Distributed Computing of Large-Scale Singular Value Decompositions

Sheng Fang
Pembroke College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Hilary 2018

To my grandmother

Acknowledgements

I would like to express my greatest gratitude to my supervisor, Prof. Raphael Hauser, for his guidance and support throughout my doctoral studies. His valuable suggestions and encouragement during my work made it possible.

I thank my parents for their continuous support. I cannot express how grateful I am for their love and patience. I would also thank all my friends in Oxford. The warmth and strength of your friendship carry me through the difficulties.

I am also grateful to have received financial support from EPSRC and the China Scholarship Council.

Abstract

Big data projects increasingly make use of networks of heterogeneous computational resources for scientific computing purposes. To meet the requirements of big data applications and harness the power of modern computing architectures, novel highly parallel algorithms for numerical linear algebra computations are needed that rely on as little node synchronicity and data communication as possible. Distributed algorithms are particularly important in situations where the data itself is naturally distributed across several or many servers, or where the data collection is decentralised. The singular value decomposition (SVD) is one of the fundamental matrix decompositions and a cornerstone of numerical linear algebra. The computation of leading part SVDs plays an essential role in a variety of models and algorithms and dominates their computational costs.

In this thesis, we analyse a distributed algorithm for leading part SVD computations of large-scale matrices. The algorithm is well adapted to cloud computing or computer networks, as it satisfies the loose coupling requirement (low synchronicity, low communication overhead). The algorithm was first introduced by R. Hauser and D. Goodman and was first published in the latter's DPhil thesis [47]. We will give the geometric intuition behind the algorithm and introduce it formally, but the main part of this thesis will focus on the analysis of the distributed SVD algorithm and its experimental validation.

Two different approaches will be proposed to investigate the convergence of the algorithm. The first approach utilizes classical matrix analysis tools, leading to an error bound. The second approach of analysis provides more geometric insights. The framework based on geometric observations gives bounds on the global aggregation of numerical approximation errors that occur in local computations. Bounds on these local errors are studied probabilistically under the assumption of random input.

Finally, we report numerical experiments, as well as the discussion of the variants of the algorithm. Applications in matrix optimisation problems, including low-rank matrix completion and sparse principal component analysis, will also be presented, and we use these to obtain an experimental validation of the algorithm under non-random input.

Contents

List of Figures

Chapter 1

Introduction

Thanks to technological advances in data gathering and storage, massive data-sets have been gathered in diverse areas such as climate science, cosmology, medicine, the internet, and engineering. Novel mathematical and statistical models have been developed to extract meaningful information from this data, and the central computational issue of such methods are matrix computations. Distributed methods for matrix computations are required in this context, as the data is often distributed on computer networks or in a cloud infrastructure. Singular value decomposition (SVD) is one of the fundamental matrix decompositions and a cornerstone of numerical linear algebra. It has grown enormously important in data science and engineering in recent years with applications ranging from web search models to the physical and biological sciences. In this chapter, we discuss distributed computing in modern computer architectures and review existing numerical methods for SVD.

1.1 Distributed computing and loosely coupled requirements

Computer networks, multi-core processors and graphics cards make parallel computing more and more affordable to standard users of scientific computing software. Many conventional parallel algorithms have been designed with supercomputers in mind as a computational resource. They often require synchronicity of nodes. Any lack of synchronicity affects load balancing and reduces the performance of such algorithms. In order to enable this type of computing, the architecture of supercomputers was therefore optimised to render node-to-node communication as time-efficient as possible.

However, classical parallel algorithms are not always suitable for new types of parallel computing resources, which have very different characteristics: the processing speeds of different nodes can be very heterogeneous, either because different types of processors are used, or because the nodes are busy running a variety of other threads. Node failure is not unlikely, either because individual nodes can be switched off or delayed indefinitely by a third party user or thread. This results in loss of synchronicity which may severely slow down the speed of classical parallel methods. Finally, in networks of distributed computing resources, the node-to-node communication is particularly slow in comparison to the time required by the numerical computations executed at each node (the internet may have to be used for node-to-node communication). In such a situation, the communication overhead can become the dominant factor in terms of overall execution time, monetary cost and energy consumption. Examples of large data-sets for which the last three costs are all very significant include the data held by the *climateprediction.net* project at the University of Oxford or the new NHS database of patient data. In both cases one would like to perform principal component analysis (PCA) on the data matrices. This involves computing leading part singular value decompositions of matrices whose rows or columns are distributed over multiple servers.

To harness the full potential of distributed computing resources and distributed data-sets for the purposes of scientific computing, new types of algorithms are needed. We call such algorithms loosely coupled or distributed, as they need to satisfy the following requirements:

i) Communication overheads must be minimised as much as possible.

ii) The algorithm must be as unreliant as possible on node synchronicity.

iii) To render a method resistant to node failure, it must be able to meaningfully employ other nodes while a failed process is restarted.

Distributed computing has attracted much attention from both researchers and practitioners in scientific computing and other fields. Let us discuss the above three loose-coupling requirements in more details.

There are two sources of costs in distributed computing: arithmetic operations and communication. By 'communication', we refer to the movement of data between different processors. In modern large-scale computing, communication costs are becoming a central issue. The typical speed of computer is on the order of teraflops while the typical communication speed is on the order of 10 gigabytes per second.

The time to perform one flop on a computer has been reducing by about 59% per year over the 16-year period from 1988 to 2004, but the communication rate, that is, the memory bandwidth, has lagged behind by increasing by about 23% per year from 1995 to 2005 [48]. At the same time, power consumption is becoming a more and more important consideration in scientific computing today. In their paper [7], Balige et al. presented an analysis of energy consumption in cloud computing. They pointed out that communications account for a significant percentage of the total energy consumption in cloud computing. Communication avoidance is therefore capturing ever closer attention in the field of scientific computing.

A group of communication minimising methods for linear algebra were proposed by Demmel et al. [8], [9], [30], including SVD, eigendecomposition and rank revealing QR. In [9], Ballard, Demmel, and Dumitriu proposed a family of spectral divide-and-conquer algorithms for eigenproblems and SVD that use only QR decompositions and matrix multiplications. The divide-and-conquer step is based on a randomized rank-revealing factorization method and the spectrum is divided into two parts.

The development of the internet made crowd-sourcing a possible mode of collaboration, both by collecting data in a decentralised manner and by making heterogenous distributed computing resources available for arithmetic operations. Data-intensive computing can take advantage of this type of collaboration. An example is the the *climateprediction.net* project [1], [74], [90], the world's largest climate modelling experiment, which runs climate models on volunteers' personal computers. It distributes computing tasks to machines world-wide. This allows the project to answer important and difficult questions about how the climate will change in the 21st century. Principal component analysis is used in the data analysis. It involves computing leading part singular value decompositions of submatrices whose data are distributed over multiple servers.

In the model of *climateprediction.net*, as well as many other distributed computing platforms, node failure occurs with non-negligible probability. When designing an algorithm for distributed heterogeneous computing resources, resistance to node failure is an essential requirement. Asynchronicity is another challenge. Unlike supercomputers, heterogeneous computing environments cannot guarantee synchronicity. In the synchronous case, the computing time is determined by the slowest or busiest computing node, which could take much longer than the average time a node takes to complete this task. Any synchronicity requirement of the algorithm would waste time and good use of computing resources. It is thus a great advantage of an algorithm that can avoid the need for synchronicity.

A good start in designing loosely coupled algorithms is to develop salient methods for matrix factorisations, as this problem sits at the heart of numerical linear algebra and scientific computing. Classical matrix-factorization algorithms typically involve sequential left- and right- matrix-vector or matrix-matrix multiplications (e.g. multiplication by Householder vectors, application of Givens rotations, orthogonal iteration etc.) that require node synchronicity and cause significant communication overheads.

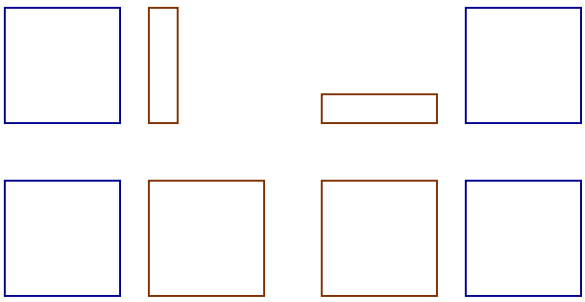

Figure 1.1: Left- and right- matrix-vector or matrix-matrix multiplications.

Figure 1.1 illustrates left- and right- matrix-vector or matrix-matrix multiplications. If the columns of the blue matrix are distributed over several nodes in blocks of data, right-multiplication with the red vector or red matrix requires all nodes to work in synchronicity and communicate partial results before they can be summed – see Figure 1.3 for a depiction of this process. The situation is similar with regards to left-multiplication when the rows of the blue matrix are distributed. The requirements on loosely coupled linear algebra algorithms is to avoid any such products as much as possible.

Let us consider a matrix-vector product for a matrix with distributed columns. While individual nodes perform part of the summation, their results need to be communicated to allow for taking the total sum. If the matrix is of size $m \times n$ and is distributed over q nodes, this results in communication of $O(mq)$ floating point numbers, no matter in which order the sum is assembled, and it requires all local computations to have completed before taking the final sum can succeed. This requirement of synchronicity may greatly reduce the speed and efficiency of the methods in heterogeneous computing.

From here on, we will be focused on loosely coupled algorithms for the computation of a leading part SVD of a matrix that is subdivided into blocks of data held in the memory of different nodes, see Figure 1.2. More specifically, the SVD algorithms we will discuss are based on distributing the data of a matrix by blocks of columns.

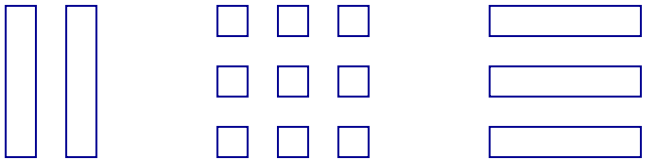

Figure 1.2: Subdivision of a matrix into blocks of data held in the memory of different nodes.

These algorithms can be easily adapted to the situations where the data is distributed in blocks of rows (apply the algorithm to the transpose of the matrix, and take the transpose of the final result) or into blocks of columns that are further distributed as blocks of rows (apply our algorithms on two recursive levels).

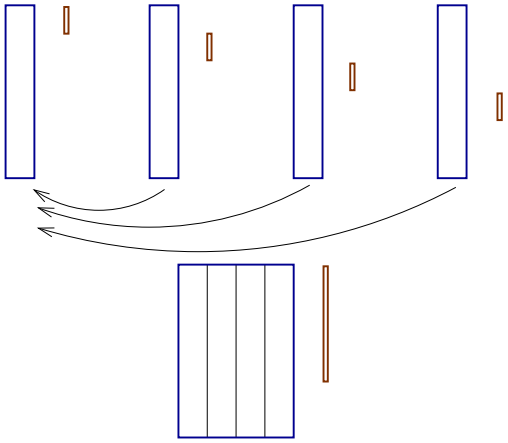

Figure 1.3: Communication of parallel matrix-vector right-multiplication with column-splitting.

Example 1.1. *The data of the Climateprediction.net experiment at Oxford University constitutes a dense matrix of size* $O(10^7 \times 10^6)$ *and is distributed in blocks of columns across dozens of servers located around the globe. A single matrix-vector product would require the communication of* $O(10^8)$ *floating point numbers.*

By design of our algorithm, we do not require the data matrix to be sparse. We do not specifically discuss the issue of exploiting sparsity either, but we remark that when a matrix is sparse, then at least the local computations in the so-called seed nodes (explain later), which account for the dominant part of the arithmetic costs, deal with sparse matrices, and all the algorithms we employ in this context can be replaced by their sparsity-exploiting variants.

The algorithm we will present in this thesis was developed by Hauser and Goodman and was first published in Goodman's DPhil thesis [47]. Its convergence analysis is the main focus of this thesis. Before giving details and analysis of the algorithm, let us review SVD and existing algorithms.

1.2 SVD and existing algorithms

Matrix computations, especially matrix decompositions are fundamental in scientific computing. The efficiency of matrix computations determines the performance of many numerical methods in science and engineering. SVD is one of the basic matrix decompositions, whose applications range from least-squares approximation, rank determination, low-rank matrix approximation, principal component analysis to chemical physics [65], genetic analysis [4, 53], financial mathematics [39] and face recognition [95, 96]. In many of these applications, SVD computations dominate the computational costs. Often an SVD calculation is required for the purposes of the inner-most loop of iterative algorithms, and the matrices are often large and dense. It is therefore desired to have new algorithms for the computation of the SVD or leading part SVD of large and dense matrices that can make efficient use of the available computing resources and infrastructure.

Given a matrix $A \in \mathbb{R}^{m\times n}$, an SVD [45, 54, 55] of A is given by

$$A = USV^{\mathrm{T}} = \sum_{t=1}^{k} \sigma_t u_t (v_t)^{\mathrm{T}}, \tag{1.1}$$

where $k = \min(m, n)$, U is an $m \times m$ orthogonal matrix, V is a $n \times n$ orthogonal matrix, with transpose V^{T}, and S is a $m \times n$ diagonal matrix with non-negative real numbers in decreasing order on the diagonal. Any matrix $A \in \mathbb{R}^{m\times n}$ has an SVD, and if the diagonal entries of S are all different from each other, the SVD is unique [51]. The diagonal entries of S are called the *singular values* of A. If the singular values satisfy

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_m = 0, \tag{1.2}$$

then rank(A)$= r$. The columns u_i $(i = 1, \ldots, m)$ of U are known as the *left singular vectors* of A and the columns v_i $(i = 1, \ldots, n)$ of V are the *right singular vectors* of A.

In many applications, instead of the full SVD, it is sufficient to acquire only a leading part SVD, i.e., the largest p singular values and the corresponding singular

vectors for $p < m, n$. A p-leading SVD of A is

$$A_p = U_p S_p V_p^{\mathrm{T}} = \sum_{t=1}^{p} \sigma_t u_t (v_t)^{\mathrm{T}}, \tag{1.3}$$

where U_p is a $m \times p$ Stiefel matrix composed of the first p columns of U from the full SVD and V_p is a $n \times p$ Stiefel matrix composed of the first p columns of V from the full SVD. They are the truncated p leading part of U and V, respectively. A *Stiefel matrix* is a matrix with mutually orthonormal columns. Matrix S_p is the $p \times p$ leading part of S. In many applications, $p \ll \min(m, n)$.

The operator 2-norm of a matrix A is defined as

$$\|A\|_2 = \sup_{x \neq 0} \left(\frac{\|Ax\|}{\|x\|} \right), \tag{1.4}$$

where $\|x\|$ is the 2-norm of vector x.

The Frobenius norm of a matrix A is defined as

$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} \|a_{ij}\|^2 \right)^{1/2}. \tag{1.5}$$

A p-leading SVD is the best rank-p approximation of A in the sense of the 2-norm, i.e.,

$$U_p S_p V_p^{\mathrm{T}} = \underset{D \in \mathbb{R}^{m \times n}, rank(D) \leq p}{\operatorname{argmin}} \|A - D\|_2, \tag{1.6}$$

and the approximation is quantified by

$$\|A - U_p S_p V_p^{\mathrm{T}}\|_2 = \sigma_{k+1}. \tag{1.7}$$

Likewise, it is also the best rank-p approximation of A in terms of the Frobenius norm,

$$U_p S_p V_p^{\mathrm{T}} = \underset{D \in \mathbb{R}^{m \times n}, rank(D) \leq p}{\operatorname{argmin}} \|A - D\|_F, \tag{1.8}$$

and the approximation is quantified by

$$\|A - U_p S_p V_p^{\mathrm{T}}\|_F = \sqrt{\sum_{t=p+1}^{r} \sigma_t^2}. \tag{1.9}$$

In this thesis, without further specification, we use $\|\cdot\|_2$ to denote the 2-norm of a matrix and $\|\cdot\|_F$ to denote the Frobenius norm of a matrix. The 2-norm of a vector is denoted by $\|\cdot\|$.

The optimality properties (1.6) and (1.8) make leading part SVDs useful in many low-rank or low-dimension approximation applications. One of the examples is principal component analysis (PCA). PCA is a classical tool in multivariate data analysis [41]. It is applied for the purpose of dimensionality reduction, feature extraction and data visualisation [58]. Given an input data matrix X, which is usually a covariance or correlation matrix, the goal of PCA is to find a lower dimensional space that maximises the variance of the projected data among subspaces of a given dimension. We want to find a sequence of factors ranked by variance. Each factor is a linear combination of the variables. This technique is used to reduce the number of dimensions of a model while retrieving most of the information diversity (variance) contained in the simplified model. Numerically, PCA is computed through the SVD of the covariance matrix. We assume $X \in \mathbb{R}^{n\times n}$ is a mean-centered data matrix with columns corresponding to features (or random variables of interest) and rows corresponding to i.i.d. samples of the feature vectors. Consider the SVD of X,

$$X = USV^{\mathrm{T}} = \sum_{i=1}^{n} \sigma_i u_i (v_i)^{\mathrm{T}}. \tag{1.10}$$

Then, the column vectors of V are the principal components directions of X. The first right singular vector v_1 is the first leading principal component direction. Vector $z_1 = Xv_1 = \sigma_1 u_1$ is the first principal component (PC) of X. It has the largest variance among all normalised linear combinations of the columns of X. Vector v_2 and $z_2 = Xv_2 = \sigma_2 u_2$ are the second largest principal component direction and PC, respectively.

SVD plays an important role in various applications. The numerical methods for SVD computing have been evolving with the developments in computing hardware, theory of scientific computing and the requirements arising in new problems. In the following section, we will review some important algorithms in the literature, including classical SVD algorithms, randomized methods and communication-efficient methods.

1.2.1 Classical SVD algorithms

SVD dates back to Beltrami [10] and Jordan [60, 61] in the 1870s as a topic in differential geometry. Around 1910, Picard first used the term *singular values* [80]. Significant developments in numerical algorithms for SVD started by Golub and Kahan in 1965 [42]. The Golub-Kahan algorithm is still one of the most-used methods today. In the following, we give a short review of the Golub-Kahan algorithm.

