RANDOMIZED SINGULAR VALUE DECOMPOSITION (RSVD)

A Parallel randomized-SVD Decomposition with its Application in image compression

Seyedehyekta Kamaneh

Student No. 10959155

January 2024

Contents

1

Introduction

Singular value decomposition has its roots in linear algebra, dating back to the late 19th century with contributions from mathematicians like Eugenio Beltrami and Camille Jordan. The modern formulation and algorithm for SVD were established in the mid-20th century by Golub and Kahan. However, the high computational cost associated with traditional SVD methods led to the exploration of randomized approaches.

In particular, the concept of randomized methods emerged as a powerful solution to construct low-dimensional subspaces capturing essential spectral information. Halko, Martinsson, and Tropp (2009) argue that randomized algorithms provide a robust tool for constructing approximate matrix factorizations. This project draws heavily from their work to formulate and address the challenges associated with the implementation of randomized Singular Value Decomposition

2

Problem Formulation

In this section, we detail the mathematical foundations and algorithms that form the basis of our implementation. The formulation encompasses three key components: the randomized-SVD (rSVD) algorithm, the SVD with Powermethod, and QR Factorization. Each of these elements plays a vital role in achieving the objectives of our project.

2.1 RSVD as a Low-rank Approximation

The list of commonly employed matrix decompositions comprises the pivoted QR factorization, the eigenvalue decomposition, and the singular value decomposition (SVD). These methods reveal the numerical range of a matrix. Frequently, truncated variants of these factorizations are employed to represent a low-rank approximation of a given matrix.

$$A_{m\times n} \approx B_{m\times k} \cdot C_{k\times n} \tag{2.1}$$

The inner dimension k is sometimes called the numerical rank of the matrix. When the numerical rank is much smaller than either dimension m or n, a factorization such as (2.1) allows the matrix to be stored inexpensively and to be multiplied rapidly with vectors or other matrices.

The task of computing a low-rank approximation to a given matrix can be split naturally into two computational stages. The first is to construct a low-dimensional subspace that captures the action of the matrix. The second is to restrict the matrix to the subspace and then compute a standard factorization (QR, SVD, etc.) of the reduced matrix (Halko et al., 2011).

Given a matrix $A \in \mathbb{R}^{m\times n}$, the randomized SVD aims to approximate the singular value decomposition as follows:

$$A \approx U\Sigma V^T \tag{2.2}$$

U : Left singular vector matrix of size $m \times k$,

Σ : Diagonal matrix of singular values of size $k \times k$,

V^T : Transposed right singular vector matrix of size $k \times n$.

The randomized SVD proceeds as follows:

Randomized Sampling: Multiply A by a randomly generated matrix $\Omega \in \mathbb{R}^{n \times k}$, where k is a user-specified parameter:

$$Y = A\Omega \tag{2.3}$$

QR Factorization: Perform QR factorization on Y to obtain matrices Q and R, where Q is an $m \times k$ matrix with orthonormal columns.

$$Y = QR \tag{2.4}$$

Perform SVD: Compute the SVD of the product $Q^T A$, denoted as

$$Q^T AV = \tilde{U}\Sigma V^T \tag{2.5}$$

where $\tilde{U}$ is an $k \times k$ matrix, Σ is a $k \times k$ diagonal matrix, and V^T is a $k \times n$ matrix.

$$B = \tilde{U}\Sigma V^T \tag{2.6}$$

Approximation: The approximation of the original matrix A is given by

$$A = Q\tilde{U}\Sigma V^T \tag{2.7}$$

Algorithm (1) shows the steps to be done to perform the rSVD.

Algorithm 1: Prototype for Randomized-SVD

Data: Given a $m \times n$ matrix A, a target number k of singular vectors, and an exponent q (say q=1 or q=2)

Stage A:

Generate an $n \times 2k$ Gaussian test matrix Ω

Form $Y = (AA)^q A\Omega$ by multiplying alternately with A and A^*

Construct a matrix Q whose columns form an orthonormal basis for the range of Y

Stage B:

Form $B = Q^* A$

Compute an SVD of the small matrix: $B = \tilde{U}\Sigma V^*$

Set $U = Q\tilde{U}$

2.2 SVD with Powermethod

The Singular Value Decomposition (SVD) is a fundamental matrix factorization technique that decomposes a matrix into the product of three matrices: $A = U\Sigma V^T$, where U and V are orthogonal matrices, and Σ is a diagonal matrix containing the singular values of A.

The power method is an iterative numerical algorithm that can be employed to compute the dominant singular value and corresponding singular vector of a given matrix. The power method is particularly useful for large matrices when direct computation of the SVD might be computationally expensive (n.d.).

Algorithm (2) outlines the power method utilized to compute the initial set of singular vectors. Subsequently, a total of $min(m, n)$ steps, involving a combination of deflation and the power method, are employed to attain the complete Singular Value Decomposition (algorithm (3)).

Algorithm 2: The Power Iteration to get the singular vectors

Generate x such that $x^{(i)} \sim \mathcal{N}(0, 1)$.
$s \leftarrow \frac{\log\left(\frac{4\log(2n/\delta)}{\epsilon\delta}\right)}{2\lambda}$.
for i *in* $[1, \ldots, s]$ **do**
    $x \leftarrow A^T.Ax_{i-1}$.
$v_1 \leftarrow \frac{x_i}{\|x_i\|_2}$.
$\sigma_1 \leftarrow \|Av_1\|_2$.
$u_1 \leftarrow \frac{Av_1}{\sigma^{(i)}}$.
Return (σ_1, u_1, v_1).

