitioner of METIS (<http://glaros.dtc.umn.edu/gkhome/metis>) was often slower.

	Send/Recv	Bcast	Allgather	Allreduce
3 GPUs, 2 neighbors/GPU				
Time (ms)	312.9 (271.8)	239.7	145.9	2.9
Buffer (MB)	72.8	74.7	223.7	2.4
12 GPUs, 11 neighbors/GPU				
Time (ms)	1012.0 (977.1)	355.9	155.3	5.8
Buffer (MB)	15.2	18.9	226.3	2.4

(a) $(n, d) = (5000, 5000)$ with $k = 30$.

	Send/Recv	Bcast	Allgather	Allreduce
3 GPUs, 2 neighbors/GPU				
Time (ms)	572.3 (533.0)	543.6	260.7	3.8
Buffer (MB)	75.3	76.2	228.6	0.6
12 GPUs, 11 neighbors/GPU				
Time (ms)	1737.0 (1740.0)	688.9	264.4	6.7
Buffer (MB)	17.4	19.1	229.4	0.6

(b) $(n, d) = (10000, 3000)$ with $k = 60$.

Figure 10: Communication statistics of SpMM with D^T for Netflix matrix.

In this paper, we focus on the tall-skinny matrix D , which is the case in the applications we considered. To perform SpMM with D^T , our next implementation keeps the input and output vectors, $\hat{P}$ and $\hat{Q}$, in the 1D block row distribution, but uses D^T in the 1D block column (see Figure 7(b)). Since the columns of D^T are the same as the rows of D on each GPU, we do not need to separately store D^T and D .¹¹ In this implementation, each GPU first computes SpMM with its local parts of D^T and $\hat{P}$, and then copies the partial result to the CPU. Then, the MPI process computes the final result $\hat{Q}$ by a global all-reduce, and copies its local part back to the GPU. Hence, this requires each MPI process to store the global vectors $\hat{Q}$. However, when D^T has the power-law distribution, performing SpMM with D^T in the 1D block row requires each GPU to store the much longer global vectors $\hat{P}$ (see Figure 8(b), where the hypergraph partitioning reduced the median and minimum numbers of required vector elements, but not the maximum which is on the critical path). Figure 9 clearly demonstrates the advantage of this all-reduce communication. Furthermore, partitioning D^T in the 1D block column often led to a higher performance of SpMM on each GPU as the local submatrix becomes more square than tall-skinny.

Though other techniques may improve the performance of SpMM (e.g., sub-communicator with 2D distribution [5]), in this paper, we use MPI_Allgather and MPI_Allreduce for SpMM with D and D^T , respectively.¹² In other words, we perform the sparse computation on each GPU, but rely on the dense communication among them (which is effective for the matrix with the power-law distribution as shown above). In addition, we partition the matrix based on the natural matrix ordering, i.e., the second partitioning scheme above, which is effective for the collective communication. Though our MPI implementation may not be optimal, in this paper, we focus on developing an efficient algorithm for updating the SVD and studying the performance of this specific reference implementation, and not on developing an optimal communication scheme for a specific matrix structure on a specific hardware. In a real application, repartitioning/redistributing the data may not be feasible. Our solver only requires the result of SpMM with the local data. Then,

¹¹CuSPARSE SpMM in CSC uses atomic operations. To guarantee the reproducibility of the results, we explicitly stored D^T in the CSR format.

¹²Our implementation uses either the point-to-point or the collective communication depending on the average number of neighbors. For all the experiments in this paper, the collective communication was selected.

the final result can be computed through communication, which tends to be the bottleneck. Our objective of the paper is to develop an efficient algorithm with a small communication cost.