The SVD of a matrix A is closely related to the eigendecompositions of the symmetric matrices $A^{\mathrm{T}}A$,

$$A^{\mathrm{T}}A = VS^2V^{\mathrm{T}}. \tag{1.11}$$

Mathematically, we can reduce the computation of the SVD of A to the eigenvalue decomposition of $A^{\mathrm{T}}A$. However, this approach may be unstable, because the eigenvalue problem may be more sensitive to perturbations. Alternatively, the Golub-Kahan method implicitly applies the symmetric QR algorithm to $A^{\mathrm{T}}A$.

In the first step, it reduces A to upper bidiagonal form by application of Householder transformation. For a hyperplane, there is a unit vector v that is orthogonal to the hyperplane. The Householder matrix associated with this vector is given by

$$F = I - \frac{2vv^{\mathrm{T}}}{v^{\mathrm{T}}v}. \tag{1.12}$$

Matrix F is unitary, in fact it is a reflection in the hyperplane $V^{\perp}$. A Householder transformation uses unitary matrix

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & F \end{bmatrix}, \tag{1.13}$$

to introduce zeros to the first column of a matrix by left multiplication, or the first row of a matrix by right multiplication. The vector v is chosen as

$$v = \|x\|e_1 - x, \tag{1.14}$$

where x is the vector of the $2, \ldots, m$-th entries of the first column or the first row, and e_1 is the same dimensional vector whose first entry is 1 and other entries are zero.

Given a matrix A, we apply Householder transformation alternately on the left and the right. Each left multiplication introduces a column of zeros below the diagonal, while each right multiplication introduces a row of zeros to the right of the first superdiagonal. At the end of this process, A is reduced to a bidiagonal form,

$$U_B^{\mathrm{T}}AV_B = B, \tag{1.15}$$

where B is upper bidiagonal, i.e. B only has non-zeros on its diagonal and the first superdiagonal. Through this step, one implicitly tridiagonalises $A^{\mathrm{T}}A$, i.e., $V_BA^{\mathrm{T}}AV_B^{\mathrm{T}} = B^{\mathrm{T}}B$ is tridiagonal.

In the second step of Golub-Kahan algorithm, we calculate the SVD of the bidiagonal matrix B. A standard method is implicit-shift QR algorithm. The resulting λ estimates an eigenvalue of $A^{\mathrm{T}}A$, which is the square of a singular value of A.

Though the Golub-Kahan method is widely used, its application to very large matrices is limited. The bidiagonalisation costs $4mn^2 - \frac{4}{3}n^3$ flops, which is too costly when n is large. Another shortcoming of Golub-Kahan method in practice is that it computes the full SVD, while only a leading part SVD is needed in many applications.

Another popular approach to compute the SVDs is the Lanczos method. It or its variants are widely used for large and sparse SVD; especially when extreme singular values and vectors are needed.

The Lanczos method is motivated by optimising the *Raleigh quotient*, which is defined as

$$r(w) = \frac{w^{\mathrm{T}} A^{\mathrm{T}} A w}{w^{\mathrm{T}} w}. \tag{1.16}$$

The maximum and minimum values of $r(w)$ are equal to $\sigma_1^2(A)$ and $\sigma_n^2(A)$ respectively, and the Lanczos process iteratively builds up bases

$$Q_k = [q_1, \ldots, q_k] \tag{1.17}$$

of subspaces

$$\mathcal{K}_k = \mathrm{range}(Q_k) \tag{1.18}$$

with the properties that $\max_{w \in \mathcal{K}_k} r(w)$ gradually increases in k and converges to $\max_{w \in \mathbb{R}^k} r(w)$, while $\min_{w \in \mathcal{K}_k} r(w)$ decreases and converges to $\min_{w \in \mathbb{R}^k} r(w)$. We define M_k, m_k, u_k and v_k by

$$M_k = r(u_k) = \sigma_1^2(AQ_k) \leq \sigma_1^2(A), \tag{1.19}$$

$$u_k = Q_k \underset{\|y\|=1}{\mathrm{argmax}}\ r(Q_k y), \tag{1.20}$$

$$m_k = r(v_k) = \sigma_k^2(AQ_k) \geq \sigma_k^2(A), \tag{1.21}$$

and

$$v_k = Q_k \underset{\|y\|=1}{\mathrm{argmin}}\ r(Q_k y). \tag{1.22}$$

It is natural to choose the next subspace $\mathcal{K}_{k+1}$ such that it contains the steepest ascent directions

$$\nabla r(u_k) = \frac{2}{u_k^{\mathrm{T}} u_k}(Au_k - r(u_k)u_k), \tag{1.23}$$

and

$$\nabla r(v_k) = \frac{2}{v_k^{\mathrm{T}} v_k}(Av_k - r(v_k)v_k). \tag{1.24}$$

On the other hand, we want to tridiagonalise $A^{\mathrm{T}}A$, i.e., to make $Q_k^{\mathrm{T}} A^{\mathrm{T}} A Q_k$ tridiagonal. The Krylov subspace is defined as,

$$\mathcal{K}_k = \mathrm{range}\{q_1, Aq_1, \ldots, A^{k-1}q_1\}, \tag{1.25}$$

where q_1 is an initial vector, and satisfies both requirements simultaneously. The SVD of A is approximated by the SVD of the tridiagonal matrix $Q_k^{\mathrm{T}} A^{\mathrm{T}} A Q_k$. The convergence properties of the Lanczos method were studied by the Kaniel-Paige theory, see [63], [79] and [87].

The Lanczos method does not generate the need to store large intermediate submatrices and the leading singular values emerge long before the tridiagonalisation is finished. Therefore, it is particularly powerful for large and sparse symmetric eigenproblems. But A drawback of Lanczos method is that rounding errors greatly affect its performance.

A practical Lanczos-type method is the block-Lanczos method, the block generalization of the Lanczos process. In the block-Lanczos method, the initial vector is replaced by k independent vectors $\{x_1, x_2, \ldots, x_k\}$. A similar Lanczos iteration is carried out until a block bidiagonal matrix $\bar{T}$ is constructed. Efficient Block Lanczos schemes for eigenproblems were introduced by Golub and Underwood [44], [97], Cullum and Donath [26], [27], Lewis [69] and Ruhe [86]. Golub, Luk and Overton's paper [43] specifically studies the block-Lanczos method for SVD.

Lanczos and related methods provide iterative techniques for calculating the leading part SVD of a matrix in which part of the calculations can be processed in parallel on a regular network of processors. After each iteration step of any of these techniques, a significant number of processors have to communicate information to each other on the current results of the iteration before carrying out the next iteration step. This means that processors are interlocked at every iteration step. Such interlocking means that communication latency and waiting for processor to synchronize will be a limiting factor on the speed of processing, and failure of processors can result in severe delays. This may, in practical terms, prohibit such parallel processing over a distributed network, wherein the speed of communication is significantly lower than the processing speed of a processor. Even in the parallel processing environment on non-distributed systems, communication latency can be the over-riding limiting factor on processing speed, with communication speed far lower than CPU speed. These limitations of Lanczos-type methods make them inadequate for heterogenous computing, especially when synchronicty is not guaranteed and communication is slow.

1.2.2 Randomized methods

An important group of state-of-the-art leading part SVD algorithms are randomized methods. In this section, we review the family of algorithms, which have been at-

tracting a great deal of attention recently in many fields, such as numerical linear algebra, theoretical computer science, statistics and machine learning. The use of randomness leads to faster and simpler algorithms which are easier to analyse. The outputs of randomized methods are interpretable and robust, especially when part of the data is missing or inaccurate, and they are well adapted to the novel computational architectures.

Randomized Numerical Linear Algebra (RandNLA) is one of the most exciting advances in this direction. The use of randomization is not new in scientific computing [50], [76], but demands from large-scale computing applications have led to fresh interest in developing randomized methods for large matrix problems.

Probabilistic algorithms of low-rank approximation of large matrices are one of the important advances in large scale matrix computing.

Such methods for low-rank matrix approximation usually proceed via two computational stages. In the first stage, an approximate basis for the range space of the input matrix is obtained through random sampling. Given an $m \times n$ matrix A and a number c, a basic random sampling algorithm usually carries out two steps:

1. Compute sampling probabilities $\{p_i\}_{i=1}^n$.

2. Randomly select and rescale c columns of A according to these probabilities to form an $m \times c$ matrix C.

The sampling probabilities are usually chosen to reflect the relative 'importance' of the columns. For example, in [34], the sampling probability is defined as

$$p_i = \frac{\|A^{(i)}\|_2^2}{\|A\|_F^2},$$

where $A^{(i)}$ is the i-th column of A. This is called the length-squared sampling. We remark that in randomized sampling methods, row sampling is used for tall and thin matrices, while column sampling is used for short and wide matrices.

An alternative approach to random sampling, is to right-multiply matrix A with an $n \times c$ random projection matrix Ω and calculate a low-rank decomposition of the reduced matrix. Early works include Johnson and Lindenstrauss [57], who carry out an orthogonal projection onto a random c-dimensional space. Other random projections were proposed in [40], [28], [56], [3] and [75].

In the second stage, matrix C is used to compute an approximate factorisation of A, which can be done by well-established deterministic methods, such as the Golub-Kahan method or Lanczos' method. Let us take SVD as an example: suppose Ω is

an $n \times c$ matrix computed in the first stage that gives the basis of a c dimensional subspace. Let $C = A\Omega$. We compute the k-leading SVD of matrix $C = US\hat{V}^{\mathrm{T}}$, which has running time $O(mck)$. Then $A \approx USV^{\mathrm{T}}$, where $V = \Omega\hat{V}$, is an approximation of the SVD of A.

Let us discuss Algorithm 1 in [34] as an example. This was one of the first randomized SVD algorithms. It uses random column sampling in the first stage. More refined random sampling methods were developed and analysed in [72], [35], [14] and [85].

Algorithm 1 A fast Monte Carlo algorithm for SVD

Input $A \in \mathbb{R}^{m\times n}$, c and k are positive integers such that $1 \le k \le c \le n$, probabilities $\{p_i\}_{i=1}^n$ such that $p_i > 0$ and $\sum_{i=1}^n p_i = 1$
For $t = 1 : c$
  Pick $i_t \in \{1, \ldots, n\}$ with Prob$[i_t = i] = p_i$
  Set $C^{(t)} = A^{(i_t)}/\sqrt{cp_{i_t}}$
End
Compute $C^{\mathrm{T}}C$ and its SVD: $C^{\mathrm{T}}C = \sum_{t=1}^c \sigma_t^2(C) y^t y^{t\mathrm{T}}$
Compute $h^t = Cy^t/\sigma_t(C)$ for $t = 1, \ldots, k$.
Output $H_k = [h^1, \ldots, h^k]$ and $\sigma_t(C)$, $t = 1, \ldots, k$

Suppose that U^*, S^* and V^* are outputs from the randomized SVD method that compute k-leading part SVD of A. An $n \times 2k$ standard Gaussian matrix Ω is used as test matrix in the first stage to construct $C = A\Omega$. For some matrices that have slow-decaying singular values, power iterations can be used in the second stage. Let q denote the number of power iterations. In review [49], an upper bound of the expectation of the error of the truncated k-SVD computation, with respect to the Gaussian test matrix, is given,

$$\mathbb{E}\|A - U^*S^*(V^*)^{\mathrm{T}}\|_2 \le \sigma_{k+1}(A) + \left[1 + 4\sqrt{\frac{2\min\{m,n\}}{k-1}}\right]^{-1/(2q+1)} \sigma_{k+1}(A), \quad (1.26)$$

This is also the bias of the low-rank approximation from the computed leading part SVD to matrix A. Note that $\sigma_{k+1}(A)$ is the lower bound of any rank-k approximation of A. This bound shows that, in expectation, randomized method can speed up SVD computation with only adding a small term in error.

In randomized methods such as Algorithm 1, column-selection is carried out only in the first stage and the data that is not selected is then ignored in the second stage, where the computations of an approximate decomposition is restricted to the subspace computed in the first space. In contrast to this, our new method uses column (or row)

splitting for the purpose of parallelisation, with local node computations resembling the computations on randomly sampled submatrices, but the results of the local computations are then combined in a map-reduce operation that end up taking the entirety of the data into account. This has the advantage that a high accuracy leading SVD can usually be computed in a single pass of the data. Furthermore, in data-sets in which the information relevant to the leading singular vectors is concentrated in very few columns, random sampling techniques require the number c of samples to be large relative to n, in order to be guaranteed to sample the relevant columns with high probability, thus reducing the complexity gains made through dimensionality reduction and necessitating some prior knowledge of information concentration in order to make an informed choice of c. In contrast, since our new method uses all the data, it benefits from a fixed dimensionality reduction gain independently of information concentration.

In the settings of the new algorithm, the input matrix A is only transferred to the processors once at the beginning of algorithm. During the computations, the node-to-node communication is of order $O(mk)$, which is much less than the dimension of the input matrix. Meanwhile, the local computations are also much reduced. In the seed nodes, we only carry out the local SVD computations on matrices of size $m \times \frac{n}{s}$, where s is the number of seed nodes. As this can be done in parallel, the running time is $O(\frac{mnk}{s})$, using any classical method. Once the seed nodes are finished, the local computations in other nodes are all carried out on matrices of size $m \times 2k$, Using the classical method, these computations can be done in $O(mk^2)$ time. Therefore, we can achieve much higher accuracy using the new method, with only a very small additional communication cost, and the computation time is also reduced almost inverse proportionally to the number of nodes used, on condition that each node has its own memory.

In this thesis, we will show that the error bound for our new method is $\sqrt{2s-1}\sigma_{k+1}$, where s denotes the number of seed nodes and σ_{k+1} is the $(k+1)$-th singular value of input matrix A. In theory, the error increases when using more processors. However, in practice, we note that this growth of error is very slow when using larger number of processors. Furthermore, this is a deterministic bound, while the bound (1.26) is an expectation over randomized test matrix.

Node failure is another issue in distributed computing on heterogeneous resources, such as in a computer network. Our new method is particularly resistant to node failure. When a node fails, there may be two possible situations. The first case is that the data is still available. We can transfer this data to another working node and

this has no effect to the whole computation, neither on the accuracy nor in general on the overall computation time, as our method requires no node synchronicity and the topology of the map-reduce tree can be adapted on the fly. In the second case, the data-set is missing. This is similar to the randomized column selection method. This affects the accuracy but not the total time of the computation, but the impact is restricted to the corresponding subspace.

Our method is thus well adapted to modern parallel computing architectures, especially those that combine heterogeneous computational resources. It is also an important tool when singular value and vectors are needed for large-scale matrix with high requirement of accuracy.

1.2.3 Communication efficient distributed methods

Communication efficient distributed methods [47], [70], [15], [81], [38] are another group of novel large-scale SVD methods emerging recently. These are methods that are used to compute the SVD or PCA of large matrices when data is stored in multiple processors, with the aim of achieving a small error with limited communication cost.

One such method is the communication-optimal method for distributed PCA using the column-partition model, which is almost identical to the method of Hauser and Goodman published in [47] in 2007, but was subsequently independently discovered by Boutsidis and Woodruff [15] in 2015. In the setting of the method, subsets of columns of the large matrix are distributed to the processors. Local computations are conducted to find a Stiefel matrix $U \in \mathbb{R}^{m\times k}$ such that

$$\|A - UU^{\mathrm{T}}A\|_F^2 \leq (1+\epsilon)\|A - A_k\|_F^2 \tag{1.27}$$

where $0 < \epsilon < 1$ is the accuracy parameter, A_k is the best rank-k approximation of A.

In their paper [15], Boutsidis and Woodruff presented bounds on communication for the column-partition model. They proved that the optimal lower bound is $O(skm)$ for a dense matrix, where s is the number of servers. A sparsity exploiting version of this method was also studied and an error analysis was presented in [70]. The authors studied the error of PCA as low-rank approximation to the original matrix and showed that the approximation error in Frobenius norm is at most $(1+\epsilon)$ times the optimal Frobenius norm when $O(k\epsilon^{-1})$ singular values and singular vectors are computed in the first step.

Besides SVD algorithms, related work has been devoted to communication efficient distributed methods for optimization and machine learning problems, see e.g. [36], [64], [104], [89], [68], [6] and [23].

1.3 Contributions

The central theme of this thesis is to analyse the distributed SVD algorithm of Hauser and Goodman [47]. In contrast to Woodruff and co-authors' approach, our analysis does not stop with approximation guarantees of the computed low-rank approximation of A in matrix norm, but we desire approximation estimations on all singular vectors as a function of the singular spectrum. The main contributions are the following:

- *Establishing two approaches of analysis of the algorithm.*

  We present two novel approaches to analyse the approximation error of the algorithm. The first approach focuses on left leading singular spaces to study the quality of the approximation of the leading left singular space and the SVD as a low-rank approximation to the original matrix. The second approach uses the right singular spaces and provides more geometric insights of the method: while the first analysis studies worst case scenarios, the second approach takes place in a framework with random inputs, and we subsequently validate empirically that the behaviour on non-random input is identical.

- *Numerical experiments along with discussions on variants of the algorithm and practical issues.*

  One of the advantages of the scheme is that it provides great flexibility to users. When the large problems are broken down into smaller problems, users have the choice to distribute the tasks to computational nodes depending on the problem itself and the conditions of the computing resources. Our numerical experiments show that the approximation error of the algorithm is robust under such ad-hoc choices of the computational tree.

  In order to test the algorithm on a diverse set of input, we introduce a model for sampling random matrices with various pattern of singular value separation, and we also use a sparse principal component analysis and a low-rank matrix optimisation model as a means of generating a large number of non-random input matrices that appear in applications.

1.4 Outline of the thesis

This thesis is organised as follows: in Chapter 2, we introduce the algorithm with its geometric motivation and algorithm details. In Chapters 3 and 4, we present two approaches of theoretical analysis of the algorithm. In Chapters 5 and 6, we present numerical experiments on both random and non-random matrices and applications to matrix optimisation problems. In the last chapter, we conclude and discuss future work.

Chapter 2

A Loosely Coupled Parallel SVD Algorithm

In this chapter, we introduce a loosely coupled parallel algorithm for the leading part SVD computations proposed by Hauser and Goodman [47]. The analysis, which will be shown in later chapters, is the main focus of this thesis. It satisfies the loosely coupled requirements of distributed algorithms and is well adapted to distributed computing architectures. This makes it a useful tool for large scale leading part SVD computations with heterogeneous computing resources.

We start from the geometric motivation inherited from classical methods as made clear through the presentation of an evolving sequence of SVD algorithms. The distributed algorithm will be presented at the end of this chapter.