Algorithm 3: SVD Computation using Power Method

for $i = 0, 1, \ldots, min(m, n)$ **do**
    $u, \sigma(i), v \leftarrow$ powerIteration(A)
    $A \leftarrow A - \sigma(i) \cdot u \cdot v^\top$

2.3 QR Factorization

QR factorization decomposes a matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R, i.e., $A = QR$. The orthogonal matrix Q has orthonormal columns, and R is an upper triangular matrix. This factorization is highly valuable for solving linear systems, eigenvalue problems, and least squares problems. It plays a pivotal role in various applications, offering numerical stability and computational efficiency (n.d.).

In the context of the randomized Singular Value Decomposition (rSVD), QR factorization is a crucial component in the construction of a low-dimensional subspace. The rSVD algorithm involves multiplying the original matrix A by a randomly generated matrix Ω (Randomized Sampling) and then performing QR factorization on the resulting matrix $Y = A\Omega$. The orthogonal matrix Q obtained from the QR factorization forms the basis for the low-dimensional subspace capturing essential spectral information.

2.3.1 Givens Rotations

A Givens rotation in the context of a 2D plane (or subspace) is a rotation matrix G that transforms a vector (x, y) to a new vector $(r, 0)$, introducing a zero in the second component.

The Givens rotation matrix G is given by:

$$G = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$$

where c and s are the cosine and sine of the rotation angle θ, respectively.

The rotation angle θ is determined by the elements a and b of the vector (x, y) that we want to rotate. The values of c and s are computed as follows:

$$c = \frac{a}{r}, \quad s = \frac{b}{r}$$

where $r = \sqrt{a^2 + b^2}$.

2.3.2 Givens Rotations in QR Decomposition

The QR decomposition using Givens rotations involves systematically introducing zeros into the lower triangular part of the matrix A. The idea is to perform a series of Givens rotations on pairs of elements in A, eliminating the entries below the main diagonal (Tapia-Romero et al., 2020).

Consider a square matrix A of size $m \times m$. The Givens rotation is applied to the submatrix $A(k : k+1, j : j+1)$ where $k < j$, introducing zeros below the element $A(k, j)$.

The Givens rotation matrix $G^{(k,j)}$ is defined as:

$$G^{(k,j)} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$$

where c and s are computed to satisfy:

$$c = \frac{A(k, j)}{\sqrt{A(k, j)^2 + A(j, j)^2}}, \quad s = \frac{A(j, j)}{\sqrt{A(k, j)^2 + A(j, j)^2}}$$

The Givens rotation is applied to the submatrix $A(k : k + 1, j : j + 1)$ as follows:

$$A' = G^{(k,j)T} \cdot A$$

The resulting matrix A' has zeros below the element $A(k, j)$.

Iterate this process for each pair $k < j$ to transform the entire matrix into upper triangular form.

$$A_1 = (G^{(1,2)})^T \cdot A, \quad A_2 = (G^{(1,3)})^T \cdot A_1, \quad \ldots, \quad A_{m-1} = (G^{(1,m)})^T \cdot A_{m-2}$$

The orthogonal matrix Q is the product of the transposes of the Givens rotation matrices:

$$Q = (G^{(1,2)})^T \cdot (G^{(1,3)})^T \cdot \ldots \cdot (G^{(1,m)})^T$$

The upper triangular matrix R is the last transformed matrix A_{m-1}.

3

Implementation

With a solid understanding of the mathematical foundations, we transition to the practical aspects of our project. In this section, we detail how we translated the formulated methodologies into a working C++ implementation. From data structures and input handling to the intricacies of the randomized SVD algorithm and the incremental approach, we provide a comprehensive overview of our implementation strategy.

3.1 Data Structure

In our C++ implementation, we utilized the Eigen library to handle the storage of matrices and vectors efficiently. This choice of data structures aimes to balance memory efficiency and computational speed while leveraging the features provided by the Eigen library. This library optimizes memory access by organizing matrices in a contiguous block of memory. With a column-major order as default, Eigen uses a one-dimensional array to represent the matrix, allowing for efficient memory access. However, in this project, Eigen is only used as a means of storing and accessing matrix cells effectively and most of the matrix operations are being manually implemented.

3.2 Matrix Operations

In this subsection, the focus is more on the technical aspect of the project, particularly concentrating on the parallelization of matrix operations. Recognizing the significance of efficient numerical computations, we have employed the Message Passing Interface (MPI) library to distribute and parallelize all matrix operations. This strategic use of MPI not only optimizes computational speed but also facilitates the handling of large-scale matrices that may exceed the memory capacity of a single processor. We recall the most important matrix operations in our project as matrix-vector multiplication, matrix-matrix multiplication, and matrix transposition.

3.2.1 Matrix-Vector Multiplication

As seen in algorithm (2) the power method consists of iteratively multiplying the matrix to be decomposed to a normal vector. Therefore this operation is pivotal in our

implementation of the Singular Value Decomposition. Through MPI, we distribute the rows of the matrix across multiple processors, harnessing parallel processing to perform simultaneous computations. This approach not only accelerates the matrix-vector multiplication but also enhances scalability, making our implementation well-suited for matrices of varying sizes.

The adopted strategy involves partitioning rows among various processors. Each processor is responsible for the multiplication of its assigned local rows, denoted as $A[i:j]$, by the vector b. Once all processors complete their computations, the results are gathered by the root processor and subsequently distributed back to all processors. To illustrate this process with three processors, Figure 3.1 provides a visual representation. Algorithm (4) shows the specific tasks assigned to each processor.