5.2 Orthogonalization Kernels

For our experiments in this paper, we used the classical Gram-Schmidt (CGS) [12] to orthogonalize a set of vectors against another set of vectors (block orthogonalization, or BOrth in short) and the Cholesky QR (CholQR) [32] to orthogonalize the set of vectors against each other. In our previous studies, these algorithms obtained great performance on multiple GPUs on a single compute node [35] or on a hybrid CPU/GPU cluster [38]. To orthonormalize $\hat{Q}$ by CholQR, each GPU first computes the block dot products of its local vectors $\hat{Q}^{(g)}$ (i.e., $G^{(g)} := \hat{Q}^{(g)T} \hat{Q}^{(g)}$), where $\hat{Q}^{(g)}$ is the local matrix of $\hat{Q}$ distributed to the g -th GPU. Then, the resulting matrix $G^{(d)}$ is copied to the CPU, and the MPI process computes the Gram matrix G via a global all-reduce (i.e., $G = \sum_{g=1}^{n_g} G^{(g)}$, where n_g is the number of available GPUs). Next, each MPI process redundantly computes the Cholesky factorization of the Gram matrix on the CPUs (i.e., $RR^T := G$), and copies the Cholesky factor R to its local GPU. Finally, each GPU orthogonalizes the local part of the basis vectors through a triangular solve (i.e., $Q^{(g)} := \hat{Q}^{(g)} R^{-1}$). Hence, CholQR requires only one global reduction between the GPUs, while most of the local computation is based on BLAS-3 kernels on the GPU.

When the matrix $\hat{Q}$ is ill-conditioned, CholQR can suffer from numerical instability [32]. In our experiments with the sampling algorithms, CholQR was stable with full reorthogonalization. The reliability of CholQR may be explained by the observation that the matrices used in our experiments had the approximate low-rank-plus-shift structure, but they were not ill-conditioned (see Section 3.3). On the other hand, the updating algorithm needs to orthogonalize the columns of D , some of which could be empty. To overcome the numerical difficulty, the updating algorithm uses the Singular Value QR (SVQR) [32], which computes the upper-triangular matrix R by first computing the SVD of the Gram matrix, $USU^T := G$, followed by the QR factorization of $\Sigma^{\frac{1}{2}}U^T$. When the matrix contains an empty columns, the Cholesky factorization of the Gram matrix will fail, while SVQR overcomes this numerical challenge. Compared to the Cholesky factorization, computing the SVD and QR factorization of the Gram matrix is computationally more expensive. However, the dimension of the Gram matrix is much smaller than that of the input matrix $\hat{P}$ or $\hat{Q}$ (i.e., $d \ll m$). Hence, SVQR performs about the same number of flops as CholQR, using the same BLAS-3 kernels, and requires only one global reduce. As a result, SVQR obtains similar performance as CholQR.

CGS for BOrth is implemented similarly, and is based on the matrix-matrix multiplies on each GPU. All of our orthogonalization kernels use the optimized dense GPU kernels developed earlier [35, 39].

6. ALGORITHM COMPLEXITIES

To analyze the computational costs of the proposed randomization schemes, we separately consider the following two stages of the algorithms: 1) generating the projection basis vectors $\hat{P}$ and $\hat{Q}$, and 2) computing SVD of the projected matrix B and generating the singular vectors $\hat{U}_k$ and $\hat{V}_k$.

Figure 11 lists the numbers of floating point operations (flops) required by the first stage, where dense matrix-matrix multiply (GEMM) is used for BOrth. When Random-1 generates ck basis vectors and iterates h times, it requires about $c(2h - 1) \times$ more flops for

Method	SpMM	GEMM	CholQR
Update	$2k \cdot \text{nnz}(D)$	$2mkd$	$2md^2$
Update-inc	$2k \cdot \text{nnz}(D)$	$2mkd$	$2m\hat{d}d$
Random-1	$4r \cdot \text{nnz}(D)$	$4(m+n)kr$	$2(m+n)r^2$
Random-2,3	$4r \cdot \text{nnz}(D)$	$4mkr$	$2(m+n)r^2$

Figure 11: Flop counts for first stage (generation of basis vectors). For the randomized algorithm, we show the flop counts for one iteration, where r is the number of basis vectors generated (e.g., in our experiments, $r = 2k$). For the updating algorithm to reorthogonalize $\hat{D}$ against U_k , the matrix becomes fully dense in SpMM (i.e., $\text{nnz}(D) = md$). Update-inc updates the partial SVD by incrementally adding $\hat{d}$ columns at a time.

Method	Compute SVD of B	Generate U_k and V_k
Update	$O((k+d)^3)$	$O((m+n)k(k+d))$
Update-inc	$O(\frac{d}{\hat{d}}(k+\hat{d})^3)$	$O(\frac{d}{\hat{d}}(m+n)k(k+\hat{d}))$
Random-1	$O(r^3)$	$O((m+n)kr)$
Random-2	$O((k+d)(k+r)^2)$	$O((m+n)k(k+r))$
Random-3	$O((k+r)^3)$	$O((m+n)k(k+r))$

Figure 12: Flop counts for second stage (updating partial SVD). Update-inc updates the partial SVD by incrementally adding $\hat{d}$ columns at a time.