2.1 An evolving sequence of SVD algorithms

Let A be an $m \times n$ real matrix, and let

$$A \approx A_p = U_p S_p V_p^{\mathrm{T}} \tag{2.1}$$

be its p leading part SVD. A_p is a rank-p approximation of A. For simplicity of exhibition, we assume that the singular values of A do not coalesce, i.e.,

$$\sigma_1(A) > \sigma_2(A) > \cdots > \sigma_m(A),$$

although assuming that $\sigma_p(A) > \sigma_{p+1}(A)$ is enough in practice. We denote the range space of V_p by range(V_p). Following the definition of Golub-Van Loan [45], we write

$$\mathrm{dist}(S_1, S_2) = \|P_1 - P_2\|_2$$

for the distance between two subspaces S_1, S_2 of $\mathbb{R}^n$, where P_i is the orthogonal projection onto S_i. If W_i $(i = 1, 2)$ is a Stiefel matrix whose column vectors form the orthogonal basis of space S_i, then $P_i = W_i W_i^{\mathrm{T}}$.

We motivate our algorithm via a sequence of methods that depart from the classical Power method and gradually evolve into our method of choice via successive amendments.

2.1.1 Point of departure

Because range(V_p) is the leading eigenspace of $A^{\mathrm{T}}A$, we can adapt eigen-decomposition methods to find V_p. The Power Method is the simplest and most fundamental method for solving for eigenvalues and eigenvectors. It finds the leading eigenvector through repeatedly multiplying the matrix by a 'nondeficient' normal vector. The direction of the eigenvector q_1 that corresponds to the largest eigenvalue λ_1 is magnified gradually. Because of the relation between the SVD of A and the eigendecomposition of $A^{\mathrm{T}}A$, the Power Method can be adapted for SVD, see Algorithm 2.

Algorithm 2 Power Method for SVD

Input: a unit vector $q^{[0]} \in \mathbb{R}^n$
For $k = 1 : \ell$
  $z = A^{\mathrm{T}} A q^{[k-1]}$
  $q^{[k]} = z / \|z\|$
End for
  $\sigma^{[k]} = \sqrt{(q^{[k]})^{\mathrm{T}} A^{\mathrm{T}} A q^{[k]}}$
Output: $\sigma^{[k]}$ as an approximation to the largest singular value and $q^{[k]}$ as the approximate singular vector.

A number of criteria could be used for terminating the loop instead of using a fixed number of iterations, one of them is the condition

$$\|q^{[k]} - q^{[k-1]}\| < \text{tol},$$

where 'tol' is a tolerance set by the users.

When the initial vector $q^{[0]}$ is chosen in 'general position', the following theorem describes the convergence behaviour of Algorithm 2.

Theorem 2.1 (Wilkinson [103]). *Define* θ_k *as*

$$\cos(\theta_k) = |q_1^{\mathrm{T}} q^{[k]}|,$$

where q_1 is the first eigenvector of $A^{\mathrm{T}}A$. If $q^{[0]}$ is chosen so that

$$\cos(\theta_0) \neq 0,$$

then

$$|\sin(\theta_k)| \leq \tan(\theta_0) \left|\frac{\sigma_2}{\sigma_1}\right|^{2k},$$
$$|(\sigma^{[k]})^2 - (\sigma_1)^2| \leq |(\sigma_1)^2 - (\sigma_m)^2| \tan(\theta_0) \left|\frac{\sigma_2}{\sigma_1}\right|^{2k}.$$

Orthogonal iteration is a straightforward generalisation of the Power Method due to Wilkinson [103]. It can be used to compute higher-dimensional invariant subspaces. It takes an $n \times p$ Stiefel matrix $V^{[0]}$ as an educated guess of V_p and gradually magnifies the subspaces that correspond to the leading singular values through iterations. A QR factorisation is used in orthogonal iteration. The QR factorisation of a matrix Z is given by $Z = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. In this context, we denote the QR factorisation of Z as $[Q, R] = qr(Z)$.

Algorithm 3 Orthogonal Iteration

Input: an $n \times p$ Stiefel matrix $V^{[0]}$
For $k = 1 : \ell$
  $Z = A^{\mathrm{T}}AV^{[k-1]}$
  $[V^{[k]}, \sim] = qr(Z)$
End for
$[U^{[\ell]}, S^{[\ell]}] = qr(AV^{[\ell]})$
Output: $[U^{[\ell]}, S^{[\ell]}, V^{[\ell]}]$ as the approximate SVD of A

The termination criteria can be chosen as,

$$\|V^{[\ell]}(V^{[\ell]})^{\mathrm{T}} - V^{[\ell-1]}(V^{[\ell-1]})^{\mathrm{T}}\|_2 < \mathrm{tol},$$

where 'tol' is a tolerance set by the users. The following theorem due to Wilkinson characterizes the approximation of (U, S, V) after ℓ iterations.

Theorem 2.2 (Wilkinson [103])**.** *If $V^{[0]}$ is chosen so that*

$$\min_{u \in \mathrm{range}(V_p), v \in \mathrm{range}(V^{[0]})} \frac{|u^{\mathrm{T}}v|}{\|u\|_2\|v\|_2} > 0,$$

then

$$\operatorname{dist}(\operatorname{range}(V_p), \operatorname{range}(V^{[\ell]})) \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_p(A)}\right|^{\ell}\right),$$

$$\left|s_{ii}^{[\ell]} - \sigma_i(A)\right| \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_p(A)}\right|^{\ell}\right),$$

$$\operatorname{dist}(\operatorname{range}(U_p), \operatorname{range}(U^{[\ell]})) \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_p(A)}\right|^{\ell}\right).$$

Algorithm 4 Orthogonal Iteration with Ritz Acceleration [91]

Input: an $n \times p$ Stiefel matrix $V^{[0]}$
For $k = 1 : \ell$
  $Z = A^{\mathrm{T}} A V^{[k-1]}$
  $[\tilde{V}^{[k]}, \sim] = qr(Z)$
  $[\sim, \sim, Q] = svd(A\tilde{V}^{[k]})$
  $V^{[k]} = \tilde{V}^{[k]} Q$
End
$V^{[\ell]} \leftarrow V^{[\ell]} Q$
Output: $[U, S, V] = svd(AV^{[\ell]})$

In a seminal article [91], Stewart proposed a variant of Algorithm 3, given in Algorithm 4. Here, a leftarrow $\leftarrow$ means an overwrite. Since $\operatorname{range}(\tilde{V}^{[k]}) = \operatorname{range}(V^{[k]})$, Algorithm 4 operates on the same subspace as Algorithm 3 in each iteration, but the choice of basis of this space is better aligned with the p leading right singular vectors, and this has the consequence of accelerating the convergence. We remark that in exact arithmetic, the basis change inside the loop is not necessary, but in finite precision computations it helps improve the convergence. With this benefit, the SVD step does not increase much of the cost, because $A\tilde{V}^{[k]}$ is only of size $m \times p$. The following theorem is a convergence result for Algorithm 4.

Theorem 2.3 (Stewart [91])**.** *Ritz acceleration improves the estimates of Theorem 2.2 as follows, where $V_{1:i}^{[\ell]}$ consists of the first i columns of $V^{[\ell]}$ and analogously for $U_{1:i}^{[\ell]}$, then*

$$\operatorname{dist}(\operatorname{range}(V_{1:i}), \operatorname{range}(V_{1:i}^{[\ell]})) \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_i(A)}\right|^{\ell}\right),$$

$$\left|s_{ii}^{[\ell]} - \sigma_i(A)\right| \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_i(A)}\right|^{\ell}\right),$$

$$\operatorname{dist}(\operatorname{range}(U_{1:i}), \operatorname{range}(U_{1:i}^{[\ell]})) \leq O\left(\left|\frac{\sigma_{p+1}(A)}{\sigma_i(A)}\right|^{\ell}\right).$$

2.1.2 From Ritz to Lanczos acceleration

The Ritz acceleration idea in Algorithm 4 can be motivated as follows:

Motivation 2.1. *Let $U_p S_p V_p^{\mathrm{T}}$ be the p-leading part SVD of A, and let $W \in \mathbb{R}^{n\times q}$ be a Stiefel matrix with $q \geq p$. Consider the following optimisation problem,*

$$\begin{aligned} \min_{Q\in\mathbb{R}^{n\times p}} \ & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ s.t. \ & Q^{\mathrm{T}}Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(W). \end{aligned} \tag{2.2}$$

An approximation $\hat{Q}$ of the minimiser Q^* of this problem is obtained by $\hat{Q} = WP$, where P is the p-leading right singular matrix of AW, i.e. $[\sim, \sim, P] = svds(AW, p)$.

Here, as elsewhere in the text, we use Matlab notation by referring to the p-leading part SVD as $svds(\cdot, p)$. A detailed discussion on this geometric motivation will be presented in Chapter 4.

On the basis of the geometric motivation, we can now amend the idea of Ritz acceleration to a similar framework in which additional dimensions are included in the search space in each iteration. We call this type of acceleration *Lanczos acceleration* because it corresponds to a cropped block-Lanczos method in which only the salient $q = 2p$ dimensions of the Krylov subspace are kept, the rest being discarded, so as to keep the complexity of each iteration constant.

The input is an educated guess $Q^{[0]}$ of a $n \times 2p$ Stiefel matrix that would ideally satisfy $\mathrm{range}(V_p) \subset \mathrm{range}(Q^{[0]})$.

Algorithm 5 Orthogonal Iteration with Lanczos Acceleration

Input: an $n \times 2p$ Stiefel matrix $Q^{[0]}$
For $k = 1 : \ell$
  $Z = AQ^{[k-1]}$
  $[U^{[k]}, S^{[k]}, P] = svds(Z, p)$
  $V^{[k]} = Q^{[k-1]}P$
  $[Q^{[k]}, \sim] = qr(\begin{bmatrix} V^{[k]}, & A^{\mathrm{T}}U^{[k]} \end{bmatrix})$
End for
Output: $[U, S, V] = svd(AV^{[\ell]})$

The matrix that appears in the argument of the QR factorization on the last line of the loop is written in block form. The work per iteration of Algorithms 4 and 5 is exactly the same. Lanczos acceleration is at least as fast as Ritz acceleration. The

extra acceleration pays off in the initial phase of the algorithm; in the final phase Lanczos acceleration behaves like Ritz acceleration with a time lag of one iteration.

Algorithm 5 is not parallel. However, the idea of extracting optimal subspaces (and an orthogonal basis thereof as a coordinate representation) from somewhat larger subspaces will be recycled in a cascadic fashion.

2.1.3 The framework of the distributed SVD algorithm

In order to progress toward a parallel algorithm, we now describe three types of node computations from which such an algorithm will be composed.

i) Seed nodes: original data of the matrix A enters the system in column-split form. An approximate leading right-singular space is extracted from the local data.

ii) Combination nodes: two or more sets of approximate leading right singular spaces are added together, and an approximate leading space is extracted from the combined space.

iii) Extraction node: a node in which the full p-leading part SVD is extracted from the local data.

Here, a node could be a core in a multi-core processor or a CPU in a computer cluster. A simple instantiation of a work flow is shown in Figure 2.1. The black nodes at the top of the diagram act as seed nodes. The results of their local computations are combined at the white combination nodes, and finally the grey node at the bottom of the diagram represents a combination node followed by an extraction node. We refer to one such flow from top to bottom as a *sweep*, as we will subsequently discuss multi-sweep versions.

More general work flow patterns may be used in which any node that has recently completed their local computations combine their output data at a combination node in an ad-hoc fashion, thus running asynchronously. See Figure 2.2 for a depiction of such a work flow.

In the next sections, we give specifications of the three types of nodes.

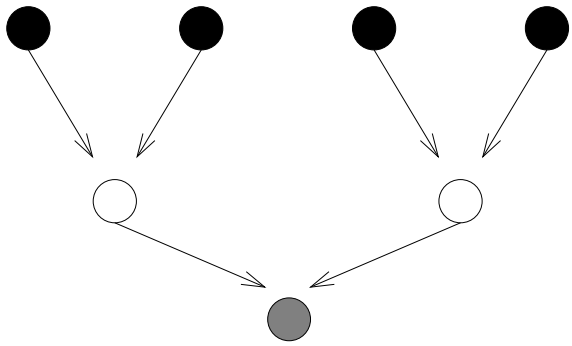

Figure 2.1: A simple instantiation of a work flow on a binary recombining tree.

2.1.4 Specification of seed nodes

We initialise the algorithm in seed nodes. The inputs of a seed node are a pair of matrices $(AQ_{\text{in}}, Q_{\text{in}})$, where Q_{in} is a Stiefel matrix of size $n \times k$, with $k > p$. We use this data structure to represent the subspace range(Q_{in}) of $\mathbb{R}^n$, with a coordinate representation given by an orthogonal basis, and the action of the linear operator A restricted to this subspace as expressed in this coordinate system. The total size of this data is $(m + n)k$.

In local computations, we find an approximate solution of the optimisation problem

$$\begin{aligned} Q_{\text{out}} \approx \arg \min_{Q \in \mathbb{R}^{n \times p}} & \operatorname{dist}(\operatorname{range}(V_p), \operatorname{range}(Q)) \\ s.t. & Q^{\mathrm{T}} Q = I_p, \\ & \operatorname{range}(Q) \subset \operatorname{range}(Q_{\text{in}}). \end{aligned}$$

Using Motivation 2.1, the approximation is carried out by applying the following steps:

i) $[\sim, \sim, P] = svds(AQ_{\text{in}}, p)$,

ii) $Q_{\text{out}} = Q_{\text{in}} P$,

iii) $AQ_{\text{out}} = AQ_{\text{in}} P$.

In the first step, one can use any method to evaluate $svds(\cdot)$. When the size of the local data is small enough such that it can be held in local memory, any conventional method can be used. When the data is so large that it cannot be held in local memory, the parallel algorithm we will describe below can be applied at a second, recursive

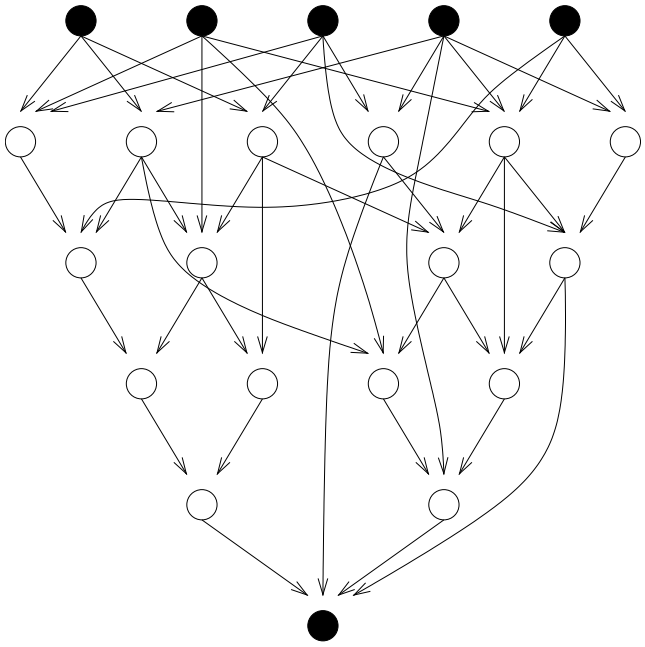

Figure 2.2: A general work flow pattern.

layer to the transpose of AQ_{in}. Stiefel matrix P is of size $k \times p$, AQ_{out} is of size $m \times p$, and Q_{out} is of size $n \times p$.

The complexity of the first step of leading part SVD computation depends on the choice of local SVD method. If classical methods are used, the typical complexity is $O(pmn)$. The complexity of the second and the third step is $O(nkp)$ and $O(mkp)$, respectively. Since $p \ll \min(m, n)$, the complexity of local computations is very small compared with the dimension of the problem.

The output matrices $(AQ_{\text{out}}, Q_{\text{out}})$ have the same form as the input, but Q_{out} is now a Stiefel matrix of size $n \times p$ that forms an orthogonal basis of the p-dimensional leading subspace of range(Q_{out}), while AQ_{out} represents the action of the linear operator A restricted to this subspace and with respect to the chosen basis. The total size of the output data is $(m + n)p$.

2.1.5 Seeding the seed nodes

The easiest way to generate the input for all seed nodes is to split A into sets of columns. Formally, this can be seen as generating the input data $(AQ^{[i]}, Q^{[i]})$ in the

following fashion: choose b integers $\ell_1, \ldots, \ell_b$ such that $\sum_i \ell_i = n$ and set

$$Q^i = \begin{bmatrix} 0 \\ \mathrm{I}_{\ell_i} \\ 0 \end{bmatrix},$$

and

$$AQ^i = A\left(:, \sum_{t=1}^{i-1} \ell_t + 1 : \sum_{t=1}^{i} \ell_t\right).$$

In many applications, the column-wise splitting occurs naturally when the data for different sets of columns are computed in parallel. When the natural splitting is by blocks of rows, we apply the method to A^{T}.

One advantage of this setting is that the algorithm can be started even before all the column data is assembled or computed. When a block of columns is added later, the computation can incorporate the new data to update the leading subspaces already identified.

2.1.6 Specification of combination nodes

The inputs of a combination node come from outputs of seed nodes or other combination nodes. They are $b \geq 2$ pair of matrices $(AQ_{\mathrm{in}}^{[j]}, Q_{\mathrm{in}}^{[j]})$ $(j = 1, \ldots, b)$, where $Q_{\mathrm{in}}^{[j]}$ are $n \times p_j$ Stiefel matrices.

Similar to seed nodes, local computations in combination nodes find an approximate solution of optimisation problem,

$$\begin{aligned} Q_{\mathrm{out}} \approx \arg \min_{Q \in \mathbb{R}^{n \times p}} & \operatorname{dist}(\operatorname{range}(V_p), \operatorname{range}(Q)) \\ s.t. & Q^{\mathrm{T}} Q = \mathrm{I}_p, \\ & \operatorname{range}(Q) \subset \sum_{j=1}^{b} \operatorname{range}(Q_{\mathrm{in}}^{[j]}). \end{aligned}$$

Here $\sum_{j=1}^{b} \operatorname{range}(Q_{\mathrm{in}}^{[j]})$ is the sum of the spaces $\operatorname{range}(Q_{\mathrm{in}}^{[j]})$ $(j = 1, \ldots, b)$.