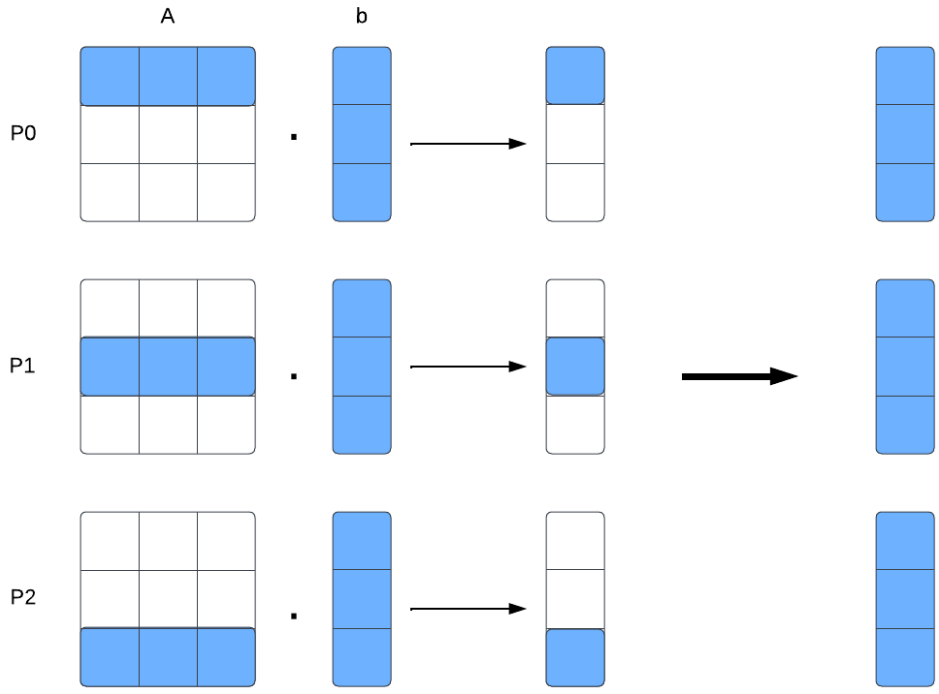


Figure 3.1: *Illustration of the*

Algorithm 4: Distributed Matrix-Vector Multiplication

Data: Given a matrix A of size mxn and a vector b of size n

$h \leftarrow$ number of local rows

for $i = 0, 1, \ldots, h$ **do**

  compute $Row_i(\mathbf{A})$·b

Broadcast results to all processors

3.2.2 Matrix-Matrix Multiplication

The computational intensity of the first stage in the rSVD algorithm, as highlighted in (2.1), is primarily attributed to multiple matrix-matrix multiplications involving large matrices. Given the computational expense associated with this stage, optimizing an efficient matrix multiplication scheme becomes of great importance.

To perform the parallel matrix-matrix multiplication, a technique similar to the previously explained matrix-vector multiplication is employed, with the distinction that the matrix partitioning occurs along columns, specifically for the second matrix. As the matrix entries are stored in a columns-major order in Eigen this choice helps with maintaining the locality of data. As for the first matrix we process each row one by one sequentially. In essence, if we consider a matrix A of dimensions $m \times n$ and another matrix B of dimensions $n \times k$, this methods involves m iterations of vector-matrix multiplication. Notably, the matrices involved in these multiplications are of reduced sizes, with each having dimensions $\frac{k}{p}$, where p represents the number of processors.

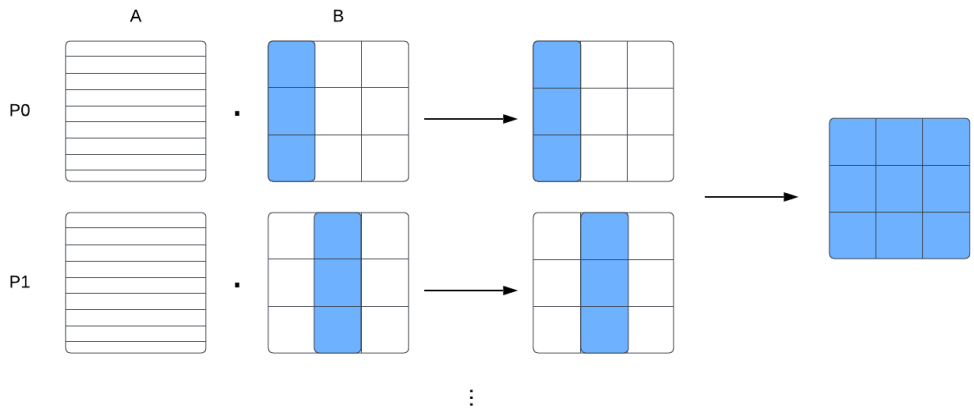


Figure 3.2: *Illustration of the*

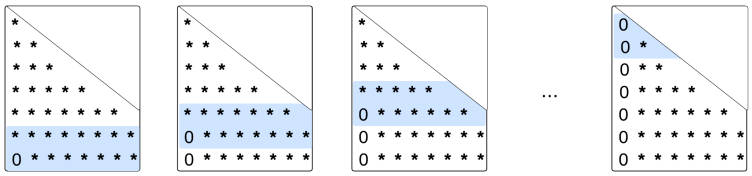
Algorithm 5: Distributed Matrix-Matrix Multiplication