SpMM than Update-inc. On the other hand, Update-inc performs about $\frac{d(\hat{d}+k)}{hck^2(2+c)} \times$ more flops for orthogonalizing these basis vectors. In our experiments, we let Random-1 iterate twice with the oversampling parameter set to be equal to the rank of the approximation (i.e., $h = 2$ and $c = 2$). Hence, Random-1 performs about $6 \times$ more flops for SpMM, while Update-inc requires about $\frac{d}{k} \frac{\hat{d}+k}{16k} \times$ more flops for the orthogonalization. Clearly, when a large number of columns must be added, the orthogonalization becomes the bottleneck in Update-inc (e.g., $\frac{d}{k} = O(10^2)$ in our performance studies). On the other hand, Random-1 can be computationally more efficient than Update-inc, but even on a small number of GPUs, its execution time is often dominated by SpMM. In the end, with $c = 2$ and $h = 2$, Random-1 performs more flops than Update-inc when each column of D has more than $\frac{m}{7} (\frac{\hat{d}+k}{k} - 16\frac{k}{\hat{d}})$ nonzeros in average. For instance, when $k = 100$, $\hat{d} = 200$, and $d = 1000$ or 5000, Random-1 performs more flops than Update-inc when D is more than 20% or 38% dense, respectively (e.g., Netflix matrix is about 1.1% dense). For a smaller k , the randomized algorithm has a greater computational advantage over the updating algorithm.

Compared to the matrix operation $U_k V_k^T$ performed by Random-1, there are about twice as many flops in the operation $U_k U_k^T$ of Random-2 or Random-3 (i.e., $n \ll m$). However, Table 11 shows that all the randomization schemes perform about the same number of flops in GEMM. This is because when Random-2 or Random-3 applies the matrix operation $D^T(I - U_k U_k^T)$ to the vectors $\hat{P}$, these vectors are already orthogonal to U_k . Hence, the deflation operation $I - U_k U_k^T$ needs to be applied only after SpMM with D . Nevertheless, to maintain the orthogonality of the basis vectors, in our experiments, we reorthogonalized $\hat{P}$ against U_k . Therefore, compared to Random-1, both Random-2 and Random-3 performed about twice as many flops for GEMM.

Table 12 shows the required flop counts for the second stage. The flop count of computing SVD of B with Random-2 depends

Method	#words	#messages
Update	$O(d(d+k))$	$O(1)$
Update-inc	$O(d(\hat{d}+k))$	$O(\frac{d}{h})$
Random-1,2,3	$O(((d+r)+g(D))rh)$	$O(h)$

Figure 13: Communication costs of updating and randomized algorithms, where $g(D_j)$ is the total number of the nonlocal vector elements needed for SpMM with the j -th block D_j in 1D block row distribution, and h is the number of power iterations performed by the randomized algorithm.

linearly on the number of columns in D , while they depend only on k and r with both Random-1 and Random-3. Since our implementation of the randomized algorithm let each MPI process redundantly compute the SVD of the projected matrix B , this serial bottleneck can become significant on a much smaller number of GPUs with Random-2 (i.e., $k, r \ll d$).

Table 13 shows the required communication costs of the updating and randomized algorithms, where the communication volume of SpMM with D is bounded by the number of columns in D (i.e., $g(D) \leq d$). Clearly, compared to the updating algorithm, the randomized algorithm could require a greater communication volume for SpMM. For example, in our experiments, we set both r and $\hat{d}$ to be $2k$, while $r = 2$. Hence, in those experiments, the randomized algorithm could have communicated three times more words than the updating algorithm. On the other hand, the randomized algorithm has a significantly lower communication latency cost, which can be the bottleneck on a modern computer (e.g., $\frac{d}{h\hat{d}} = O(100)$ in our experiments). Since the updating algorithm has a large computation cost for orthogonalization, on a small number of GPUs, its execution time is typically dominated by the computation. But, as the number of GPUs increases, the parallel scaling of the updating algorithm can be limited by its high latency cost. On the other hand, due to its computation efficiency, while the randomized algorithm is often much faster than the updating algorithm, its execution time can be affected by the communication cost even on a small number of GPUs.