Using Motivation 2.1, the approximation is carried out by applying the following algorithm:

i) $[\tilde{U}, \tilde{S}, \tilde{V}] = svds([\, AQ_{\mathrm{in}}^{[1]} \ \ldots \ AQ_{\mathrm{in}}^{[b]} \,], p)$,

ii) $[Q_{\mathrm{out}}, R] = qr([\, Q_{\mathrm{in}}^{[1]} \ \ldots \ Q_{\mathrm{in}}^{[b]} \,]\tilde{V})$,

iii) $AQ_{\mathrm{out}} = [\, AQ_{\mathrm{in}}^{[1]} \ \ldots \ AQ_{\mathrm{in}}^{[b]} \,]\tilde{V}R^{-1}$.

An alternative equivalent approach is to carry out the following calculation:

i) $[\tilde{U}, \tilde{S}, \tilde{V}] = svds(\left[\begin{smallmatrix} AQ_{\text{in}}^{[1]} & \dots & AQ_{\text{in}}^{[b]} \end{smallmatrix}\right], p)$,

ii) $\begin{bmatrix} \tilde{Q}^{[1]} \\ \vdots \\ \tilde{Q}^{[b]} \end{bmatrix} := \begin{bmatrix} Q_{\text{in}}^{[1]} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & Q_{\text{in}}^{[b]} \end{bmatrix} \tilde{V},$

iii) $[Q_{\text{out}}, R] = qr(\tilde{Q}^{[1]} + \cdots + \tilde{Q}^{[b]})$,

vi) $AQ_{\text{out}} = \begin{bmatrix} AQ_{\text{in}}^{[1]} & \dots & AQ_{\text{in}}^{[b]} \end{bmatrix} \tilde{V} R^{-1}.$

The outputs are $(AQ_{\text{out}}, Q_{\text{out}})$. In exact arithmetic, the outputs and complexity of the two approaches for the local computation are identical.

2.1.7 Specification of extraction nodes

In the extraction node, the approximate leading right singular space or the approximate SVD is extracted. The input $(AQ_{\text{in}}, Q_{\text{in}})$ is the output of a combination node.

The local computations is the following:

i) $[\tilde{U}, \tilde{S}, P] = svds(AQ_{\text{in}}, p)$,

ii) $\tilde{V} = Q_{\text{in}} P$,

iii) $A\tilde{V} = AQ_{\text{in}} P$.

The output is $(A\tilde{V}, \tilde{V})$ if another sweep is taken; or $(\tilde{U}, \tilde{S}, \tilde{V})$ an approximate p-leading part SVD of A in the case where no further sweeps are carried out.

2.2 A mostly loosely coupled algorithm

The above built elements can be used to construct Algorithm 6, a parallel algorithm that is loosely coupled except at the start of each sweep, where synchronisation is necessary in the function initializeNextSweep. Its outer iteration complexity is at most the iteration complexity of Orthogonal Iteration with Ritz acceleration, as the latter dominates the former, just as in the strongly coupled case.

Algorithm 6 A mostly loosely coupled algorithm

Initialisation:

1) Get seed input data (as described in Section 2.1.5 or educated guess $(\tilde{U}, \tilde{S}, \tilde{V})$)
2) While seed node input data-sets available
   launch seed node
end while
3) Until all data combined into a single data-set $(AQ_{\text{out}}, Q_{\text{out}})$
   launch combination node
end until
4) Launch extraction node and replace $(\tilde{U}, \tilde{S}, \tilde{V})$ by new values

Main Loop:

While $\|\tilde{V}\tilde{V}^{\mathrm{T}} - \tilde{V}_{-}\tilde{V}_{-}^{\mathrm{T}}\|_2 >$ tol, where $\tilde{V}_{-}$ denotes value before last update
   1) Call function initializeNextSweep($\tilde{U}$)
   2) Until all data combined into a single data-set $(AQ_{\text{out}}, Q_{\text{out}})$
      launch combination node
   end until
   3) Launch extraction node and replace $(\tilde{U}, \tilde{S}, \tilde{V})$ by new values
end while

function initializeNextSweep($\tilde{U}$)

1) For $j = 1, \ldots, k$ (for all seed node inputs)
   $W^{[j]} = \tilde{U}^{\mathrm{T}} A Q_{\text{in}}^{[j]}$
end for
2) $[Q, R] = qr([\, W^1 \ldots W^k \,]^{\mathrm{T}})$
3) For $j = 1, \ldots, k$
   $Z^j = (AQ_{\text{in}}^j)(W^j)^{\mathrm{T}} R^{-1}$
end for
4) $AQ = \sum_{j=1}^{k} Z^j$
5) For $j = 1, \ldots, k$
   launch combination node with input $(AQ_{\text{in}}^j, Q_{\text{in}}^j)$, (AQ, Q)
end for

2.3 A loosely coupled SVD heuristic

The synchronisation in the initialization of the next sweep of Algorithm 6 is required to compute $(AQ_{\text{in}}^j, Q_{\text{in}}^j)$, where

$$[Q_{\text{in}}^j, \sim] = qr(A^{\text{T}}\tilde{U}).$$

Note that if it was the case that $A = \tilde{U}\tilde{S}\tilde{V}^{\text{T}}$, then we would have

$$A^{\text{T}}\tilde{U} = \tilde{V}\tilde{S},$$
$$Q = \tilde{V}.$$

In reality, $A \approx \tilde{U}\tilde{S}\tilde{V}^{\text{T}}$ when A is nearly rank deficient, so that we can use

$$(A\tilde{V}, \tilde{V}) \approx (AQ, Q)$$

as an approximation that can be obtained without synchronisation. To obtain a loosely coupled heuristic version of Algorithm 6, we thus only need to change the function initializeNextSweep.

Chapter 3

Convergence Analysis Via Left Singular Spaces

Our first approach to a convergence analysis of the parallel SVD algorithm investigates the quality of the approximate leading part SVD after a single sweep with the standard binary combination workflow. We will derive the following bound of the approximation error in the left leading singular space,

$$\|A - \breve{U}\breve{U}^{\mathrm{T}}A\|_2 \leq \sqrt{2s-1}\sigma_{p+1}(A), \tag{3.1}$$

where $\breve{U}$ is the matrix composed of the approximate left leading singular vectors calculated by a binary combination version of the parallel algorithm which is described in Algorithm 10 in this chapter, s is the number of seed nodes and $\sigma_{p+1}(A)$ is the $(p+1)$-th singular value of A.

This result reveals a measure of the closeness of the range space of $\breve{U}$ to the left leading singular space U_p, since $\|A - U_pU_p^{\mathrm{T}}A\|_2 \leq \sigma_{p+1}(A)$ and U_p minimises $\|A - UU^{\mathrm{T}}A\|_2$ among all Stiefel matrices of size $n \times p$. Another interpretation of (3.1) is that it gives a bound of the error on the leading part SVD as a low-rank approximation of the original matrix.

3.1 Introduction

Let A be an $m \times n$ real matrix. Given a positive integer $p \ll \min(m,n)$, our aim is to compute the p leading part SVD of A,

$$A \approx A_p = U_pS_pV_p^{\mathrm{T}}.$$

We assume that the p-th and $(p+1)$-th singular value of A do not coalesce. In the case where the p-th and $(p+1)$-th singular value of A are coalescing, the p-leading

singular subspace is not unique. In this case the algorithm returns an approximation of one of the p-leading singular subspaces.

We study a 'standard' version of the parallel algorithm, detailed in Algorithm 10. By 'standard', we mean the following:

- The method follows the binary combination tree illustrated in Figure 2.1.
- n is divisible by $s = 2^q$, where s is number of seed nodes and $q+1$ is the number of levels of the binary execution tree. The columns are partitioned equally into 2^q submatrices of dimension $m \times \frac{n}{2^q}$. Each submatrix is distributed into one of the seed nodes.

We make these assumptions for convenience of the analysis that follows. In practice, users do not need to adhere to these ways of initialisation and combination. Variations of the algorithm will be discussed in Chapter 5.

The output matrix $\breve{U}$ gives an approximation of the p-leading left singular vectors. We derive an upper bound on the difference between A and its projection into the space generated by column vectors of $\breve{U}$, as stated in Theorem 3.1.

Theorem 3.1. *Let A be a given $m \times n$ matrix, and let*

$$A \approx \breve{U}\breve{S}\breve{V}^{\mathrm{T}} \tag{3.2}$$

be its approximate p-leading SVD computed by Algorithm 10. Then

$$\|A - \breve{U}\breve{U}^{\mathrm{T}}A\|_2 \leq \sqrt{2s-1}\sigma_{p+1}(A). \tag{3.3}$$

The error bound on the right hand side of (3.3) is given in terms of the number of seed nodes and the $(p+1)$-th singular value of A. In the special case where there is only one node, i.e., $s = 1$, the inequality (3.3) becomes the same as the classical result [45]:

$$\min_{D \in \mathbb{R}^{m\times n}, \mathrm{rank}(D) \leq p} \|A - D\|_2 = \|A - U_p S_p V_p^{\mathrm{T}}\|_2 = \sigma_{p+1}(A). \tag{3.4}$$

When there is only one seed node, the whole problem is the only subproblem. It is equivalent to computing the SVD directly and accurately. Therefore, when $s = 1$, the bound is tight.

The leading part SVD is frequently used as a low-rank approximation of matrices. In low-rank matrix approximation problems, we aim at solving the following problem,

$$\min_{D \in \mathbb{R}^{m\times n}, \mathrm{rank}(D) \leq p} \|A - D\|_2. \tag{3.5}$$

Given a fixed rank p, we want to find the closest matrix in terms of the 2-norm, that has rank smaller than or equal to p. The p leading part SVD yields an optimal solution $A_p = U_p S_p V_p^{\mathrm{T}}$ to this problem, with optimal objective value $\sigma_{p+1}(A)$.

Therefore, when the first p leading left singular vectors are known, the minimizer can be constructed by

$$D^* = U_p U_p^{\mathrm{T}} A. \tag{3.6}$$

Thus, when $\breve{U}$, an approximation of the p-leading left singular vectors of A is computed, it is therefore of interest to study the difference between A and the low-rank approximation $\breve{U}\breve{U}^{\mathrm{T}}A$.

The remainder of this chapter is organised as follows: In Section 3.2, we give a brief introduction of a binary combination version of the algorithm, with notation introduced. The analysis of errors and the proof of Theorem 3.1 are presented in Section 3.3. In the last section, we discuss the tightness of the error bound with examples and numerical tests.

3.2 Algorithm details

We study a standard binary combination version of Algorithm 10, which we will now describe again to introduce the notation we will use in the analysis. As seen later, the algorithm uses three different kinds of nodes: seed nodes, combination nodes and an extraction node. In seed nodes, column-splitting is used to initialise the algorithm. In the variant of the algorithm studied here, two sets of outputs from seed nodes or combination nodes are used to form the input of a new combination node. Thus, the data flow follows a binary combining tree, which we call the execution tree.

To initialise the algorithm, we split the columns of the matrix A and distribute the blocks of columns to the seed nodes, as stated in Section 2.1.5. If the execution tree has $q+1$ levels, that is to say, it has $s = 2^q$ seed nodes, we split A as follows.

$$A = [A_{1,1}, A_{1,2}, \ldots, A_{1,2^q}], \tag{3.7}$$

where $A_{1,i}$, $i = 1, 2, \ldots, 2^q$, is a submatrix of a set of $\frac{n}{2^q}$ columns of A. By assumption, n is divisible by 2^q and the columns are equally distributed. Thereafter, we compute the SVD of $A_{1,i}$. The details of the computations in seed nodes are given in Algorithm 7.

We number the nodes as follows: a node is numbered $(k,\ i)$ if it is the i-th node on the k-th level of the execution tree. In a seed node $(1, i)$, where $i = 1, \ldots, 2^q$, a

Algorithm 7 Seed nodes $(1, i)$

0) Input: $A_{1,i} \in \mathbb{R}^{m\times(n/2^q)}$
1) Compute the p-leading SVD: $A_{1,i} \approx U_{1,i}S_{1,i}V_{1,i}^{\mathrm{T}}$
2) Output: $A_{l,j}^2 = U_{1,i}S_{1,i}$, $V_{l,j}^2 = V_{1,i}$, where $l = (i+1)/2$, $j = 1$ if $i \equiv 1(\mathrm{mod}\ 2)$; $l = i/2$, $j = 2$, otherwise.

pair of outputs $(A_{l,j}^2, V_{l,j}^2)$, where $l = (i+1)/2$ and $j = 1$ if $i \equiv 1(\mathrm{mod}\ 2)$; $l = i/2$ and $j = 2$ otherwise, are used as inputs in combination nodes $(2, l)$. At combination nodes $(2, i)$, we concatenate $A_{i,1}^2$ and $A_{i,2}^2$ to form $A_{2,i}$. Block diagonal matrix $\hat{V}_{2,i}$ has diagonal $V_{i,1}^2$ and $V_{i,2}^2$. Similarly, at combination nodes (k, i), $A_{k,i}$ is formed by the concatenation of $A_{i,1}^k$ and $A_{i,2}^k$. Matrix $\hat{V}_{2,i}$ is a block diagonal matrix whose diagonal is $V_{i,1}^k$ and $V_{i,2}^k$. Hence, we have the equalities

$$A_{k,i} = (A_{i,1}^k, A_{i,2}^k) \tag{3.8}$$

and

$$\hat{V}_{k,i} = \begin{pmatrix} V_{i,1}^k & 0 \\ 0 & V_{i,2}^k \end{pmatrix}. \tag{3.9}$$

Algorithm 8 Combination nodes (k, i)

0) Input: $A_{i,1}^k$, $A_{i,2}^k$, $V_{i,1}^k$ and $V_{i,2}^k$
1) Let $A_{k,i} = (A_{i,1}^k, A_{i,2}^k)$, $\hat{V}_{k,i} = \begin{pmatrix} V_{i,1}^k & 0 \\ 0 & V_{i,2}^k \end{pmatrix}$
2) Compute the p-leading part SVD: $A_{k,i} \approx U_{k,i}S_{k,i}(V_{k,i})^{\mathrm{T}}$
3) Compute thin QR factorization: $\hat{V}_{k,i}V_{k,i} = Q_{k,i}R_{k,i}$
4) Output: $A_{l,j}^{k+1} = U_{k,i}S_{k,i}(R_{k,i})^{-1}$, $V_{l,j}^{k+1} = Q_{k,i}$, where $l = (i+1)/2$, $j = 1$ if $i \equiv 1(\mathrm{mod}\ 2)$; $l = i/2$ $j = 2$, otherwise.

The details of the computations in combination nodes (k, i) are given in Algorithm 8. We note that a thin-QR factorisation of an $m \times n$ matrix B with $m \geq n$ is defined as,

$$B = QR,$$

where Q is an $m \times n$ Stiefel matrix and R is a $n \times n$ upper triangular matrix.

The column vectors of the output matrix $V_{l,j}^{k+1}$ form a basis of an approximation of the p-leading right singular space, which gets refined through the local computations at node (k, i). While the leading right singular space approaches the exact p leading right singular subspace of A, $A_{l,j}^{k+1}$ keeps the information that is needed to construct the approximate leading SVD in the extraction node.

The extraction node takes as input the output of the last combination nodes and extracts the approximate SVD of A by executing additional steps:

- $\breve{A} = U_{q+1,1} S_{q+1,1} (R_{q+1,1})^{-1}$
- Compute the p-leading part SVD: $\breve{A} = \breve{U}\breve{S}\bar{V}^{\mathrm{T}}$
- Compute $\breve{V} = Q_{q+1,1}\bar{V}$

Outputs $[\breve{U}, \breve{S}, \breve{V}]$ are returned as the approximate SVD of A. The details of an extraction node are given in Algorithsm 9.

Algorithm 9 Extraction nodes $(q+1, 1)$

0) Input: $A_{1,1}^{q+1}$, $A_{1,2}^{q+1}$, $V_{q,1}^{q+1}$ and $V_{1,2}^{q+1}$
1) Let $A_{q+1,1} = (A_{1,1}^{q+1}, A_{1,2}^{q+1})$, $\hat{V}_{q+1,1} = \begin{pmatrix} V_{1,1}^{q+1} & 0 \\ 0 & V_{1,2}^{q+1} \end{pmatrix}$
2) Compute the p-leading part SVD: $A_{q+1,1} \approx U_{q+1,1} S_{q+1,1} (V_{q+1,1})^{\mathrm{T}}$
3) Compute thin QR factorization: $\hat{V}_{q+1,1} V_{q+1,1} = Q_{q+1,1} R_{q+1,1}$
4) $\breve{A} = U_{q+1,1} S_{q+1,1} (R_{q+1,1})^{-1}$.
5) Compute the p-leading part SVD: $\breve{A} = \breve{U}\breve{S}\bar{V}^{\mathrm{T}}$.
6) $\breve{V} = Q_{q+1,1}\bar{V}$
7) Output: $[\breve{U}, \breve{S}, \breve{V}]$

Figure 3.2 gives an illustration of an execution tree with $q = 4$ levels and $s = 8$ seed nodes. A full description of the framework of the algorithm is given in Algorithm 10.

Algorithm 10 A binary combination version of the distributed SVD algorithm

0) Input: $A \in \mathbb{R}^{m \times n}$
Perform the binary tree computations by invoking
1) Algorithm 7 on seed nodes $(1, i)$, $i = 1, \ldots, 2^q$
2) Algorithm 8 on combination nodes (k, i), $k = 2, \ldots, q$ and $i = 1, \ldots, 2^{q-k+1}$
3) Algorithm 9 on extraction nodes $(q+1, 1)$
4) Output: $[\breve{U}, \breve{S}, \breve{V}]$

3.3 Proof of Theorem 3.1

In this section, we present a proof of Theorem 3.1. Let us start from a lemma which gives an upper bound of $\|A - \tilde{U}\tilde{U}^{\mathrm{T}}A\|$ for any $m \times p$ Stiefel matrix $\tilde{U}$.