Data: Given a matrix A of size mxn and a matrix B of size nxk

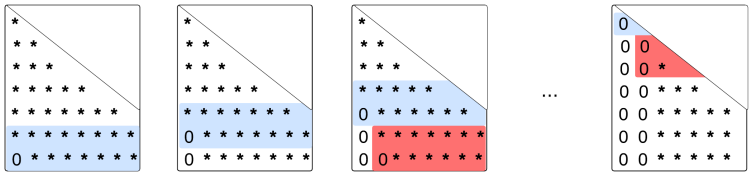
1 $h \leftarrow$ number of local rows
2 **for** $i = 0, 1, \ldots, m$ **do**
3 **for** $j = 0, 1, \ldots, h$ **do**
4 compute $Row_i(\mathbf{A}) \cdot Col_j(\mathbf{B_{local}})$
5 Broadcast results to all processors

3.3 Parallel QR Factorization

When applying Givens rotation to a matrix A to annihilate the a_{ij} element, the rotation matrix affects two rows of the matrix, rows $i-1$ and i. The result of the rotation makes the jth element of row i zero. It is possible to parallelize the computation of the columns of rows $i-1$ and i, because they are computations that do not generate dependencies between the computations of the columns (Andrew et al., n.d.).



(a) *Serial version*



(b) *Parallel version*

Figure 3.3: *Main Caption*

Figure 3.3(a) illustrates the sequential execution of Givens rotations to annihilate one column in a matrix. As shown, after the initial two iterations, there is no dependency between $a_{i,0}$ and $a_{m-1,1}$ allowing these two operations to proceed concurrently and eliminate two columns simultaneously. Figure 3.3(b) further demonstrates this concurrency when two processors collaborate to annihilate consecutive columns.

In our MPI implementation, the columns of the matrix A are distributed among processors and a latter processor can start performing rotations only if the former processor has finished its last second row of the last column, otherwise, the dependencies don't leave room for parallelization. After each iteration processors broadcast the new values of the rows they changed and synchronized with each other. Obviously, this approach wouldn't give a high parallelization result unless the number of processors is as great or at least close to the number of columns of the matrix.

4

Results

In this section, we present the outcomes and performance metrics of our implemented algorithms. Initially, we highlight the speed-ups achieved through the parallel implementation then, we shift our focus to the accuracy and applicability of the rSVD algorithm.

Performance and Speed-up

Table (4.1) provides an overview of CPU time during matrix-vector multiplication across various implementations. The multiplication is assumed to be on a square matrix of size n and a vector of size n. Three implementations were performed; one using Eigen's internal multiplication, one a serial naive multiplication without any trace of parallel processing, and finally the distributed version tested with 1 to 4 processors. Observing the results, it becomes evident that for smaller data sizes, the communication overhead outweighs its benefits. However, as the data size increases, the distributed version showcases a substantial speed-up of 3.85 when executed on 4 processors.

Table (4.2) follows the same structure focusing on matrix-matrix multiplication. In contrast to the matrix-vector multiplication scenario, this time, significant speed-up is evident right from the outset. As the size of data grows to 400×400, the speed-up reaches 3.03, running on 4 processors. This outcome underscores the scalability and efficiency achieved through the parallel implementation of matrix-matrix multiplication.

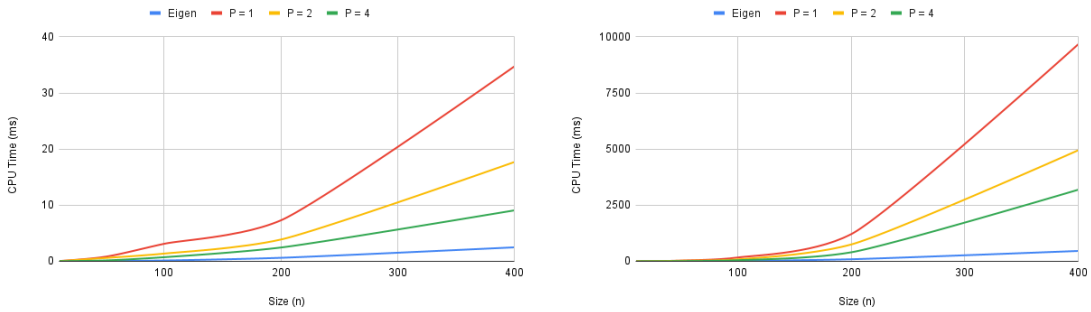
Table 4.1: *CPU time (ms) Matrix Vector multiplication*

Size (n)	Eigen	Serial Manual	MPI P = 1	MPI P = 2	MPI P = 3	MPI P = 4
10	0.006	0.03	0.049	0.027	0.027	0.029
50	0.045	0.942	0.822	0.595	0.282	0.158
100	0.147	3.275	3.116	1.371	0.989	0.73
200	0.622	10.004	7.293	3.876	2.884	2.455
400	2.487	30.827	34.721	17.685	10.09	9.073

Figure (4.1) visualizes the data presented in Tables (4.1) and (4.2), portraying a plot that compares the CPU time for each operation when executed on 1, 2, and 4 processors. The results are also contrasted with Eigen's implementation for comparison.

Table 4.2: *CPU time (ms) Matrix Matrix multiplication*

Size (n)	Eigen	Serial Manual	MPI P = 1	MPI P = 2	MPI P = 3	MPI P = 4
10	0.05	0.341	0.347	0.158	0.196	0.167
50	2.494	42.182	32.524	25.307	11.196	8.46
100	19.027	160.84	169.436	86.29	79.249	48.857
200	91.452	1084.279	1211.181	750.403	580.405	401.278
400	460.445	8163.854	9670.533	4947.943	3877.577	3189.448



(a) *Matrix-Vector Multiplication* **(b)** *Matrix-Matrix multiplication*

Figure 4.1: *Components Speed-up*

Figure 4.2(a) illustrates the speed-up achieved through the implementation of the described optimizations across all components. While a significant difference exists between 1 and 2 processors, the subsequent increase in speed-up is not as drastic and does not exhibit a proportional relationship to the processor count.