7. PERFORMANCE RESULTS

We now compare the performance of the proposed randomization schemes with that of the updating algorithm on a hybrid CPU/GPU cluster. For our studies, we let the updating algorithm add $k + \ell$ columns at a time, while all the randomization schemes generate $k + \ell$ basis vectors, where $\ell = k$. Hence, all the algorithms require about the same amount of memory. Also, we orthogonalized both $\hat{P}$ and $\hat{Q}$ with full reorthogonalization. In some cases [31], the randomized algorithms may only need to orthogonalize $\hat{Q}$, reducing its orthogonalization cost significantly (i.e., $d \ll m$).

Figure 14 compares the performance of the randomization schemes. Compared to Random-1, Random-3 spends about twice as much time in GEMM due to the full re-orthogonalization used to maintain the orthogonality of the basis vectors $\hat{P}$ against U_k . Then, compared to Random-3, Random-2 spends more time in GEMM because it requires additional SpMM and GEMM to generate its projected matrix B (i.e., $U_k^T D$ and $U_k^T \hat{P}$). However, the main difference is in the time spent computing the SVD of the projected matrix B and generating the singular vectors. This is because the projection subspace of Random-2 is greater than that of Random-3, which is larger than that of Random-1. Since our implementation lets each MPI process redundantly compute the SVD, compared to Random-1 or Random-3, this serial bottleneck became significant

Method	n_g	SpMM(A^T/A)	GEMM	CholQR	SVD
Random-1	3	153 / 99	8	47	1.5+2.0
Random-2	3	382 / 99	19	47	6935+3.9
Random-3	3	153 / 99	22	46	3.6+3.0
Random-1	12	27 / 66	4.4	17	1.5+0.9
Random-2	12	65 / 65	7.9	16	7014+2.5
Random-3	12	27 / 65	8.9	16	3.8+1.3
Random-1	48	18 / 108	4.0	10	1.6+0.6
Random-2	48	35 / 107	5.4	11	7035+2.1
Random-3	48	19 / 108	6.6	10	3.9+0.6

(a) $(n, d) = (5000, 5000), k = 30$.

Method	n_g	SpMM(A^T/A)	GEMM	CholQR	SVD
Random-1	3	186 / 127	21	141	6.3+5.6
Random-2	3	466 / 127	57	140	746+9.4
Random-3	3	187 / 128	66	139	19+9.0
Random-1	12	36 / 130	8.2	42	6.5+1.9
Random-2	12	89 / 123	18	41	742+3.7
Random-3	12	37 / 125	21	41	19+2.8
Random-1	48	20 / 194	5.6	18	6.5+1.0
Random-2	48	41 / 194	8.3	18	755+2.2
Random-3	48	21 / 196	9.6	18	26+1.3

(b) $(n, d) = (10000, 3000), k = 60$.

Figure 14: Strong parallel scaling results in milliseconds for Netflix matrix. Under “SVD,” we show, separately, the time spent for SVD of B and generating singular vectors.

in Random-2 on a smaller number of GPUs.

Figure 15 compares the performance of the randomized algorithm with that of the updating algorithm. Clearly, the updating algorithm can spend significantly longer time in the orthogonalization, leading to a great speedup obtained by the randomized algorithm (i.e., the speedups of up to 14.1). At the same time, the speedup decreased on a larger number of GPUs. This is because the execution time of the randomized algorithm is dominated by SpMM, whose strong parallel scaling suffered from the increasing inter-GPU communication cost for this relatively small-scale matrix. On the other hand, the updating algorithm was still spending a significant amount of its execution time for the orthogonalization which was still compute intensive and scaled over the small number of the GPUs. On a larger number of GPUs, compared to the randomized algorithm, the updating algorithm is expected to suffer from the greater communication latency. Figure 16 visualize the performance comparison.

Figures 17 and 18 show the similar performance results for the document-document matrix used in a previous LSI study [40]. The matrix row contains 2,559,430 documents, and each column contains about 4,176 nonzero entries. The weak parallel scaling results, in particular, show the advantages of the randomized algorithm due to its ability to compress the desired information into a small projection subspace using a small number of data passes. For the updating algorithm, the accumulated cost of the SVDs of the projected matrices also became significant.