Lemma 3.1. *Let A be an $m \times n$ matrix and $\tilde{U}$ an $m \times p$ Stiefel matrix. Let $\tilde{\mathcal{H}} =$ range$(\tilde{U})$ and $\tilde{\mathcal{H}}^{\perp}$ the orthogonal complement of $\tilde{\mathcal{H}}$. Then,*

$$\|A - \tilde{U}\tilde{U}^{\mathrm{T}}A\|_2 \leq \max_{z \in \tilde{\mathcal{H}}^{\perp}, \|z\|=1} \|z^{\mathrm{T}}A\|. \tag{3.10}$$

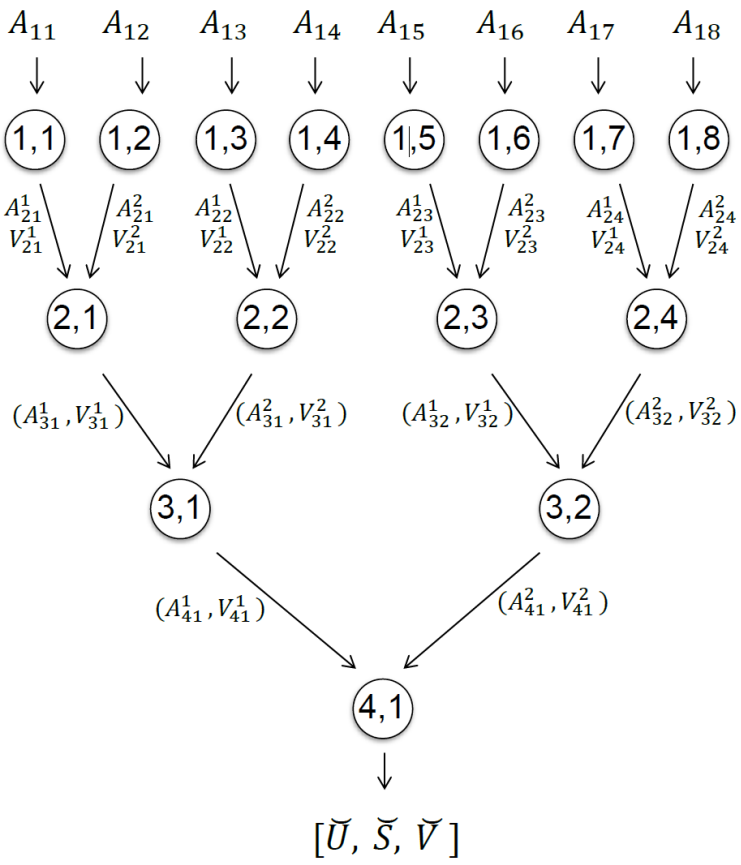

Figure 3.1: An illustration of an execution tree with 8 seed nodes.

Proof. For any unit vector $x \in \mathbb{R}^m$, there exists a decomposition $x = \alpha y + \beta z$, where $y \in \tilde{\mathcal{H}}$ and $z \in \tilde{\mathcal{H}}^\perp$ are unit vectors, and $\alpha, \beta \in \mathbb{R}$ such that $\alpha^2 + \beta^2 = 1$. Therefore,

$$\begin{aligned}
\|A - \tilde{U}\tilde{U}^{\mathrm{T}}A\|_2 &= \max_{x\in\mathbb{R}^m, \|x\|=1} \|x^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2 \\
&\leq \max_{y\in\tilde{\mathcal{H}}, \|y\|=1, z\in\tilde{\mathcal{H}}^\perp, \|z\|=1, \alpha^2+\beta^2=1} |\alpha| \|y^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2 \\
&\quad + |\beta| \|z^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2 \\
&\leq \max_{y\in\tilde{\mathcal{H}}, \|y\|=1} \|y^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2 + \max_{z\in\tilde{\mathcal{H}}^\perp, \|z\|=1} \|(z^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2.
\end{aligned}$$

The first equality on the first line follows the definition of the 2-norm of a matrix and using the fact that $\|M\|_2 = \|M^{\mathrm{T}}\|_2$ for any matrix M. The inequality on the second line derives from the triangle inequality of vector norms, and the last inequality holds since $|\alpha|, |\beta| \leq 1$.

The first term $\max_{y\in\tilde{\mathcal{H}}, |y|=1} \|y^{\mathrm{T}}(A - \tilde{U}\tilde{U}^{\mathrm{T}}A)\|_2$ is zero because y is in the space generated by the columns of $\tilde{U}$. The projection of y onto the space $\tilde{\mathcal{H}}$ is the same as y. Hence, $y^T(A - \tilde{U}\tilde{U}^{\mathrm{T}}A) = 0$ for all $y \in \tilde{\mathcal{H}}$. In the second term, $z^{\mathrm{T}}\tilde{U}\tilde{U}^{\mathrm{T}}A$ is zero because z is orthogonal to range($\tilde{U}$). Therefore, (3.10) holds. □

In this lemma, we proved an upper bound on $\|A - \tilde{U}\tilde{U}^{\mathrm{T}}A\|_2$. In the following lemmas, we will take $\breve{U}$ as $\tilde{U}$. For $\breve{U}$, we will give a bound for the right hand side of the inequality in Lemma 3.1 in terms of the $(p+1)$-th singular values of some intermediate matrices in the algorithm.

Before continuing, we review a theorem about the p-leading SVD.

Theorem 3.2. *[45] If $A \approx U_p S_p V_p^{\mathrm{T}}$ is the p-leading SVD of A, then*

$$\|AA^{\mathrm{T}} - U_p S_p^2 U_p^{\mathrm{T}}\|_2 = \sigma_{p+1}^2(A) \tag{3.11}$$

The following two lemmas are preparations for the proofs of later results. In the first lemma, we focus on the combination nodes on the second level. The second lemma is an analogy of other combination nodes and the extraction node.

Lemma 3.2. *At combination node $(2, i), i = 1, \ldots, 2^{q-1}$, we have*

$$\|[A_{1,2i-1}, A_{1,2i}][A_{1,2i-1}, A_{1,2i}]^{\mathrm{T}} - A_{2,i}(A_{2,i})^{\mathrm{T}}\|_2 \leq \sigma_{p+1}^2(A_{1,2i-1}) + \sigma_{p+1}^2(A_{1,2i}). \tag{3.12}$$

Proof.

$$
\begin{aligned}
& \|[A_{1,2i-1}, A_{1,2i}][A_{1,2i-1}, A_{1,2i}]^{\mathrm{T}} - A_{2,i}(A_{2,i})^{\mathrm{T}}\|_2 \\
= \; & \|[A_{1,2i-1}, A_{1,2i}][A_{1,2i-1}, A_{1,2i}]^{\mathrm{T}} - [A_{i,1}^2, A_{i,2}^2][A_{i,1}^2, A_{i,2}^2]^{\mathrm{T}}\|_2 \\
= \; & \|[A_{1,2i-1}, A_{1,2i}][A_{1,2i-1}, A_{1,2i}]^{\mathrm{T}} \\
& -[U_{1,2i-1}S_{1,2i-1}, U_{1,2i}S_{1,2i}][U_{1,2i-1}S_{1,2i-1}, U_{1,2i}S_{1,2i}]^{\mathrm{T}}\|_2 \\
= \; & \|A_{1,2i-1}(A_{1,2i-1})^{\mathrm{T}} + A_{1,2i}(A_{1,2i})^{\mathrm{T}} \\
& -U_{1,2i-1}(S_{1,2i-1})^2(U_{1,2i-1})^{\mathrm{T}} - U_{1,2i}(S_{1,2i})^2(U_{1,2i})^{\mathrm{T}}\|_2 \\
\leq \; & \|A_{1,2i-1}(A_{1,2i-1})^{\mathrm{T}} - U_{1,2i-1}(S_{1,2i-1})^2(U_{1,2i-1})^{\mathrm{T}}\|_2 \\
& +\|A_{1,2i}(A_{1,2i})^{\mathrm{T}} - U_{1,2i}(S_{1,2i})^2(U_{1,2i})^{\mathrm{T}}\|_2 \\
= \; & \sigma_{p+1}^2(A_{1,2i-1}) + \sigma_{p+1}^2(A_{1,2i}).
\end{aligned}
$$

The first equality follows from (3.8). We obtain the second and third equality by simple substitutions and calculations. The inequality is a result of the triangle equality of norms, and the last equality follows Theorem 3.2 □

Lemma 3.3. *At combination node* $(k, i), k = 2, \ldots, q, i = 1, \ldots, 2^{q-k+1}$*, we have*

$$\|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} - A_{k+1,i}(A_{k+1,i})^{\mathrm{T}}\|_2 \leq \sigma_{p+1}^2(A_{k,2i-1}) + \sigma_{p+1}^2(A_{k,2i}).$$

Proof.

$$
\begin{aligned}
& \|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} - A_{k+1,i}(A_{k+1,i})^{\mathrm{T}}\|_2 \\
= \; & \|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} - [A_{i,1}^{k+1}, A_{i,2}^{k+1}][A_{i,1}^{k+1}, A_{i,2}^{k+1}]^{\mathrm{T}}\|_2 \\
= \; & \|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} \\
& -[A_{i,1}^{k+1}(V_{i,1}^{k+1})^{\mathrm{T}}, A_{i,2}^{k+1}(V_{i,2}^{k+1})^{\mathrm{T}}][A_{i,1}^{k+1}(V_{i,1}^{k+1})^{\mathrm{T}}, A_{i,2}^{k+1}(V_{i,2}^{k+1})^{\mathrm{T}}]^{\mathrm{T}}\|_2 \\
= \; & \|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} \\
& -[U_{k,2i-1}S_{k,2i-1}(R_{k,2i-1})^{-1}(Q_{k,2i-1})^{\mathrm{T}}, U_{k,2i}S_{k,2i}(R_{k,2i})^{-1}(Q_{k,2i})^{\mathrm{T}}] \\
& [U_{k,2i-1}S_{k,2i-1}(R_{k,2i-1})^{-1}(Q_{k,2i-1})^{\mathrm{T}}, U_{k,2i}S_{k,2i}(R_{k,2i})^{-1}(Q_{k,2i})^{\mathrm{T}}]^{\mathrm{T}}\|_2 \\
= \; & \|[A_{k,2i-1}, A_{k,2i}][A_{k,2i-1}, A_{k,2i}]^{\mathrm{T}} \\
& -[U_{k,2i-1}S_{k,2i-1}(V_{k,2i-1})^{\mathrm{T}}(\hat{V}_{k,2i-1})^{\mathrm{T}}, U_{k,2i}S_{k,2i}(V_{k,2i})^{\mathrm{T}}(\hat{V}_{k,2i})^{\mathrm{T}}] \\
& [U_{k,2i-1}S_{k,2i-1}(V_{k,2i-1})^{\mathrm{T}}(\hat{V}_{k,2i-1})^{\mathrm{T}}, U_{k,2i}S_{k,2i}(V_{k,2i})^{\mathrm{T}}(\hat{V}_{k,2i})^{\mathrm{T}}]^{\mathrm{T}}\|_2
\end{aligned}
$$

$$
\begin{aligned}
&= \|A_{k,2i-1}(A_{k,2i-1})^{\mathrm{T}} + A_{k,2i}(A_{k,2i})^{\mathrm{T}} - U_{k,2i-1}(S_{k,2i-1})^2(U_{k,2i-1})^{\mathrm{T}} \\
&\quad -U_{k,2i}(S_{k,2i})^2(U_{k,2i})^{\mathrm{T}}\|_2 \\
&\leq \|A_{k,2i-1}(A_{k,2i-1})^{\mathrm{T}} - U_{k,2i-1}(S_{k,2i-1})^2(U_{k,2i-1})^{\mathrm{T}}\|_2 \\
&\quad +\|A_{k,2i}(A_{k,2i})^{\mathrm{T}} - U_{k,2i}(S_{k,2i})^2(U_{k,2i})^{\mathrm{T}}\|_2 \\
&= \sigma_{p+1}^2(A_{k,2i-1}) + \sigma_{p+1}^2(A_{k,2i}).
\end{aligned}
$$

The first equality follows step 1 of Algorithm 8. The second equality holds for the reason that $V_{i,1}^{k+1}$ and $V_{i,2}^{k+1}$ are Stiefel matrices. In the third equality, we substitute $A_{i,1}^{k+1}$ and $A_{i,2}^{k+1}$ by $U_{k,2i-1}S_{k,2i-1}(R_{k,2i-1})^{-1}$ and $U_{k,2i}S_{k,2i}(R_{k,2i})^{-1}$, as well as $V_{i,1}^{k+1}$ and $V_{i,2}^{k+1}$ by $Q_{k,2i-1}$ and $Q_{k,2i}$.

In the thin QR factorisation step, we have $Q_{k,2i-1}R_{k,2i-1} = \hat{V}_{k,2i-1}V_{k,2i-1}$. Since both $\hat{V}_{k,2i-1}$ and $V_{k,2i-1}$ are Stiefel matrices and $R_{k,2i-1}$ is the $p \times p$ identity matrix under exact arithmetic, we have

$$
(R_{k,2i-1})^{-1}(Q_{k,2i-1})^{\mathrm{T}} = (Q_{k,2i-1})^{\mathrm{T}} = (V_{k,2i-1})^{\mathrm{T}}(\hat{V}_{k,2i-1})^{\mathrm{T}}.
$$

Likewise,

$$
(R_{k,2i})^{-1}(Q_{k,2i})^{\mathrm{T}} = (Q_{k,2i})^{\mathrm{T}} = (V_{k,2i})^{\mathrm{T}}(\hat{V}_{k,2i})^{\mathrm{T}}.
$$

Therefore, the fourth equality holds. The rest of the proof is similar to the proof of Lemma 3.2. □

With these two lemmas, we prepared preliminaries for the proof of Theorem 3.1. In the next lemma, we focus on the matrices $A_{k,i}$ in combination nodes.

Lemma 3.4. *For $k = 2, \ldots, q+1$ and $i = 1, \ldots, 2^{n-k+1}$, there exists a Stiefel matrix $Y_{k,i}$ such that $A_{k,i} = AY_{k,i}$.*

Proof. We construct the Stiefel matrix $Y_{k,i}$ following the data flow and track back upon the seed nodes.

$$
\begin{aligned}
A_{k,i} &= [A_{i,1}^k, A_{i,2}^k] \\
&= [U_{k-1,2i-1}S_{k-1,2i-1}(R_{k-1,2i-1})^{-1}, U_{k-1,2i}S_{k-1,2i}(R_{k-1,2i})^{-1}] \\
&= [U_{k-1,2i-1}S_{k-1,2i-1}(V_{k-1,2i-1})^{\mathrm{T}}V_{k-1,2i-1}, U_{k-1,2i}S_{k-1,2i}(V_{k-1,2i})^{\mathrm{T}}V_{k-1,2i}] \\
&= [U_{k-1,2i-1}S_{k-1,2i-1}(V_{k-1,2i-1})^{\mathrm{T}}(\hat{V}_{k-1,2i-1})^{\mathrm{T}}Q_{k-1,2i-1}, \\
&\quad U_{k-1,2i}S_{k-1,2i}(V_{k-1,2i})^{\mathrm{T}}(\hat{V}_{k-1,2i})^{\mathrm{T}}Q_{k-1,2i}]
\end{aligned}
$$

$$
\begin{aligned}
&= [U_{k-1,2i-1}S_{k-1,2i-1}(V_{k-1,2i-1})^{\mathrm{T}}, U_{k-1,2i}S_{k-1,2i}(V_{k-1,2i})^{\mathrm{T}}] \\
&\quad \begin{pmatrix} (\hat{V}_{k-1,2i-1})^{\mathrm{T}}Q_{k-1,2i-1} & 0 \\ 0 & (\hat{V}_{k-1,2i})^{\mathrm{T}}Q_{k-1,2i} \end{pmatrix} \\
&= (A_{k-1,2i-1}, A_{k-1,2i}) \begin{pmatrix} (\hat{V}_{k-1,2i-1})^{\mathrm{T}}Q_{k-1,2i-1} & 0 \\ 0 & (\hat{V}_{k-1,2i})^{\mathrm{T}}Q_{k-1,2i} \end{pmatrix}. \qquad (3.13)
\end{aligned}
$$

The first and second equalities are basic substitutions. In the third equality, we add the $V_{k-1,2i-1}$ and $V_{k-1,2i}$ because they are Stiefel matrices. Matrices $(R_{k-1,2i-1})^{-1}$ and $(R_{k-1,2i})^{-1}$ are dropped because they are identity matrices. The fourth equality use the thin QR factorisation step as in the proof of the last lemma.

For convenience, we define

$$
B^r_{s,t} = (A_{r,s}, \ldots, A_{r,t}) \qquad (3.14)
$$

and

$$
D^r_{s,t} = \mathrm{diag}((\hat{V}_{r,s})^{\mathrm{T}}Q_{r,s}, \ldots, (\hat{V}_{r,t})^{\mathrm{T}}Q_{r,t})) \qquad (3.15)
$$

Then, (3.13) can be writen as,

$$
A_{k,i} = B^{k-1}_{2i-1,2i}D^{k-1}_{2i-1,2i}. \qquad (3.16)
$$

The right hand side is only related to matrices on the $(k-1)$-th level. Now, we do the same calculations on $A_{k-1,2i-1}$ and $A_{k-1,2i}$, and repeat this process upwards to the seed nodes.

$$
\begin{aligned}
A_{k,i} &= (B^{k-2}_{4i-3,4i-2}D^{k-2}_{4i-3,4i-2}, B^{k-2}_{4i-1,4i}D^{k-2}_{4i-1,4i})D^{k-1}_{2i-1,2i} \\
&= B^{k-2}_{4i-3,4i}D^{k-2}_{4i-3,4i}D^{k-1}_{2i-1,2i} \\
&= \cdots \\
&= B^{2}_{(i-1)\times 2^{k-2}+1,i\times 2^{k-2}}D^{2}_{(i-1)\times 2^{k-2}+1,i\times 2^{k-2}} \cdots D^{k-2}_{4i-3,4i}D^{k-1}_{2i-1,2i} \\
&= (A_{1,(i-1)2^{k-1}+1}, A_{1,(i-1)2^{k-1}+2}, \ldots, A_{1,(i-1)2^{k-1}+2^{k-1}}) \\
&\quad \mathrm{diag}(V_{1,(i-1)2^{k-1}+1}, V_{1,(i-1)2^{k-1}+2}, \ldots, V_{1,(i-1)2^{k-1}+2^{k-1}}) \\
&\quad D^{2}_{(i-1)\times 2^{k-2}+1,i\times 2^{k-2}} \cdots D^{k-2}_{4i-3,4i}D^{k-1}_{2i-1,2i} \\
&= A(I^{[(i-1)2^{k-1}+1]}, I^{[(i-1)2^{k-1}+1]}, \ldots, I^{[(i-1)2^{k-1}+2^{k-1}]}) \\
&\quad \mathrm{diag}(V_{1,(i-1)2^{k-1}+1}, V_{1,(i-1)2^{k-1}+2}, \ldots, V_{1,(i-1)2^{k-1}+2^{k-1}}) \\
&\quad D^{2}_{(i-1)\times 2^{k-2}+1,i\times 2^{k-2}} \cdots D^{k-2}_{4i-3,4i}D^{k-1}_{2i-1,2i} \\
&=: AY_{k,i},
\end{aligned}
$$

where $I^{[l]}$ is defined as follows: let $m^{[i]}$ be the number of columns of $A^{[i]}$, $i = 1, \ldots, 2^n$, $n_1^{[i]} = \Sigma_{i=1}^{l-1} m^{[i]}$, $n_2^{[i]} = m^{[l]}$ and $n_3^{[i]} = \Sigma_{i=l+1}^{2^q} m^{[i]}$.

$$I^{[l]} = (0_{m^{[l]} \times n_1^{[i]}}, I_{m^{[l]} \times n_2^{[i]}}, 0_{m^{[l]} \times n_2^{[i]}})^{\mathrm{T}}.$$

Matrix $Y_{k,i}$ has mutually orthonormal columns since it is a multiplication of a sequence of Stiefel matrices.