Accuracy and Applicability

As we conclude our project, the focus now shifts to the accuracy of the decomposition. As detailed in Section 2, the precision of this method is heavily influenced by the rate at which the spectrum of the given matrix decays, whether slowly or rapidly. Notably, in cases where matrices exhibit a slow decay in the spectrum and singular values are closely spaced, this method may not yield promising results.

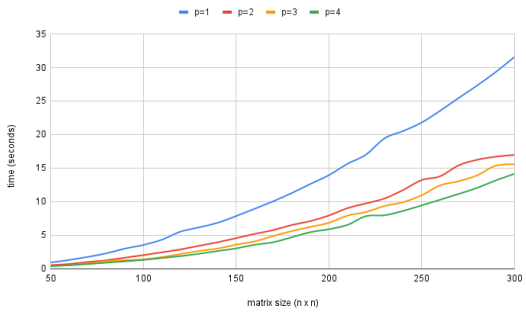
To examine the accuracy of results obtained from the rSVD, our study is divided into three distinct groups of matrices. The first group, denoted as A consists of very poorly shaped matrices, including diagonal matrices with closely spaced diagonal entries, the Identity matrix, and random matrices with a normal distribution $N(0, 1)$. The second group referred to as B, consists of block matrices of the form:

$$\begin{bmatrix} U & 0 \\ 0 & L \end{bmatrix}$$

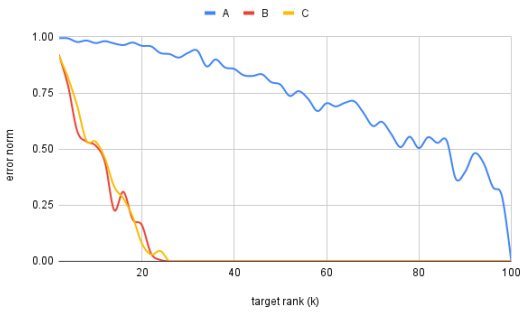
Where the values of the block U are significantly larger or smaller than those of L. The third group, C, comprises matrices that are not full-rank, with a rank approximately half

the size of their dimension. All the matrices are square and of the same size $n = 100$. Figure 4.2(a) shows how the error norm of these categories of matrices changes as we increase the target rank (k) from $k = 2$ to the size of the matrix, $k = n$. The results are averaged over a set of 20 matrices per group.

We observe rapid convergence of the error norm to zero for groups B and C. By selecting a target rank k=20, we can achieve a highly accurate SVD decomposition with significantly lower computational costs. However, in the case of group A, regardless of choosing a larger target rank, the error does not decrease to a satisfactory range. Only when k becomes as large as the size of the matrix, essentially performing a full SVD decomposition, do we observe signs of convergence.



(a) *Caption for Image 1*



(b) *Caption for Image 2*

Figure 4.2

In conclusion, rSVD emerges as a cost-effective and accurate alternative to SVD, particularly when dealing with matrices characterized by a rapidly decaying spectrum. However, its applicability diminishes in cases where the matrix spectrum does not exhibit quick decay. This limitation is particularly notable due to the reliance on the power method for finding singular values, which may not converge accurately when multiple dominant singular values are present in the matrix spectrum.

5

Application: Image Compression

5.1 Overview

Image compression is a crucial aspect of signal processing, efficient storage, transmission, and processing of visual data. It serves as a solution to challenges associated with storage capacity, bandwidth constraints, and computational resources. Compression, achieved by minimizing redundant information, enables faster transmission, reduces storage requirements, and enhances overall system efficiency.

Different techniques are utilized in image compression, such as transform-based methods like Discrete Cosine Transform (DCT), wavelet-based methods, and predictive coding. Each method strikes a balance between compression efficiency and the preservation of perceptual image quality (Dhawan, n.d.).

5.2 Data Structure

Within the image compression framework, the `Image` class serves as a data structure, encapsulating various functionalities for handling image data and compression operations. Here's a breakdown of key components within this data structure:

- Image Representation
  The heart of the `Image` class lies in the `imagematrix` member, an Eigen matrix representing the image. This matrix captures the pixel values of the image, forming the foundation for subsequent compression operations.

- SVD Components
  To facilitate compression, the class incorporates matrices and vectors to store SVD components:
  `left_singular`: Matrix storing the left singular vectors during compression.
  `singular`: Vector storing the singular values during compression.
  `right_singular`: Matrix storing the right singular vectors during compression.

- Image Metadata
  The class maintains essential metadata to characterize the image

`originalWidth`: Original width of the image. `originalHeight`: Original height of the image. `channels`: Number of color channels in the image.

- Pixel Value Range
  To ensure accurate normalization and denormalization, the class includes:
  `original_min`: Minimum pixel value in the original image.
  `original_max`: Maximum pixel value in the original image.

- File I/O Operations
  The `load` and `save` methods enable the loading and saving of image data from/to files. Additionally, `load_compressed` and `save_compressed` handle compressed image data.

- Image Manipulation
  Methods like `downscale` and `upscale` modify the image dimensions, while `normalize` and `deNormalize` ensure pixel values are within the desired range.