8. COMMUNICATION-AVOIDING IMPLEMENTATION

Our performance results on a hybrid CPU/GPU cluster demonstrated the potential of the randomized algorithm to obtain significant speedups over the updating algorithm. However, the execution time of the randomized algorithm is often dominated by SpMM, and due to the particular sparsity structure of the matrix, its parallel scalability can be quickly limited by the inter-GPU communication. In this section, we describe our effort to address this limitation based on the algorithm [19] that applies the matrix powers

Method	n_g	SpMM(A^T/A)	GEMM	TSQR	SVD	Total
Update-inc	3	95	491	2990	274/282	4212
Random-1	3	153 / 99	8	47	1.5+2.0	314
Update-inc	12	44	177	1010	275+99	1664
Random-1	12	27 / 66	4.4	17	1.5+0.9	118
Update-inc	48	60	118	563	276+64	1121
Random-1	48	18 / 108	4.0	10	1.6+0.6	145

(a) $(n, d) = (5000, 5000), k = 30$.

Method	n_g	SpMM(A^T/A)	GEMM	TSQR	SVD	Total
Update-inc	3	107	446	2513	455+148	3720
Random-1	3	186 / 127	21	141	6.3+5.6	492
Update-inc	12	28	137	921	452+51	1630
Random-1	12	36 / 138	8.2	42	6.5+1.9	234
Update-inc	48	29	7.5	546	454+26	1167
Random-1	48	20 / 194	5.6	18	6.5+1.0	243

(b) $(n, d) = (10000, 3000), k = 60$.

Figure 15: Strong parallel scaling results in milliseconds for Netflix matrix. Update-inc and Random-1 use SVQR and CholQR for tall-skinny QR (TSQR), respectively.

Figure 16: Strong parallel scaling results for Netflix matrix with $(n, d) = (5000, 5000)$ and $k = 30$.

to dense vectors in a communication-avoiding fashion, where the matrix is represented as a sparse matrix plus a low-rank matrix. While the algorithm can generate different types of basis vectors, we focus on generating the monomial basis vectors. In addition, we use Random-3 as our example, but the other randomization schemes can be extended in the same fashion. In fact, Random-2 and Random-3 generate the same basis vectors.

Random-3 is equivalent to the normalized power iteration applied to the following matrix $\tilde{A}_k$ with the starting vectors $\tilde{Q}$:

$$\tilde{A}_k = \begin{pmatrix} 0 & (I - U_k U_k^T) D \\ D^T & 0 \end{pmatrix} \quad \text{and} \quad \tilde{Q} = \begin{pmatrix} 0 \\ Q \end{pmatrix}.$$

This coefficient matrix $\tilde{A}_k$ can be split into the following sparse and

Method	SpMM(A^T/A)	GEMM	TSQR	SVD	Total
12 GPUs					
Update-inc	228.3	4,357	20,570	2,146+1,352	29,080
Random-1	284.8 / 192.6	24.7	172.8	4.6+6.4	691.5
48 GPUs					
Update-inc	177.5	1,266	5,454	2,161+438.4	9,856
Random-1	86.1 / 99.9	8.8	49.6	4.6+2.0	253.2
96 GPUs					
Update-inc	166.1	836.1	3,032	2,147+305.4	6,877
Random-1	70.7 / 94.6	6.7	30.2	4.5+1.2	212.7

(a) $k = 50$.

Method	SpMM(A^T/A)	GEMM	TSQR	SVD	Total
12 GPUs					
Update-inc	326.0	7,208	25,310	6,081+2,022	41,320
Random-1	574.8 / 377.3	76.6	427.3	20.5+22.3	1,500
48 GPUs					
Update-inc	108.7	1,946	6,445	6,059+588.1	15,420
Random-1	162.2 / 201.3	22.9	112.5	26.4+6.1	535.3
96 GPUs					
Update-inc	77.8	1,110	3,494	6,047+351.3	11,350
Random-1	119.3 / 183.0	14.0	62.5	23.8+0.3	411.4

(b) $k = 100$.

Figure 17: Strong parallel scaling in milliseconds for Inktomi matrix with $(n, d) = (30000, 20000)$.