□

For convenience, we use

$$\bigoplus_{i=1}^{l} A^i = [A^1, A^2, \ldots, A^l]$$

to denote the concatenation of matrices $A^1, A^2, \ldots, A^l$.

In the following lemma, we derive a bound for $\|z^{\mathrm{T}} A\|$, $z \in \tilde{\mathcal{H}}^{\perp}$, in terms of $\sigma_{p+1}(A_{k,i})$.

Lemma 3.5. *Suppose $z \in \tilde{\mathcal{H}}^{\perp}$ with $\|z\| = 1$. Then,*

$$\|z^{\mathrm{T}} A\|^2 \leq \Sigma_{l=1}^{2^q} \sigma_{p+1}^2(A_{1,l}) + \Sigma_{k=2}^{q+1} \Sigma_{i=1}^{2^{q-k+1}} \sigma_{p+1}^2(A_{k,i}).$$

Proof.

$$\begin{aligned} \|z^{\mathrm{T}} A\|^2 &= z^{\mathrm{T}} A A^{\mathrm{T}} z \\ &= z^{\mathrm{T}} A_{q+1,1} (A_{q+1,1})^{\mathrm{T}} z + (z^{\mathrm{T}} A A^{\mathrm{T}} z - z^{\mathrm{T}} A_{q+1,1} (A_{q+1,1})^{\mathrm{T}} z). \end{aligned} \quad (3.17)$$

For $z \in \tilde{\mathcal{H}}^{\perp}$, the first term $z^{\mathrm{T}} A_{q+1,1} (A_{q+1,1})^{\mathrm{T}} z$ takes its maximal value $\sigma_{p+1}^2(A_{q+1,1})$ when z equals the $(p+1)$-st left singular vector of $A_{q+1,1}$.

The second term is bounded by

$$\|A A^{\mathrm{T}} - A_{q+1,1} (A_{q+1,1})^{\mathrm{T}}\|_2. \quad (3.18)$$

Next, let us derive a bound for (3.18).

$$\begin{aligned} & \|A A^{\mathrm{T}} - A_{q+1,1} (A_{q+1,1})^{\mathrm{T}}\|_2 \\ \leq & \left\| \left(\bigoplus_{i=1}^{2^q} A_{1,i} \right)^{\mathrm{T}} - \left(\bigoplus_{i=1}^{2^{q-1}} A_{2,i} \right) \left(\bigoplus_{i=1}^{2^{q-1}} A_{2,i} \right)^{\mathrm{T}} \right\| \\ & + \left\| \left(\bigoplus_{i=1}^{2^{q-1}} A_{2,i} \right) \left(\bigoplus_{i=1}^{2^{q-1}} A_{2,i} \right)^{\mathrm{T}} - \left(\bigoplus_{i=1}^{2^{q-2}} A_{3,i} \right) \left(\bigoplus_{i=1}^{2^{q-2}} A_{3,i} \right)^{T} \right\| + \ldots \\ & + \|[A_{q,1}, A_{q,2}][A_{q,1}, A_{q,2}]^{\mathrm{T}} - A_{q+1,1} (A_{q+1,2})^{\mathrm{T}}\| \end{aligned}$$

$$
\begin{aligned}
\leq\ & \{\|[A_{1,1}, A_{1,2}][A_{1,1}, A_{1,2}]^{\mathrm{T}} - A_{2,1}(A_{2,1})^{\mathrm{T}}\| + \dots \\
& + \|[A_{1,2^q-1}, A_{1,2^q}][A_{1,2^q-1}, A_{1,2^q}]^{\mathrm{T}} - A_{1,2^{q-1}}()A_{1,2^{q-1}})^{\mathrm{T}}\|\} \\
& + \{\|[A_{2,1}, A_{2,2}][A_{2,1}, A_{2,2}]^{\mathrm{T}} - A_{3,1}(A_{3,1})^{\mathrm{T}}\| + \dots \\
& + \|[A_{2,2^{q-1}-1}, A_{2,2^{q-1}}][A_{2,2^{q-1}-1}, A_{2,2^{q-1}}]^{\mathrm{T}} - A_{3,2^{q-2}}(A_{3,2^{q-2}})^{\mathrm{T}}\|\} + \dots \\
& + \|[A_{q,1}, A_{q,2}][A_{q,1}, A_{q,2}]^{\mathrm{T}} - A_{q+1,1}(A_{q+1,1})^{\mathrm{T}}\| \\
\leq\ & \sigma_{p+1}^2(A_{1,1}) + \sigma_{p+1}^2(A_{1,2}) + \dots + \sigma_{p+1}^2(A_{1,2^q-1}) + \sigma_{p+1}^2(A_{1,2^q}) \\
& \sigma_{p+1}^2(A_{2,1}) + \sigma_{p+1}^2(A_{2,2}) + \dots + \sigma_{p+1}^2(A_{2,2^{q-1}-1}) + \sigma_{p+1}^2(A_{2,2^{q-1}}) + ... \\
& ... + \sigma_{p+1}^2(A_{q,1}) + \sigma_{p+1}^2(A_{q,2}) \\
=\ & \Sigma_{l=1}^{2^q}\sigma_{p+1}^2(A_{1,l}) + \Sigma_{k=2}^{q}\Sigma_{i=1}^{2^{q-k+1}}\sigma_{p+1}^2(A_{k,i}).
\end{aligned}
$$

The first and second inequalities follow from basic substitutions, rearrangement of the submatrices and the triangle inequality. The third inequality follows from Lemmas 3.2 and 3.3.

We finish this proof by adding the first term in (3.17) back in. □

In the following two lemmas, we show that each term in the bound of Lemma 3.5 is smaller than $\sigma_{p+1}^2(A)$ so that we can achieve a bound only consisting of terms related to the original matrix A.

Lemma 3.6. *$\sigma_{p+1}^2(A_{1,l}) \leq \sigma_{p+1}^2(A)$, for all $l = 1, \dots, 2^q$.*

Proof. With $I^{[l]}$ defined as in the proof of Lemma 3.4, we have $A_{1,l} = AI^{[l]}$.

By Lemma 3.3.1 of [55], we have

$$\lambda_{p+1}((A_{1,l})^{\mathrm{T}}A_{1,l}) = \lambda_{p+1}((I^{[l]})^{\mathrm{T}}A^{\mathrm{T}}AI^{[l]}) \leq \lambda_{p+1}(A^{\mathrm{T}}A),$$

which implies

$$\sigma_{p+1}^2(A_{1,l}) \leq \sigma_{p+1}^2(A).$$

□

Lemma 3.7. *$\sigma_{p+1}^2(A_{k,i}) \leq \sigma_{p+1}^2(A)$, for all $k = 2, \dots, q+1$ and $i = 1, \dots, 2^{n-k+1}$.*

Proof. As proved in Lemma 3.4, there exists a Stiefel matrix $Y_{k,i}$ such that $A_{k,i} = AY_{k,i}$. By Lemma 3.3.1 of [55], we have

$$\lambda_{p+1}((A_{k,i})^{\mathrm{T}}A_{k,i}) = \lambda_{p+1}((Y_{k,i})^{\mathrm{T}}A^{\mathrm{T}}AY_{k,i}) \leq \lambda_{p+1}(A^{\mathrm{T}}A),$$

which implies

$$\sigma_{p+1}^2(A_{k,i}) \leq \sigma_{p+1}^2(A).$$

□

Finally, we obtain our theorem by summing all the bounds derived so far.

Proof.

$$\begin{aligned}
\|A - \breve{U}\breve{U}^{\mathrm{T}}A\|_2 &\leq \max_{z\in\tilde{\mathcal{H}}^{\perp},\|z\|=1} |z^{\mathrm{T}}A| \\
&\leq (\Sigma_{l=1}^{2^q}\sigma_{p+1}^2 A_{1,l} + \Sigma_{k=2}^{q+1}\Sigma_{i=1}^{2^{q-k}}\sigma_{p+1}^2(A_{k,i}))^{1/2} \\
&\leq \sqrt{2^{q+1}-1}\sigma_{p+1}(A) \\
&= \sqrt{2s-1}\sigma_{p+1}(A).
\end{aligned}$$

The first inequality follows from Lemma 3.1. The second inequality follows from Lemma 3.5. The last inequality follows from Lemmas 3.6 and 3.7. □

3.4 Tightness of the error bound

In the previous section, we proved that $\|A-\breve{U}\breve{U}^{\mathrm{T}}A\|_2$ is bounded by $\sqrt{2s-1}\sigma_{p+1}(A)$. In terms of relative error, we find

$$\frac{\|A - \breve{U}\breve{U}^{\mathrm{T}}A\|_2}{\|A\|_2} \leq \sqrt{2s-1}\frac{\sigma_{p+1}(A)}{\sigma_1(A)}. \tag{3.19}$$

In practice, thousands or more nodes may be used. As the error bound is proportional to the square root of the number of seed nodes, users may worry about the accuracy when a large number of nodes are used. A crucial question is therefore whether this error bound is tight. In this section, we answer this question empirically with examples and numerical tests.

We define γ as follows.

$$\gamma = \frac{\left(\frac{\|A-\breve{U}\breve{U}^{\mathrm{T}}A\|_2}{\sigma_{p+1}}\right)^2 + 1}{2}.$$

In the theory of Theorem 3.1, γ has an upper bound s.

Two simple extreme examples provide the first insights of the answer to the question. In the first example, the matrix A is made up of s identical matrices B and each of the seed nodes receives a copy of B. This is the most 'balanced' case in the sense of information distribution.

Example 3.1. *Let $B \in \mathbb{R}^{m\times n}$, and let $A = [B, B, \ldots, B] \in \mathbb{R}^{m\times sn}$ be the matrix made up of s replica blocks of B. Assume that $m \leq n$, let $B = U_B S_B V_B^{\mathrm{T}}$ be the thin*

SVD of B, and $S_A = \sqrt{s}S_B$. Then

$$\begin{aligned} S_A^{-1}U_B^{\mathrm{T}}A &= \frac{1}{\sqrt{s}}S_B^{-1}U_B^{\mathrm{T}}[B, B, \ldots, B] \\ &= \frac{1}{\sqrt{s}}S_B^{-1}U_B^{\mathrm{T}}[U_BS_BV_B^{\mathrm{T}}, U_BS_BV_B^{\mathrm{T}}, \ldots, U_BS_BV_B^{\mathrm{T}}] \\ &= \frac{1}{\sqrt{s}}S_B^{-1}[S_BV_B^T, S_BV_B^{\mathrm{T}}, \ldots, S_BV_B^{\mathrm{T}}] \\ &= \frac{1}{\sqrt{s}}[V_B^{\mathrm{T}}, V_B^{\mathrm{T}}, \ldots, V_B^{\mathrm{T}}]. \end{aligned}$$

Let $V_A = \frac{1}{\sqrt{s}}[V_B^{\mathrm{T}}, V_B^{\mathrm{T}}, \ldots, V_B^{\mathrm{T}}]^{\mathrm{T}}$, then

$$\begin{aligned} V_A^{\mathrm{T}}V_A &= \frac{1}{\sqrt{s}}[V_B^{\mathrm{T}}, V_B^{\mathrm{T}}, \ldots, V_B^{\mathrm{T}}] \times \frac{1}{\sqrt{s}}[V_B^{\mathrm{T}}, V_B^{\mathrm{T}}, \ldots, V_B^{\mathrm{T}}]^{\mathrm{T}} \\ &= \frac{1}{s}(V_B^{\mathrm{T}}V_B + V_B^{\mathrm{T}}V_B + \cdots + V_B^{\mathrm{T}}V_B) \\ &= I. \end{aligned}$$

Therefore, V_A is Stiefel, and $A = U_BS_AV_A^{\mathrm{T}}$ is a thin SVD of A.

Matrices A and B have the same left singular vectors. If the SVD of A is computed by the algorithm with s seed nodes and one copy of submatrix B is distributed to each of the seed nodes, the exact p leading left singular space is computed under the exact arithmetic assumption, i.e., $\|A - \breve{U}\breve{U}^{\mathrm{T}}A\|_2 = \sigma_{p+1}(A)$. Thus, we find that $\gamma = 1$ in this example, which is much smaller than the upper bound s.

In the case of Example 3.1, the estimate of Theorem 3.1 turns out to be very conservative. In the bound estimation, we were overly pessimistic about the quality of the outputs of local calculations in every node.

In the second example, we look at an 'unbalanced' case, where all of the non-zero entries concentrate in the leftmost block.

Example 3.2. *Let $B \in \mathbb{R}^{m\times n}$, and let $A = [B, 0, \ldots, 0] \in \mathbb{R}^{m\times sn}$ be a matrix made up by B and $(s-1)$ blocks of zeros of size $m \times n$. Assume that $m \leq n$, let $B = U_BS_BV_B^{\mathrm{T}}$ be the thin SVD of B, and*

$$V_A = \begin{bmatrix} V_B \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

the $m \times sn$ matrix whose first m rows are the same as V_B while all other entries are zeros. We have

$$\begin{aligned} U_B^{\mathrm{T}} A V_A &= U_B^{\mathrm{T}}[B, 0, \ldots, 0] \begin{bmatrix} V_B \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\ &= U_B^{\mathrm{T}}[U_B S_B V_B^{\mathrm{T}}, 0, \ldots, 0] \begin{bmatrix} V_B \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\ &= S_B. \end{aligned}$$

Therefore, $A = U_B S_B V_A^{\mathrm{T}}$ is the SVD of A.

If we use Algorithm 10 to compute the SVD of A, the exact leading SVD can be found with exact arithmetic in only the first seed node. Thus, $\|A - \breve{U}\breve{U}^{\mathrm{T}} A\|_2 = \sigma_{p+1}(A)$ and $\gamma = 1$, once again. Only the first seed node executes efficient computations. All of the other nodes are redundant, whereas in the analysis, we assumed that each node introduces the same numerical error. This makes the upper bound very conservative and leads to the appearance of an additional coefficient $\sqrt{2s-1}$.

In the first example, the information contained in the matrices is uniformly distributed in each input matrix in seed nodes. In the second example, the non-zero entries concentrate in the input of first seed node. In general, a matrix lies in between these two extreme examples. In the following numerical tests, we look at more general matrices.

We compute the SVDs of randomly generated matrices of size 1000 by 16284. A random matrix $A \in \mathbb{R}^{m \times n}$ is constructed as,

$$A = U_k S_k V_k^{\mathrm{T}}, \tag{3.20}$$

where k is a number greater than p, the number of leading singular vectors and singular values that we want to compute. The Stiefel matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{m \times k}$ are taken as the orthogonalisation of $m \times n$ matrices with i.i.d. standard Gaussian entries. The diagonal entries of $S \in \mathbb{R}^{k \times k}$ are taken as $100, 100/\alpha, \ldots, 100/\alpha^{k-1}$, where $\alpha > 1$ is the decrease factor of the singular values. We perform tests with matrices generated with different decrease factors and compute the 5-leading SVDs with binary execution trees of different number of levels. Table 3.1 lists the average γ in 100 tests.

α \ s	4	8	16
10	1+3.80e-12	1+8.46e-12	1+1.65e-11
4	1+1.75e-10	1+3.77e-10	1+7.78.e-10
1.01	1+1.64e-2	1+1.85e-2	1+2.69e-2
α \ s	32	64	128
10	1+3.79.e-11	1+8.86e-11	1+1.45e-10
4	1+2.05e-9	1+3.86e-9	1+7.38e-9
1.01	1+3.00e-2	1+2.68e-2	1+2.75e-2

Table 3.1: Comparision of γ with different number of seed nodes and decrease factor α

Theorem 3.1 gives s as an upper bound of γ. In the numerical tests, however, γ is just slightly greater than its optimal value 1. In practise, the performance of the algorithm is therefore always far better than the worst case scenarios considered in this analysis.

In the next chapter, we establish another approach of analysis via the leading right singular space. The framework of the second analysis is based on geometric insights of the methodology. Instead of considering the worst case scenarios, we will have to make the assumption that certain subspaces are in general positions.

Chapter 4

Convergence Analysis Via Right Singular Spaces

In the previous chapter, we analysed our algorithm by investigating the approximation quality of the leading left singular space. A different approach of analysis will be presented in this chapter. In this analysis, the leading right singular space plays an essential role. We will start from Motivation 2.1 in Chapter 2 and use this geometric intuition to construct the framework of our analysis. The analysis consists of two parts, the global error accumulation and the local error analysis. In the global analysis, we show that the growth rate of errors is linear in the number of seed nodes. For the local errors, we study the case where the input spaces W are in random position. Although exceptions exist that larger local errors may occur, most matrices behave similarly to random matrices.

This chapter is organised as follows: the details of the algorithm, as well as notations for the analysis, will be given in Section 4.1. The global error accumulation and the local error analysis are given in Sections 4.2 and 4.3, respectively. Numerical tests of the local error bound follow in Section 4.4.

4.1 A geometric observation

Given a large and dense matrix $A \in \mathbb{R}^{m\times n}$, we want to compute its p-leading SVD

$$A \approx U_p S_p V_p^{\mathrm{T}}. \tag{4.1}$$

The problem of find the space generated by the column vectors of V_p can be cast as an optimisation problem,

$$\begin{aligned} \mathbf{P} \quad & \min_{Q\in\mathbb{R}^{n\times p}} && \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ & \text{s.t.} && Q^{\mathrm{T}}Q = I_p. \end{aligned} \tag{4.2}$$

We remark that

$$\mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) = \|V_p V_p^{\mathrm{T}} - QQ^{\mathrm{T}}\|_2.$$

This problem is ill-posed, becasue V_p in the objective function is unknown to us, and hence, the objective function cannot be evaluated. Searching the nearest space to range(Q) is also costly due to the high dimension of A.