- Compression Operations
  The `compress` method employs singular value decomposition for image compression, allowing users to specify the rank (`k`) for compression. For parallel compression using MPI, the `compress_parallel` method is available.

5.2.1 Compression

The compression process for an image entails a series of coordinated steps, combining various methods outlined earlier in this report.These compression operations, when executed collectively, form a comprehensive and systematic approach to image compression, balancing computational efficiency with the preservation of image quality. In the implementation, this process unfolds as follows:

i. Downscaling the Image

The initial step involves reducing the dimensions of the image. While not universally applicable, downscaling proves beneficial in managing computational complexity, especially for large images. This pre-processing step aids in maintaining a reasonable computational load, particularly when applying randomized Singular Value Decomposition (rSVD) directly to extensive images, such as those sized 4000 by 4000 pixels. The downscaling factor is parameterized, typically set to 2 or 3, striking a balance between computational efficiency and image quality preservation.

ii. Normalizing the Matrix

Normalization is crucial for enhancing the numerical stability of the subsequent decomposition process. Given that pixel values in the original image range from 0 to 255,

normalizing scales these values to a standardized range of 0 to 1. This transformation contributes to a more stable and effective rSVD execution.

iii. Randomized SVD (rSVD)

The core compression operation involves performing randomized Singular Value Decomposition on the normalized image matrix. The chosen rank for the rSVD, denoted as k, is a key parameter influencing the level of compression achieved.

iv. Reconstructing the Matrix

Following the rSVD, the compressed image matrix is reconstructed using the obtained singular value decomposition matrices. This step aims to retain essential image information while significantly reducing data dimensions.

v. Denormalizing

The inverse operation of normalization, denormalizing, restores the pixel values to their original range (0 to 255). This step ensures the reconstructed matrix aligns with the original image in terms of pixel intensity.

vi. Upscaling to the Original Size

The final step involves restoring the image to its original dimensions through upscaling. This completes the compression process, yielding a compressed image that retains crucial features while efficiently managing storage and computational resources.

5.2.2 Storage

In addition to compression, our investigation extends to the impact on storage costs following the application of the rSVD method. We start by stating the fact that rSVD decomposes a matrix A of size m to three smaller matrices of sizes m, k, and k respectively, when $k < min(m, n)$. If we were to store the original dense matrix A the storage requirement would be $m.n$ units of storage. However, with the decomposed version and considering the diagonal nature of matrix $\mathbf{\Sigma}$ and its storage as a vector of size k, the storage requirement becomes:

$$m.k + k + k.n = k(m + n + 1)$$

A good selection of k can significantly reduce the space requirements.

In this project, we take advantage of the fact that image pixel values fall within the range (0, 255) and fit perfectly in a single byte. Therefore each data value, initially treated as a double, is converted to an integer and subsequently cast as an unsigned character. The two matrices U and V along with the vector $\mathbf{\Sigma}$ are stored sequentially in a binary file.To demarcate the beginning and end of each component, the size of each matrix is recorded as metadata within the file. This metadata, being of integer type, consumes

only 12 bytes (three integers, each of size 4 bytes). As stated befor we only need to know the value of m, n, and k, to effectively retrive the written values from the binary string.

Illustrating with a practical example, a PNG image sized 256×256, with a bit-depth of 8, typically occupies around 64 KB of space. However, by decomposing a 256×256 matrix and retaining 64 largest singular values, the storage cost decreases to 32 KB, resulting in a compelling ratio of 2:1.

5.3 Parallel Version

This application follows the implementation scheme described in the preceding section. However, due to the absence of explicit dependencies between pixels in an image, there exists potential for further parallelization, or at the very least, a different approach. The proposed strategy involves dividing the original image into smaller blocks. Subsequently, a serial version of rSVD is executed concurrently on these blocks. Once the decomposition is completed, the smaller matrices are aggregated to reconstruct the original matrix. The distribution of blocks among processors is again managed using MPI. The program assumes the number of processors to be working with is a square number x^2 aiming for efficiency in equal division along both dimensions of the matrix. The image matrix is partitioned horizontally and vertically into x segments, resulting in a totall of x^2 blocks to process.

At the beginning of the compression stage, each processor constructs its own unique, non-overlapping local matrix and performs the rSVD without internal parallelization. Subsequently, all the processors send a copy of their locally decomposed matrix, along with their relative position to the root processor to assemble the global matrix.