Figure 18: Weak parallel scaling results for Inktomi matrix, with the fixed parameters $(m, k) = (2559430, 50)$ and varied parameters $(n, d) = (n_g/12) \times (7500, 5000)$.

low-rank matrices,

$$\begin{aligned} \tilde{A}_k &= \begin{pmatrix} 0 & D \\ D^T & 0 \end{pmatrix} + \begin{pmatrix} 0 & U\hat{U}^T \\ 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & D \\ D^T & 0 \end{pmatrix} + \begin{pmatrix} U \\ 0 \end{pmatrix} \begin{pmatrix} 0 & \hat{U}^T \end{pmatrix} \quad (6) \\ &= \begin{pmatrix} 0 & D \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} U & 0 \\ 0 & I_d \end{pmatrix} \begin{pmatrix} 0 & \hat{U}^T \\ D^T & 0 \end{pmatrix}, \quad (7) \end{aligned}$$

where $\hat{U} = -D^T U$, and the low-rank matrices are of ranks k and $k + d$ in (6) and (7), respectively. The matrix split (7) is motivated by our implementation which computes SpMM with D^T by the local SpMM on each GPU, followed by the global reduce.

Figure 19 specializes the matrix powers kernel (MPK) [19] for applying s matrix powers based on the matrix split (7). In this implementation, each MPI process redundantly generates the right basis vectors F_j by applying the power iteration on the explicitly-formed Gram matrix. After the right basis vectors are generated, the local parts of the corresponding left basis vectors P are generated by the power iteration without further communication. Since the matrix D is tall-skinny (i.e., $d \ll m$), the generation of the right basis vectors requires much less computation than that needed to generate the left basis vectors. With this implementation, regardless of the sparsity structure of D , an arbitrary number of power iterations can be applied after the initial communication to generate the starting vectors E_1 and F_1 (i.e., the same communication

Figure 19: Matrix powers kernel with the matrix split (7) with initial vectors Q and step size s . Local computation is redundantly performed by each MPI process on the CPU, except the multiply with $D^T D$ which is on the GPU.

cost as the single iteration of Random-3). However, to avoid the communication, each GPU not only requires the Gram matrix, the computation of which requires a global reduce, but it also requires both the local and ghost elements of Q .

Figure 20 compares the performance of MPK with that of the standard SpMM for updating the Netflix matrix. The execution time is reduced not only because the MPI time is reduced but also because SpMM with D^T is replaced with GEMM with $D^T D$, whose dimension is much smaller than that of D^T . Since the execution time of the randomized algorithm is often dominated by that of SpMM, these results demonstrate the potential of MPK to improve the performance of the randomized algorithm shown in Section 7. Figure 21 summarizes the computational and communication costs of the implementations.

In these experiments, we have generated s matrix powers. Since the randomized algorithm requires only the last matrix power, the time to generate P can be significantly reduced. In other words, in Figure 19(a), the randomized algorithm needs to perform only the last iteration of the for-loop to generate the basis vectors P and Q (i.e., $j = s$).

Method	SpMM	GEMM	Method	#messages	#words
SpMM	$4s(\text{nnz}(D^{(g)}))r$	$4s(km/n_g)r$	SpMM	$4s$	$s(g(D) + d)r$
MPK	$4s(\text{nnz}(D^{(g)}))r + 2(s-1)(\text{nnz}(D^{(g)}) + \text{nnz}(D^T D))r$	$2s(km/n_g + 2kd)r$	MPK	2	$(g(D) + d)r$

(a) Flop count for g -th GPU.

(b) Total communication costs.

Figure 21: Computational and communication costs to apply s matrix powers. MPK requires the additional set-up costs to form the auxiliary matrix $\hat{U}$ and the Gram matrix $D^T D$.

n_g	SpMM(D^T/D)	GEMM	MPI(D^T/D)	Total
3	15.1/5.0	0.9	0.29 / 0.15	21.0
12	2.5/1.7	0.3	0.46 / 0.20	4.49

(a) SpMM.

n_g	s	SpMM(D^T/D)	Local	Generate P	MPI(D^T/D)	Total
3	1	15.24 / 4.93	0.2	0.2	0.29 / 0.13	20.5
3	2	7.65 / 2.46	0.3	2.5	0.16 / 0.06	12.9
3	4	3.82 / 1.23	0.4	3.7	0.08 / 0.03	9.1
12	1	2.42 / 1.66	0.2	0.1	0.46 / 0.20	4.54
12	2	1.25 / 0.88	0.3	0.7	0.24 / 0.13	3.14
12	4	0.66 / 0.45	0.4	1.0	0.15 / 0.08	2.47

(b) MPK.

Figure 20: 100 power iterations in seconds, for Netflix matrix with $(n, d) = (5000, 5000)$.