In order to solve this problem, we consider a sequence of subproblems that are easier to solve. For any $n \times k$ ($k \geq p$) Stiefel matrix W, we find the nearest subspace to range(V_p) in the subspace generated by the column vectors of W. This problem is formulated as,

$$\begin{aligned} \mathbf{P}(\mathbf{W}) \quad \min_{Q \in \mathbb{R}^{n \times p}} \quad & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}} Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(W). \end{aligned} \tag{4.3}$$

The column vectors of the optimal solution to this problem Q^* form a basis of the nearest subspace.

A subproblem $\mathbf{P}(\mathbf{W})$ can be solved by local computations in a node. We use the framework in Algorithm 10. The local computations in each node are slightly different from Algorithm 7, 8 and 9. The details of the local computations of the three kinds of nodes are described in Algorithm 11, 12 and 13. We remark that the inputs $W^{[i]}$ of seed nodes are orthgonal to each other.

Algorithm 11 Seed nodes $(1, i)$

0) Input: $AW^{[i]}$, where $W^{[i]} \in \mathbb{R}^{m \times (n/2^q)}$ is a Stifel matrix.
1) Compute the p-leading SVD: $AW^{[i]} \approx U^{[i]} S^{[i]} (V^{[i]})^{\mathrm{T}}$
2)Compute $\breve{V}^{[i]} = W^{[i]} V^{[i]}$
Output: $A^2_{l,j} = U^{[i]} S^{[i]}$, $V^2_{l,j} = \breve{V}^{[i]} V$, where $l = (i+1)/2$, $j = 1$ if $i \equiv 1 (\mathrm{mod}\ 2)$; $l = i/2$ $j = 2$, otherwise.

Algorithm 12 Combination nodes (k, i)

0) Input: $A^k_{i,1}$, $A^k_{i,2}$, $V^k_{i,1}$ and $V^k_{i,2}$
1) Let $A_{k,i} = (A^k_{i,1}, A^k_{i,2})$, $\breve{V}_{k,i} = (V^k_{i,1}, V^k_{i,2})$
2) Compute the p-leading part SVD: $A_{k,i} \approx U_{k,i} S_{k,i} (V_{k,i})^{\mathrm{T}}$
3) Compute thin QR factorization: $\breve{V}_{k,i} V_{k,i} = Q_{k,i} R_{k,i}$
4) Output: $A^{k+1}_{l,j} = U_{k,i} S_{k,i} (R_{k,i})^{-1}$, $V^{k+1}_{l,j} = Q_{k,i}$, where $l = (i+1)/2$, $j = 1$ if $i \equiv 1 (\mathrm{mod}\ 2)$; $l = i/2$ $j = 2$, otherwise.

In this analysis, we use the second approach of combination which is mentioned in Section 2.1.6, while in the analysis of Chapter 3, we use the first approach of

Algorithm 13 Extraction nodes $(q+1,1)$

0) Input: $A^1_{q+1,1}$, $A^2_{1+1,2}$, $V^1_{q+1,1}$ and $V^2_{q+1,2}$
1) Let $A_{q+1,1} = (A^1_{q+1,1}, A^2_{q+1,1})$, $\breve{V}_{q+1,1} = (V^1_{q+1,1}, V^2_{q+1,1})$
2) Compute the p-leading part SVD: $A_{q+1,1} \approx U_{q+1,1} S_{q+1,1} (V_{q+1,1})^{\mathrm{T}}$
3) Compute thin QR factorization: $\breve{V}_{q+1,1} V_{q+1,1} = Q_{q+1,1} R_{q+1,1}$
4) $\breve{A} = U_{q+1,1} S_{q+1,1} (R_{q+1,1})^{-1}$.
5) Compute the p-leading part SVD: $\breve{A} = \breve{U}\breve{S}\bar{V}^{\mathrm{T}}$.
6) $\breve{V} = Q_{q+1,1}\bar{V}$;
7) Output: $[\breve{U}, \breve{S}, \breve{V}]$

combination. Another difference from the previous analysis is that the input matrix includes all orthogonal matrices instead of considering only column-splitting in the first analysis.

In the following sections, we will present the analysis in two parts, a global error accumulation analysis and a local error analysis. The global analysis gives the framework of this analysis. We will investigate the error accumulation throughout the execution tree. In the local error analysis, we complete the proof by studying the approximation errors in local computations.

4.2 Global analysis

Firstly, we will construct the framework of this analysis. We will show how the estimation of the leading right singular space is refined from the root of the execution tree and how the local errors accumulate. Here, we introduce the definition of *local error*.

Definition 4.1. local error *is the difference between the approximate solution computed by the local computations of a node and the exact solutions to the optimisation problem.*

Four Stiefel matrices V_p, $\bar{V}$, $\hat{V}$ and $\tilde{V}$ are important in this analysis. They are the matrices of p-leading right singular vectors of four different matrices respectively:

- Matrix V_p is the matrix whose column vectors are the leading right singular vectors of A. The range space of V_p is what we aim to find.
- Let W be an $n \times k$ Stiefel matrix with $k \geq p$, and let

$$A_p W \approx \bar{U}\bar{S}\bar{V}^{\mathrm{T}} \tag{4.4}$$

be the p leading SVD of A_pW. Matrix $\bar{V}$ is a $k \times p$ matrix whose column vectors are the leading right singular vectors of A_pW.

- Let
$$V_p^{\mathrm{T}} W = \hat{U}\hat{S}\hat{V}^{\mathrm{T}} \tag{4.5}$$
be the SVD of $V_p^{\mathrm{T}}W$. Matrix $\hat{V}$ is the $k \times p$ matrix whose column vectors are the leading right singular vectors of $V_p^{\mathrm{T}}W$.

- Similarly, $\tilde{V}$ is the $k \times p$ matrix whose column vectors are the leading right singular vectors of AW. That is,
$$AW \approx \tilde{U}\tilde{S}\tilde{V}^{\mathrm{T}} \tag{4.6}$$
is the p-leading SVD of AW.

Matrices $\bar{V}$, $\hat{V}$ and $\tilde{V}$ are closely related to the optimal solution of problem $\mathbf{P}(\mathbf{W})$. We will use them to construct the framework of this analysis.

Before taking the local errors into consideration, we look at an idealised case in which the local computation at every node gives us the exact solution of Problem (4.3). This idealised case inspires the analysis.

4.2.1 An idealised case

We consider an idealised case, which is unlikely to happen but provides geometric insights for the general analysis. The idealised assumption we make here is stated in Assumption 4.1.

Assumption 4.1. *We assume that at each node, if the input is W and AW, we have*
$$\mathrm{range}(\tilde{V}) = \mathrm{range}(\hat{V}). \tag{4.7}$$

At seed nodes $(1, i)$, $i = 1, \ldots, 2^q$, there is an associated optimisation problem that is defined by the input Stiefel matrix $W^{[i]}$,
$$\begin{aligned} \mathbf{P}^{[\mathbf{1},\mathbf{i}]} \quad \min_{Q} \quad & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(W^{[i]}). \end{aligned} \tag{4.8}$$
The second constraint is added to confine the search space to $\mathrm{range}(W^{[i]})$. Geometrically, we seek to find the nearest subspace to $\mathrm{range}(V_p)$ in $\mathrm{range}(W^{[i]})$. It is the same as finding an orthogonal basis of the projection of V_p onto $\mathrm{range}(W^{[i]})$.

The goal of the local computations is to find an approximate solution to this problem. Let us start from the following lemma.

Lemma 4.1. *$Q^* = W\hat{V}$ is an optimal solution to problem 4.3*

Proof. For every $n \times p$ Stiefel matrix Q such that $\text{range}(Q) \subset \text{range}(W)$, there exists a $k \times p$ Stiefel matrix P such that $Q = WP$.

By the definition of distance between subspaces,

$$\text{dist}(\text{range}(V_p), \text{range}(Q)) = \|V_p V_p^{\mathrm{T}} - QQ^{\mathrm{T}}\|_2 = \|V_p V_p^{\mathrm{T}} - WPP^T W^{\mathrm{T}}\|_2.$$

Let

$$V_p^{\mathrm{T}} Q = V_p^{\mathrm{T}} WP = U_{V_p^{\mathrm{T}}Q} S_{V_p^{\mathrm{T}}Q} (V_{V_p^{\mathrm{T}}Q})^{\mathrm{T}}$$

be the full SVD of $V_p^{\mathrm{T}} Q$. Writing $\theta_1, \theta_2, \ldots, \theta_p \in [0, \pi/2]$ for the principal angles between $\text{range}(V_p)$ and $\text{range}(Q)$, we have

$$S_{V_p^{\mathrm{T}}Q} = \begin{bmatrix} \cos(\theta_1) & & & \\ & \cos(\theta_2) & & \\ & & \ddots & \\ & & & \cos(\theta_p) \end{bmatrix},$$

and

$$\text{dist}(\text{range}(V_p), \text{range}(Q)) = \sin\theta_p,$$

see [12].

Similarly, the thin SVD of $V_p^{\mathrm{T}} W$ is (4.5).

If $\vartheta_1, \vartheta_2, \ldots, \vartheta_p \in [0, \pi/2]$ are the principal angles between $\text{range}(V_p)$ and $\text{range}(W)$, then

$$\hat{S} = \begin{bmatrix} \cos(\vartheta_1) & & & \\ & \cos(\vartheta_2) & & \\ & & \ddots & \\ & & & \cos(\vartheta_p) \end{bmatrix}$$

and

$$\text{dist}(\text{range}(V_p), \text{range}(W)) = \sin\vartheta_p = \sqrt{1 - \sigma_i^2(V_p^{\mathrm{T}} W)}.$$

By Cauchy's interlacing theorem [45], for $i = 1, \ldots, p$, we have

$$\lambda_i(P^{\mathrm{T}} W^{\mathrm{T}} V_p V_p^{\mathrm{T}} WP) \leq \lambda_i(W^{\mathrm{T}} V_p V_p^{\mathrm{T}} W).$$

This implies that,

$$\cos\theta_i = \sigma_i(V_p^{\mathrm{T}} WP) \leq \sigma_i(V_p^{\mathrm{T}} W) = \cos\vartheta_i.$$

Therefore,

$$\text{dist}(\text{range}(V_p), \text{range}(Q)) = \sin\theta_p \geq \sin\vartheta_p = \sqrt{1 - \sigma_i^2(V_p^{\mathrm{T}} W)}. \tag{4.9}$$

Thus, $\sin\vartheta_p$ is a lower bound for $\mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q))$.

Notice that $Q^* = W\hat{V}$, where $P = \hat{V}$. It follows that

$$V_p^{\mathrm{T}} Q^* = V_p^{\mathrm{T}} W \hat{V} = \hat{U}\hat{S}\hat{V}^{\mathrm{T}}\hat{V} = \hat{U}\hat{S}. \tag{4.10}$$

Let θ_p^* be the p-th principal angle between $\mathrm{range}(V_p)$ and $\mathrm{range}(Q^*)$. In view of (4.10),

$$\mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q^*)) = \sin\theta_p^* = \sqrt{1-\hat{\sigma}_p^2}.$$

The lower bound of $\sin\theta_p$ in (4.9) achieved. Therefore, $Q^* = W\hat{V}$ is an optimal solution to $\mathbf{P}(\mathbf{W})$.

□

By Lemma 4.1, $Q^{[i]} = W^{[i]}\hat{V}^{[i]}$ is an optimal solution to $\mathbf{P}^{[\mathbf{1},\mathbf{i}]}$. Under Assumption 4.1, there exists a $p \times p$ orthogonal matrix P such that $\tilde{V}^{[i]} = V^{[i]}P$. We have

$$\breve{V}^{[i]} = W^{[i]}V^{[i]} = W^{[i]}\hat{V}^{[i]}P = Q^{[i]}P,$$

and then,

$$\mathrm{range}(\breve{V}^{[i]}) = \mathrm{range}(Q^{[i]}).$$

Therefore, $\breve{V}^{[i]}$ is another optimal solution to $\mathbf{P}^{[\mathbf{1},\mathbf{i}]}$.

In conclusion, under the idealised Assumption 4.1, the local computations in seed node $(1,i)$, $(i = 1,\ldots,2^q)$ find an optimal solution $\breve{V}^{[i]}$ to problem $\mathbf{P}^{[\mathbf{1},\mathbf{i}]}$.

Next, we look at the combination nodes. At the second level, the associated optimisation problem to combination nodes $(2,i)$, $(i = 1,\ldots,2^{q-1})$, is,

$$\begin{aligned} \mathbf{P}^{[\mathbf{2},\mathbf{i}]} \quad \min \quad & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(\breve{V}_i^2). \end{aligned} \tag{4.11}$$

The local computations seek to find the nearest subspace to $\mathrm{range}(V_p)$ in $\mathrm{range}(\breve{V}_i^2)$.

Matrix $\breve{V}_{2,i}$ is constructed from the outputs of the parent nodes of node $(2,i)$,

$$\breve{V}_{2,i} = (V_{i,1}^2, V_{i,2}^2).$$

Matrix $V_{i,1}^2$ is computed in the second step of the local computations in seed node $(1, 2i-1)$. It follows that

$$V_{i,1}^2 = \breve{V}^{[2i-1]} = W^{[2i-1]}V^{[2i-1]},$$

and

$$V_{i,2}^2 = \breve{V}^{[2i]} = W^{[2i]}V^{[2i]}.$$

When seeding the seed nodes, we chose $W^{[1]}, \ldots, W^{[2^q]}$ mutually orthogonal. Matrices $\breve{V}^{[2i-1]}$ and $\breve{V}^{[2i]}$ are orthogonal to each other as well because they are subspaces of $W^{[2i-1]}$ and $W^{[2i]}$, respectively. Therefore, $\mathbf{P}^{[\mathbf{2},\mathbf{i}]}$ is equivalent to $\hat{\mathbf{P}}^{[\mathbf{2},\mathbf{i}]}$,

$$\begin{aligned} \hat{\mathbf{P}}^{[\mathbf{2},\mathbf{i}]} \quad \min \quad & \operatorname{dist}(\operatorname{range}(V_p), \operatorname{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \operatorname{range}(Q) \subset \operatorname{range}((\breve{V}^{[2i-1]}, \breve{V}^{[2i]})). \end{aligned} \tag{4.12}$$

By Lemma 4.1, $(\breve{V}^{[2i-1]}, \breve{V}^{[2i]})\hat{V}_{2,i}$ is an optimal solution to $\hat{\mathbf{P}}^{[\mathbf{2},\mathbf{i}]}$, where $\hat{V}_{2,i}$ comes from the thin-SVD of $V_p^T(\breve{V}^{[2i-1]}, \breve{V}^{[2i]})$,

$$V_p^T(\breve{V}^{[2i-1]}, \breve{V}^{[2i]}) = \hat{U}_{2,i}\hat{S}_{2,i}(\hat{V}_{2,i})^{\mathrm{T}}.$$

On the other hand,

$$\begin{aligned} A_{i,1}^2 &= U^{[2i-1]}S^{[2i-1]} \\ &= U^{[2i-1]}S^{[2i-1]}(V^{[2i-1]})^{\mathrm{T}}V^{[2i-1]} \\ &= AW^{[2i-1]}V^{[2i-1]} \\ &= A\breve{V}^{[2i-1]}. \end{aligned}$$

Likewise,

$$A_{i,2}^2 = A\breve{V}^{[2i]}.$$

The input matrix of node $(2, i)$ equals

$$A_{2,i} = (A_{i,1}^2, A_{i,2}^2) = (A\breve{V}^{[2i-1]}, A\breve{V}^{[2i]}) = A(\breve{V}^{[2i-1]}, \breve{V}^{[2i]}). \tag{4.13}$$

In the local computations of node $(2, i)$, we compute the p-leading SVD of $A_{2,i}$,

$$A_{2,i} \approx U_{2,i}S_{2,i}(V_{2,i})^{\mathrm{T}}.$$

Under Assumption 4.1, we have

$$\operatorname{range}(V_{2,i}) = \operatorname{range}(\hat{V}_{2,i}).$$

Therefore, $\breve{V}_{2,i}V_{2,i} = (\breve{V}^{[2i-1]}, \breve{V}^{[2i]})V_{2,i}$ is another optimal solution to $\hat{\mathbf{P}}^{[\mathbf{2},\mathbf{i}]}$.

In step 3 of the local computation of node $(2, i)$, we compute the QR factorization of $\breve{V}_{2,i}V_{2,i}$,

$$Q_{2,i}R_{2,i} = \breve{V}_{2,i}V_{2,i}.$$

Therefore, range($Q_{2,i}$) is the same as range($\breve{V}_{2,i}V_{2,i}$). Thus, $V^2_{l,j} = Q_{2,i}$ is an optimal solution to $\hat{\mathbf{P}}^{[\mathbf{2,i}]}$, too.

As discussed, $\breve{V}^{[2i-1]}$ and $\breve{V}^{[2i]}$ are optimal solutions to problems $\mathbf{P}^{[\mathbf{1,2i-1}]}$ and $\mathbf{P}^{[\mathbf{1,2i}]}$, respectively. $\hat{\mathbf{P}}^{[\mathbf{2,i}]}$ is equivalent to

$$\begin{array}{lll} \tilde{\mathbf{P}}^{[\mathbf{2,i}]} & \min\limits_{Q} & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ & \text{s.t.} & Q^{\mathrm{T}}Q = I_p, \\ & & \mathrm{range}(Q) \subset \mathrm{range}((W^{[2i-1]}, W^{[2i]})). \end{array} \tag{4.14}$$

We conclude that $V^2_{l,j}$ is an optimal solution to $\tilde{\mathbf{P}}^{[\mathbf{2,i}]}$. Through combination node $(2, i)$, the search space for the nearest subspace to range(V_p) is augmented to range($W^{[2i-1]}, W^{[2i]}$).