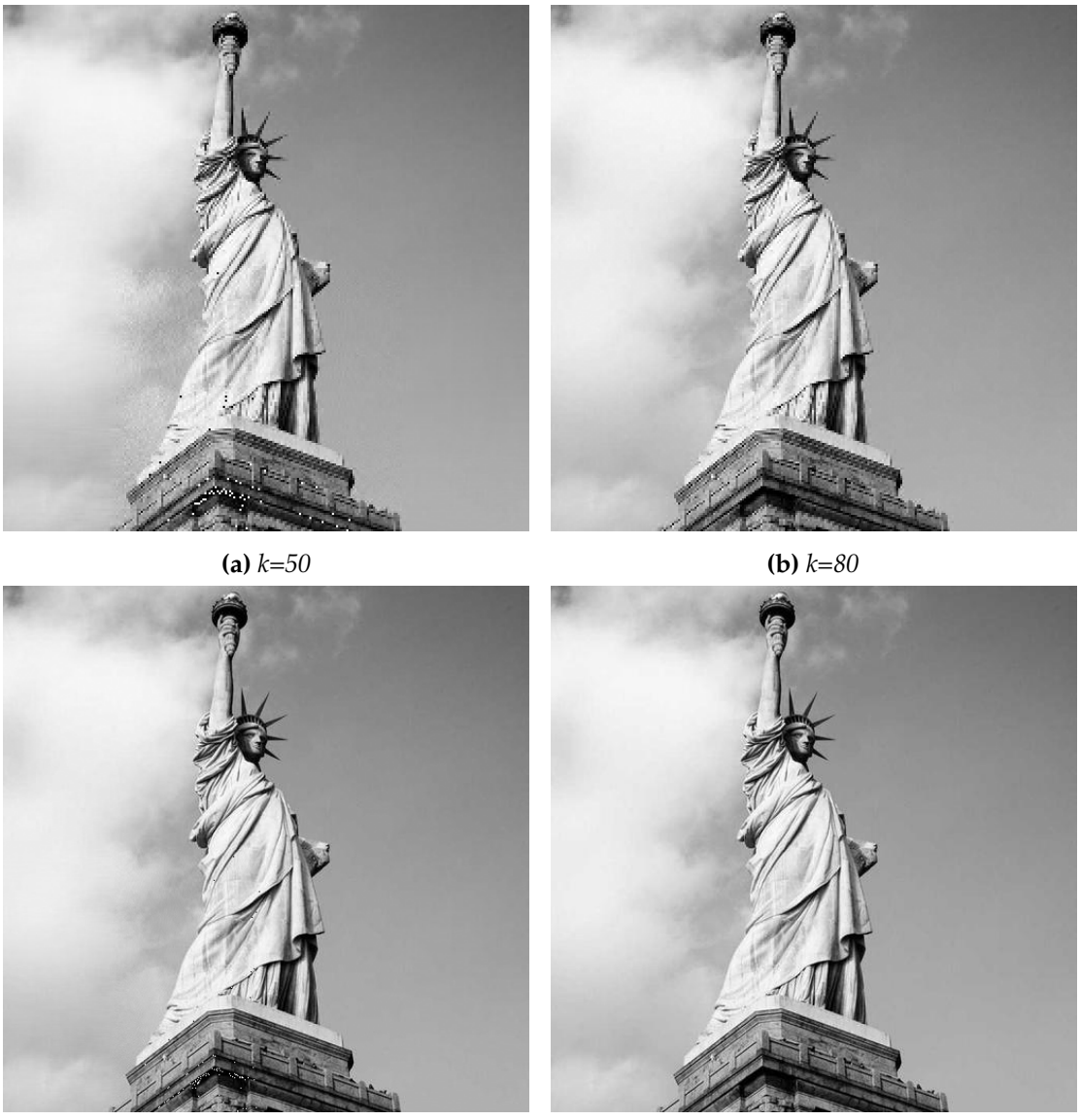
This technique works because no dependency put a sample here

5.4 Results

To assess the results obtained from compression, a delicate balance between the compression rate and image quality needs to be struck. This balance is achieved by adjusting the target rank (k). Real-world images, when represented as matrices of pixel values, often contain redundant data and can be viewed as sparse, low-rank matrices. Leveraging this characteristic, the rSVD method proves effective. Figure (5.1) illustrates the successful decomposition of an image with an original size of 512×512 by retaining only the 100 largest singular values. The figure also demonstrates how lowering the value of k impacts image quality. With $k = 50$, the general shape of the image is preserved, although with some missed pixels.

Testing the program on a collection of randomly selected grayscale images, each sized 256×256, provides insightful observations. Figure (5.2) visually represents the evolution of the error norm with increasing values of k. As k approaches 100, it becomes evident that the compression achieved is nearly lossless, as indicated by the significantly

small error. This finding emphasizes the effectiveness of the rSVD method in preserving image quality while achieving substantial compression.



(a) *k=50* **(b)** *k=80*

(c) *k=100* **(d)** *original*

Figure 5.1: *The impact of k on a* 512×512 *image*

The compression ratio is directly tied to the chosen value of k. However, to underscore the robustness of rSVD in compression, particularly with larger images, Figure 5.3 offers a glimpse into the compressed representation of an image sized 1024×1024. Remarkably, with k set to 80—roughly one-tenth of the original size—the image retains quality despite achieving a compression ratio of 5.68. This demonstration showcases the ability of the rSVD algorithm to effectively compress large images while preserving their visual fidelity.

Figure 5.4 illustrates the outcomes of the parallelization scheme detailed in Section 5.3. In subfigure (a), the execution on four processors exhibits a speed-up of approximately $SU = 3$ in comparison to a single processor.