9. CONCLUSION

We studied the randomized algorithm to update the partial singular value decomposition (SVD). Our case-studies showed the potential of such an algorithm to obtain the desired accuracy of the updated SVD with a small number of data accesses, reducing both the computational and communication costs from those of the state-of-the-art updating algorithm. Our performance results on a hybrid CPU/GPU cluster demonstrated that a significant speedup can be obtained using the randomized algorithm. The computational efficiency of the randomized algorithm also implies that the algorithm may require only a small number of GPUs to obtain a desired performance.

We also presented our effort to reduce the communication costs of the randomized algorithm. To maintain the numerical stability of the communication-avoiding implementation, we are currently studying different orthogonalization schemes such as the one-sided orthogonalization to orthogonalize only the right-singular vectors during the power iteration [31]. We are also studying block variants of orthogonalization process [1, 17] for the randomized algorithm. Compared to CholQR, these block variants have greater communication latency costs. However, by using a higher precision arithmetic only at critical parts of the algorithms, we can improve the numerical stability of the orthogonalization processes with a relatively small computational overhead [36]. In addition, to improve the performance of the solver, we are working to integrate more efficient computational kernels (e.g., a parallel SVD of the projected matrix, and an extension of the batched GEMM used in CholQR to a batched sparse GEMM for forming the Gram matrix to setup MPK, or to a batched SpMM with a transpose of a tall-skinny matrix). We focused on the performance of the algorithms on a hybrid CPU/GPU cluster, but we plan to investigate the performance on other architectures and with larger-scale data sets. For instance, compared to the hybrid cluster, the costs of SpMM relative to the orthogonalization may be smaller on a CPU cluster, making the randomized algorithm more attractive. We are also conducting the numerical analysis of the randomization schemes, and examining

the extension of the current studies to solve other type of updating problems and also to downdate the SVD.

Acknowledgments

This research was supported in part by the U.S. Department of Energy Office of Science under Award Numbers DE-FG0213ER26137 and DE-SC0010042, and the U.S. National Science Foundation under Award Number 1339822, and Intel Corporation. This work was funded in part by the Russian Scientific Foundation, Agreement N14-11-00190. We thank Eugen Vecharynski, Nicholas Knight, and Erin Carson and the members of the DOE EASIR project for helpful discussions, Theo Mary and Osni Marques for providing some of our datasets, and Endo Toshio and Satoshi Matsuoka for providing the access to the Tsubame Computer.

10. REFERENCES

- [1] J. Barlow and A. Smoktunowicz. Reorthogonalized block classical Gram-Schmidt. *Numerische Mathematik*, 123:395–423, 2013.
- [2] M. Berry. Large scale singular value computations. *International Journal of Supercomputer Applications*, 6:13–49, 1992.
- [3] M. Berry, S. Dumais, and G. O’Brien. Using linear algebra for intelligent information retrieval. *SIAM Rev.*, 37:573–595, 1995.
- [4] C. Bishop. *Pattern recognition and machine learning*. Springer, New York, 2006.
- [5] E. Boman, K. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 50:1–50:12, 2013.
- [6] A. Buluc and J. Gilbert. New ideas in sparse matrix-matrix multiplication. In *Graph Algorithms in the language of linear algebra*, chapter 14. Society for Industrial and Applied Mathematics, 2011.
- [7] Committee on the Analysis of Massive Data, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences and Their Applications, Division on Engineering and Physical Sciences, and National Research Council. *Frontiers in Massive Data Analysis*. The National Academies Press, 2013.
- [8] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. Amer. Soc. Info. Sci.*, 41:391–407, 1990.
- [9] DOE Office of Science. Synergistic challenges in data-intensive science and exascale computing, 2013. DOE Advanced Scientific Computing Advisory Committee (ASCAC) Data Subcommittee Report.
- [10] S. Fuller and L. Millett. Future of computing performance: Game over or next level? The National Academies Press, 2011.