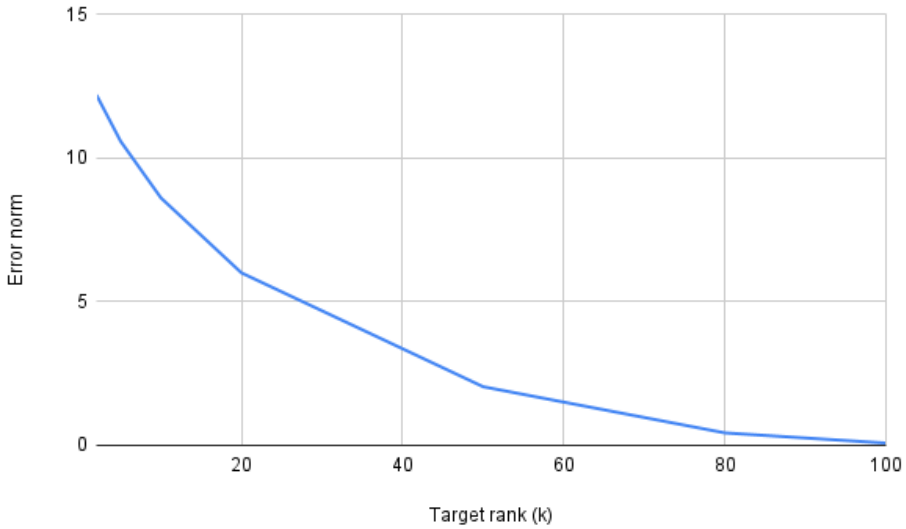
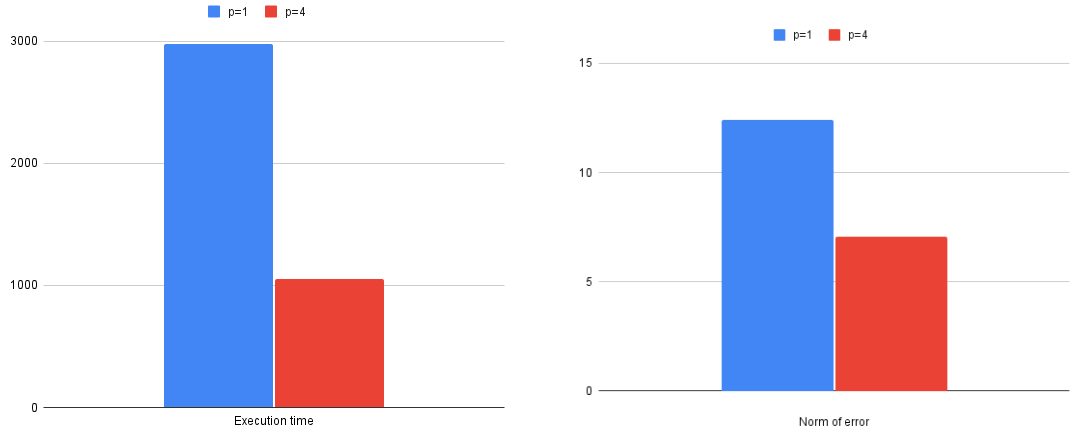


Figure 5.2: *Error norm vs. Target rank*

(a) *compressed, k=80* **(b)** *original*

Figure 5.3



(a) *Execution time* **(b)** *Error norm*

Figure 5.4

An unexpected observation emerged when comparing the error norm between the distributed and sequential versions of the program. As illustrated in Figure 5.4(b) the error norm with four processors working on four blocks of the image is markedly lower than with one processor processing the entire image. This phenomenon may be attributed to the efficiency of rSVD in finding local singular values more quickly and effectively while working on a subset of the main matrix.

6

Conclusion

In conclusion, the study examined the performance of the rSVD algorithm across different types of matrices, categorized as A, B, and C. The results, as presented in Figure 4.2(a), revealed rapid convergence of the error norm to zero for groups B and C, showcasing the algorithm's effectiveness in achieving accurate SVD decomposition with reduced computational costs, particularly when a moderate target rank (k=20) was selected.

However, for matrices in group A, the error did not decrease to a satisfactory range, even with larger target ranks. Only when the target rank (k) approached the size of the matrix did signs of convergence become apparent. This limitation underscores the algorithm's dependency on the rapid decay of the matrix spectrum.

Moving on to image compression, the report provides an extensive overview of the data structure, compression operations, storage considerations, and a parallelized version of the rSVD algorithm. The study showcased the algorithm's efficacy in compressing images while balancing computational efficiency and image quality. Figure 5.1 depicted the impact of varying k on a 512×512 image, illustrating the trade-off between compression and image quality.

Further results demonstrated the near-lossless compression achieved with larger values of k, emphasizing the robustness of rSVD in preserving image quality. Additionally, the parallelized version of the algorithm exhibited a notable speed-up when executed on four processors (Figure 5.4(a)).

An unexpected finding was observed when comparing the error norm between distributed and sequential versions of the program. Figure 5.4(b) highlighted that the error norm with four processors working on blocks of the image was significantly lower than with one processor processing the entire image. This suggests the efficiency of rSVD in finding local singular values more quickly and effectively in a distributed processing setting.

Overall, the study establishes rSVD as a valuable tool for image compression, with its performance contingent on the decay characteristics of the matrix spectrum. The results presented contribute to a comprehensive understanding of the algorithm's strengths and limitations in various applications.

Bibliography

(N.d.). Accessed on: Monday 29th January, 2024. URL: https://www.cs.yale.edu/homes/el327/datamining2013aFiles/07_singular_value_decomposition.pdf.

(N.d.). Accessed on: Monday 29th January, 2024. URL: https://courses.engr.illinois.edu/cs554/fa2015/notes/11_qr.pdf.

Andrew, Robert and Nicholas Dingle (n.d.). "Implementing QR Factorization Updating Algorithms on GPUs". In: *Journal of Parallel and Distributed Computing* (). Article history: Available online 26 March 2014.

Dhawan, Sachin (n.d.). "A Review of Image Compression and Comparison of its Algorithms". In: *International Journal of Computer Applications* (). Deptt. of ECE, UIET, Kurukshetra University, Kurukshetra, Haryana, India.

Halko, N., P. G. Martinsson, and J. A. Tropp (2011). "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2, pp. 217–288.

Tapia-Romero, Miguel, Amilcar Meneses-Viveros, and Erika Hernandez-Rubio (2020). "Parallel QR Factorization using Givens Rotations in MPI-CUDA for Multi-GPU". In: *International Journal of Advanced Computer Science and Applications* 11.5, pp. 1–2.