- [11] G. Golub, F. Luk, and M. Overton. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Trans. Math. Softw.*, 7:149–169, 1981.
- [12] G. Golub and C. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 4rd edition, 2012.
- [13] S. Graham, M. Snir, and C. Patterson. Getting up to speed: The future of supercomputing. The National Academies Press, 2004.
- [14] N. Halko, P. Martinsson, and J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53:217–288, 2011.
- [15] B. Hendrickson and T. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrix for parallel processing. *SIAM J. Sci. Comput.*, 21:2048–2072, 2006.
- [16] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, Berkeley, 2010.
- [17] W. Jalby and B. Philippe. Stability analysis and improvement of the block Gram-Schmidt algorithm. *SIAM J. Sci. Stat. Comput.*, 12:1058–1073, 1991.
- [18] I. Karasalo. Estimating the covariance matrix by signal subspace averaging. *IEEE Trans. Acoust., Speech, Signal Processing*, ASSP-34:8–12, Feb. 1986.
- [19] N. Knight, E. Carson, and J. Demmel. Exploiting data sparsity in parallel matrix powers computations. In *Parallel Processing and Applied Mathematics*, volume 8384 of *Lecture Notes in Computer Science*, pages 15–25. Springer Berlin Heidelberg, 2014.
- [20] T. Kolda and D. O’Leary. A semidiscrete matrix decomposition for latent semantic indexing information retrieval. *ACM Trans. Inf. Syst.*, 16:322–346, 1998.
- [21] R. Krovetz and W. B. Croft. Lexical ambiguity and information retrieval. *ACM Trans. Inf. Syst.*, 10:115–141, 1992.
- [22] D. Laney. 3D data management: controlling data volume, velocity, and variety. Application Delivery Strategies by META Group Inc, 2001.
- [23] M. Mahoney. Randomized algorithms for matrices and data. *Found. Trends Mach. Learn.*, 3:123–224, 2011.
- [24] P. Martinsson, A. Szlam, and M. Tygert. Normalized power iterations for the computation of SVD. In *Proceedings of the Neural and Information Processing Systems (NIPS) Workshop on Low-Rank Methods for Large-Scale Machine Learning*, 2010.
- [25] P. Menozzi, A. Piazza, and L. C.-Sforza. Synthetic maps of human gene frequencies in Europeans. *Science*, 201:786–792, 1978.
- [26] P. Paschou, E. Ziv, E. Burchard, S. Choudhry, W. R.-Cintron, M. Mahoney, and P. Drineas. PCA-correlated SNPs for structure identification in worldwide human populations. *PLoS Genetics*, 3:1672–1686, 2007.
- [27] N. Patterson, A. Price, and D. Reich. Population structure and eigenanalysis. *PLoS Genet.*, e190, 2006.
- [28] A. Price, N. Patterson, R. Plenge, M. Weinblatt, N. Shadick, and D. Reich. Principal components analysis corrects for stratification in genome-wide association studies. *Nat. Genet.*, 38:904–909, 2006.
- [29] G. Salton and M. McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, 1983.
- [30] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 158–167, 2000.
- [31] H. Simon and H. Zha. Low-rank matrix approximation using the Lanczos bidiagonalization process with applications. *SIAM J. Sci. Comput.*, 21:2257–2274, 2000.
- [32] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23:2165–2182, 2002.
- [33] J. Tougas and R. Spiteri. Updating the partial singular value decomposition in latent semantic indexing. *Comput. Statist. Data Anal.*, 52:174–183, 2007.
- [34] E. Vecharynski and Y. Saad. Fast updating algorithms for latent semantic indexing. *SIAM J. Matrix Anal. Appl.*, 35:1105–1131, 2014.
- [35] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. Improving the performance of CA-GMRES on multicores with multiple GPUs. In *Proceedings of the IEEE International Parallel and Distributed Symposium (IPDPS)*, pages 382–391, 2014.
- [36] I. Yamazaki, J. Barlow, S. Tomov, J. Kurzak, and J. Dongarra. Mixed-precision block Gram Schmidt orthogonalization procedures. 2015. Submitted to Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems.
- [37] I. Yamazaki, T. Mary, J. Kurzak, and S. Tomov. Access-averse framework for computing low-rank matrix approximations. In *Proceedings of the international workshop on high performance big graph data management, analysis, and minig*, pages 70–77, 2014.
- [38] I. Yamazaki, S. Rajamanickam, E. Boman, M. Hoemmen, M. Heroux, and S. Tomov. Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC)*, pages 933–944, 2014.
- [39] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra. Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs. In *Proceedings of the international meeting on high-performance computing for computational science (VECPAR)*, pages 17–30, 2014.
- [40] H. Zha, O. Marques, and H. Simon. Large-scale SVD and subspace-based methods for information retrieval. *Lecture Notes in Computer Science*, 1457:29–42, 1998.
- [41] H. Zha and H. Simon. On updating problems in latent semantic indexing. *SIAM J. Sci. Comput.*, 21:782–791, 2006.