Next, we carry on with the third level. For node $(3, i)$, $i = 1, \ldots, 2^{q-2}$, the associated problem is

$$\begin{array}{lll} \mathbf{P}^{[\mathbf{3,i}]} & \min & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ & \text{s.t.} & Q^{\mathrm{T}}Q = I_p, \\ & & \mathrm{range}(Q) \subset \mathrm{range}(\breve{V}_{3,i}). \end{array} \tag{4.15}$$

The local computations seek to find the nearest subspace to range(V_p) in range($\breve{V}_{3,i}$).

$$\breve{V}_{3,i} = (V^3_{i,1}, V^3_{i,2}) = (Q_{2,2i-1}, Q_{2,2i}),$$

where $Q_{2,2i-1}$ and $Q_{2,2i}$ are outputs from the parent nodes of node $(3, i)$. They are orthogonal to each other. ($\mathbf{P}^{[\mathbf{3,i}]}$) is equivalent to

$$\begin{array}{lll} \hat{\mathbf{P}}^{[\mathbf{3,i}]} & \min\limits_{Q} & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ & \text{s.t.} & Q^{\mathrm{T}}Q = I_p, \\ & & \mathrm{range}(Q) \subset \mathrm{range}((Q_{2,2i-1}, Q_{2,2i})). \end{array} \tag{4.16}$$

By Lemma 4.1, $(Q_{2,2i-1}, Q_{2,2i})\hat{V}^3_i$ is an optimal solution to $\hat{\mathbf{P}}^{[\mathbf{3,i}]}$, where $\hat{V}_{3,i}$ comes from the thin-SVD of $V_p^{\mathrm{T}}(Q_{2,2i-1}, Q_{2,2i})$,

$$V_p^{\mathrm{T}}(Q_{2,2i-1}, Q_{2,2i}) = \hat{U}_{3,i}\hat{S}_{3,i}(\hat{V}_{3,i})^{\mathrm{T}}.$$

Next, let us look at $V_{3,i}$. We have

$$\begin{aligned} A^3_{i,1} &= U_{2,2i-1}S_{2,2i-1}(R_{2,2i-1})^{-1} \\ &= U_{2,2i-1}S_{2,2i-1}(V_{2,2i-1})^{\mathrm{T}}V_{2,2i-1} \\ &= A_{2,2i-1}V_{2,2i-1} \\ &= A\breve{V}_{2,2i-1}V_{2,2i-1} \\ &= AQ_{2,2i-1}. \end{aligned}$$

The second equality hold because $R_{2,2i-1}$ is an identity matrix under exact arithmetic. The fourth equality follows from (4.13). Other steps are basic substitutions. Similarly,

$$A^3_{i,2} = AQ_{2,2i}.$$

Therefore,

$$A_{3,i} = A(Q_{2,2i-1}, Q_{2,2i}).$$

Matrix $\tilde{V}_{3,i}$ is the matrix of the p-leading right singular vectors of $A_{3,i}$. By Assumption 4.1,

$$\text{range}(V_{3,i}) = \text{range}(\hat{V}_{3,i}).$$

Therefore, $Q_{3,i} = \breve{V}_{3,i}V_{3,i} = (Q_{2,2i-1}, Q_{2,2i})V_{3,i}$ is another optimal solution to $\hat{\mathbf{P}}^{[\mathbf{3,i}]}$.

Since $Q_{2,2i-1}$ and $Q_{2,2i}$ are optimal solutions to problem $\tilde{\mathbf{P}}^{[\mathbf{2,2i-1}]}$ and $\tilde{\mathbf{P}}^{[\mathbf{2,2i}]}$, respectively. $\hat{\mathbf{P}}^{[\mathbf{3,i}]}$ is equivalent to

$$\begin{aligned} \tilde{\mathbf{P}}^{[\mathbf{3,i}]} \quad \min_{Q} \quad & \text{dist}(\text{range}(V_p), \text{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \text{range}(Q) \subset \text{range}([W^{[4i-3]}, W^{[4i-2]}, W^{[4i-1]}, W^{[4i]}]). \end{aligned} \tag{4.17}$$

Recursively, we can prove that $Q_{k,i}$ is an optimal solution to

$$\begin{aligned} \tilde{\mathbf{P}}^{[\mathbf{k,i}]} \quad \min_{Q} \quad & \text{dist}(\text{range}(V_p), \text{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \text{range}(Q) \subset \text{range}([W^{[2^{k-1}i-2^{k-1}+1]}, W^{[2^{k-1}i-2^{k-1}+2]}, \ldots, W^{[2^{k-1}i]}]). \end{aligned} \tag{4.18}$$

In the extraction node, the search space in the second constraint augments to the whole space. That is to say, we solve problem

$$\begin{aligned} \tilde{\mathbf{P}}^{[\mathbf{q+1,1}]} \quad \min_{Q} \quad & \text{dist}(\text{range}(V_p), \text{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p. \end{aligned} \tag{4.19}$$

Any solution V^* of this problem satisfies that $\text{range}(V^*) = \text{range}(V_p)$. With the final extraction steps, the exact p-leading SVD of A can thus be retrieved under Assumption 4.1.

By studying this idealised case, we gained geometric insight about the way in which the approximation of the leading right singular space gets refined through the combinations. However, Assumption 4.1 does not always hold. In the remaining part of this analysis, we drop Assumption 4.1 and take local errors into consideration.

4.2.2 Global error accumulation

As discussed, we cannot expect to find the exact solution of problem $\mathbf{P}(\mathbf{W})$ directly. Instead of seeking the exact solution, we devised our method to find an approximate solution in the local computations of each node. Since in real cases, Assumption 4.1 does not hold, we must study the difference between $\tilde{V}$, which is computed by the local computation, and $\hat{V}$, the ideal solution for which we proved that algorithm terminates with the optimal solution after one sweep. In this section, we look at how the local errors accumulate through the execution tree. We will leave the analysis of local errors to Section 4.3. At this stage, we make the following assumption on local errors.

Assumption 4.2. *The distance between* $\mathrm{range}(\tilde{V})$ *and* $\mathrm{range}(\hat{V})$ *is bounded by a small positive number* ϵ *for any input Stiefel matrix* W*. That is,*

$$\mathrm{dist}(\mathrm{range}(\hat{V}), \mathrm{range}(\tilde{V})) \leq \epsilon. \tag{4.20}$$

At any combination node, there are two sources of errors:

1. We solve a perturbed problem instead of the exact problem because the input matrices are approximate solutions from the parent nodes.

2. The local computations find an approximation of the optimal solution to the perturbed optimisation problem instead of the exact solution.

First, let us look at the first source of errors. For any node with input $n \times k$ Stiefel matrix W, we aim to find an approximate solution to $\mathbf{P}(\mathbf{W})$:

$$\begin{aligned} \mathbf{P}(\mathbf{W}) \quad \min_{Q} \quad & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(W). \end{aligned} \tag{4.21}$$

In the idealised case, $\mathrm{range}(W)$ is the sum of the range spaces of optimal solutions of previous optimisation problems. However, the idealised Assumption 4.1 does not hold exactly in practice. The local errors in previous nodes cause perturbations in the input matrices. Instead of $\mathbf{P}(\mathbf{W})$ we solve problem $\mathbf{P}(\mathbf{W_{in}})$

$$\begin{aligned} \mathbf{P}(\mathbf{W_{in}}) \quad \min_{Q} \quad & \mathrm{dist}(\mathrm{range}(V_p), \mathrm{range}(Q)) \\ \text{s.t.} \quad & Q^{\mathrm{T}}Q = I_p, \\ & \mathrm{range}(Q) \subset \mathrm{range}(W_{\mathrm{in}}). \end{aligned} \tag{4.22}$$

Let us assume an upper bound for $\mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}), \mathrm{range}(W))$, i.e.,

$$\mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}), \mathrm{range}(W)) \leq \delta. \tag{4.23}$$

In view of Lemma 4.1, $Q^* = W\hat{V}$ is the optimal solution to $\mathbf{P}(\mathbf{W})$ and $\tilde{Q}^* = W_{\mathrm{in}}\hat{V}_{\mathrm{in}}$ is the optimal solution to $\mathbf{P}(\mathbf{W}_{\mathrm{in}})$, where $\hat{V}$ and $\hat{V}_{\mathrm{in}}$ are the matrices of right singular vectors of $V_p^{\mathrm{T}}W$ and $V_p^{\mathrm{T}}W_{\mathrm{in}}$, respectively. Thus,

$$\mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}), \mathrm{range}(W)) = \sqrt{1 - \sigma_k^2(W_{\mathrm{in}}^{\mathrm{T}}W)}, \tag{4.24}$$

and

$$\mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}\hat{V}_{\mathrm{in}}), \mathrm{range}(W\hat{V})) = \sqrt{1 - \sigma_p^2(\hat{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}W\hat{V})}. \tag{4.25}$$

and

$$\begin{aligned}\sigma_p(\hat{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}W\hat{V}) &= \min_{\|x\|=\|y\|=1} |y^{\mathrm{T}}\hat{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}W\hat{V}x| \\ &= \min_{\hat{x}\in\mathrm{range}(\hat{V}),\hat{y}\in\mathrm{range}(\hat{V}_{\mathrm{in}}),\|x\|=\|y\|=1} |\hat{y}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}W\hat{x}|\end{aligned}$$

On the other hand,

$$\sigma_k(W_{\mathrm{in}}^{\mathrm{T}}W) = \min_{\|x\|=\|y\|=1} |y^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}Wx|.$$

Thus, we have

$$\sigma_p(\hat{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}W\hat{V}) \geq \sigma_k(W_{\mathrm{in}}^{\mathrm{T}}W). \tag{4.26}$$

This implies

$$\mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}\hat{V}_{\mathrm{in}}), \mathrm{range}(W\hat{V}) \leq \mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}), \mathrm{range}(W)) \leq \delta. \tag{4.27}$$

Therefore, the first source of local errors is smaller than δ, given (4.23).

Next, we show that the second source of local errors add up to ϵ to the local errors.

$$\begin{aligned}& \mathrm{dist}(\mathrm{range}(W_{\mathrm{in}}\tilde{V}_{\mathrm{in}}), \mathrm{range}(W_{\mathrm{in}}\hat{V}_{\mathrm{in}})) \\ = \;& \|W_{\mathrm{in}}\tilde{V}_{\mathrm{in}}\tilde{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}} - W_{\mathrm{in}}\hat{V}_{\mathrm{in}}\hat{V}_{\mathrm{in}}^{\mathrm{T}}W_{\mathrm{in}}^{\mathrm{T}}\|_2 \\ \leq \;& \|W_{\mathrm{in}}\|_2\|\tilde{V}_{\mathrm{in}}\tilde{V}_{\mathrm{in}}^{\mathrm{T}} - \hat{V}_{\mathrm{in}}\hat{V}_{\mathrm{in}}^{\mathrm{T}}\|_2\|W_{\mathrm{in}}^{\mathrm{T}}\|_2 \\ \leq \;& \|\tilde{V}_{\mathrm{in}}\tilde{V}_{\mathrm{in}}^{\mathrm{T}} - \hat{V}_{\mathrm{in}}\hat{V}_{\mathrm{in}}^{\mathrm{T}}\|_2 \\ = \;& \mathrm{dist}(\mathrm{range}(\tilde{V}_{\mathrm{in}}), \mathrm{range}(\hat{V}_{\mathrm{in}})) \\ \leq \;& \epsilon.\end{aligned}$$

The last inequality follows from Assumption 4.2

Combing the two sources of errors,

$$\begin{aligned} & \operatorname{dist}(\operatorname{range}(W_{\text{in}}\tilde{V}_{\text{in}}, \operatorname{range}(W\hat{V}))) \\ \leq\ & \operatorname{dist}(\operatorname{range}(W_{\text{in}}\hat{V}_{\text{in}}), \operatorname{range}(W\hat{V})) + \operatorname{dist}(\operatorname{range}(W_{\text{in}}\tilde{V}_{\text{in}}), \operatorname{range}(W_{\text{in}}\hat{V}_{\text{in}})) \\ \leq\ & \delta + \epsilon. \end{aligned}$$

The distance between the computed solution $W_{\text{in}}\tilde{V}_{\text{in}}$ of the perturbed problem and the exact solution $W\hat{V}$ to the exact problem is bounded by $\delta + \epsilon$.

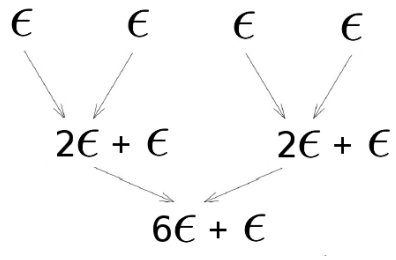

Figure 4.1: Error accumulation in a 3-level binary execution tree.

Figure 4.1 illustrates a simple example of error accumulation in a 3-level binary combination tree. In the seed nodes, there is no perturbation of the optimisation problems. The errors are one ϵ from the second sources by assumption. On the second level, node $(2, i)$ has inputs $V^2_{i,1}$ and $V^2_{i,2}$. If $Q_{1,2i-1}$ and $Q_{1,2i}$ are optimal solutions to problem $\mathbf{P}^{[\mathbf{1,2i-1}]}$ and $\mathbf{P}^{[\mathbf{1,2i}]}$ respectively, then

$$\operatorname{dist}(\operatorname{range}(V^2_{i,1}), \operatorname{range}(Q_{1,2i-1})) = \sqrt{1 - (\sigma_p((V^2_{i,1})^{\mathrm{T}} Q_{1,2i-1}))^2} \leq \epsilon,$$

and

$$\operatorname{dist}(\operatorname{range}(V^2_{i,2}), \operatorname{range}(Q_{1,2i})) = \sqrt{1 - (\sigma_p((V^2_{i,2})^{\mathrm{T}} Q_{1,2i}))^2} \leq \epsilon.$$

The perturbation of input $\breve{V}^2_i = (V^2_{i,1}, V^2_{i,2})$ is

$$\begin{aligned} & \operatorname{dist}(\operatorname{range}((V^2_{i,1}, V^2_{i,2})), \operatorname{range}((Q_{1,2i}, Q_{1,2i-1}))) \\ =\ & \|(V^2_{i,1}, V^2_{i,2})(V^2_{i,1}, V^2_{i,2})^{\mathrm{T}} - (Q_{1,2i}, Q_{1,2i-1})(Q_{1,2i}, Q_{1,2i-1})^{\mathrm{T}}\|_2 \\ =\ & \|V^2_{i,1}(V^2_{i,1})^{\mathrm{T}} + V^2_{i,2}(V^2_{i,2})^{\mathrm{T}} - Q_{1,2i-1}(Q_{1,2i-1})^{\mathrm{T}} - Q_{1,2i}(Q_{1,2i})^{\mathrm{T}}\|_2 \\ \leq\ & \|V^2_{i,1}(V^2_{i,1})^{\mathrm{T}} - Q_{1,2i-1}(Q_{1,2i-1})^{\mathrm{T}}\|_2 + \|V^2_{i,2}(V^2_{i,2})^{T} - Q_{1,2i}(Q_{1,2i})^{\mathrm{T}}\|_2 \\ \leq\ & 2\epsilon. \end{aligned}$$

The error of the first source of errors is therefore 2ϵ. Another ϵ comes from the second source of errors. Overall, the error is bounded by 3ϵ for the output of a node

in the second level. On the third level, same argument shows that the error is 7ϵ with 6ϵ stemming from the first source of errors and one ϵ from the second source of errors. We summarize the conclusion of the analysis into Theorem 4.1

Theorem 4.1. *The overall local error of the approximate leading right singular space is bounded by $(s-1)\epsilon$, where s is the number of seed nodes of the binary execution tree.*

This analysis shows that the growth rate of the errors is linear in the number of seed nodes. Our problem is thus reduced to finding the bound ϵ of local errors. This is the goal of the next section.

4.3 Analysis of local errors

In this section, we focus on finding the error bound ϵ of Assumption 4.2. The local analysis includes 2 steps:

i) The spaces range($\bar{V}$) and range($\hat{V}$) are the same space.

ii) range($\check{V}$) is close to range($\bar{V}$), hence close to range($\hat{V}$).

In the following lemma, we prove that range($\bar{V}$) is the same as range($\hat{V}$).

Lemma 4.2. *$\bar{V}$ and $\hat{V}$ are $n \times p$ Stiefel matrices described in (4.4) and (4.5). Then,* range($\bar{V}$) = range($\hat{V}$).

Proof. By substitution, we find that

$$A_p W = U_p S_p V_p^{\mathrm{T}} W = U_p S_p \hat{U} \hat{S} \hat{V}^{\mathrm{T}}. \tag{4.28}$$

If $S_p \hat{U} \hat{S} = U_s S_s V_s^T$ is the SVD of $S_p \hat{U} \hat{S}$, then

$$A_p W = U_p S_p \hat{U} \hat{S} \hat{V}^{\mathrm{T}} = U_p U_s S_s V_s^{\mathrm{T}} \hat{V}^{\mathrm{T}} = (U_p U_s) S_s (\hat{V} V_s)^{\mathrm{T}}. \tag{4.29}$$

Note that $S_p \hat{U} \hat{S}$ is a $p \times p$ square matrix. U_s and V_s are therefore orthogonal matrices. $U_p U_s$ and $\hat{V} V_s$ are Stiefel matrices. Thus, $A_p W = (U_p U_s) S_s (\hat{V} V_s)^{\mathrm{T}}$ is a SVD of $A_p W$. Next, we have range($\bar{V}$) = range($\hat{V} V_s$), because both sides equal the p-leading right singular space of $A_p W$. On the other hand, range($\hat{V}$) = range($\hat{V} V_s$). As a result, it follows that

$$\text{range}(\hat{V}) = \text{range}(\bar{V}). \tag{4.30}$$

□

Lemma 4.2 settles the first step. For step 2, we note that $\tilde{V}$ and $\bar{V}$ are the matrices consisting of the p leading right singular vectors of AW and A_pW, respectively. We start from the following lemma.

Lemma 4.3. *$|\tilde{\sigma}_l - \bar{\sigma}_l| \leq \sigma_{p+1}$, where $\tilde{\sigma}_l$ and $\bar{\sigma}_l$, $l = 1, \ldots, p$, are the singular values of AW and A_pW, respectively.*

Proof.

$$\begin{aligned}
\|A_pW - AW\|_2 &= \|U_pS_pV_p^{\mathrm{T}}W - USV^{\mathrm{T}}W\|_2 \\
&= \|U_pS_pV_p^{\mathrm{T}}W - [U_p, U_c]\begin{bmatrix} S_p & \\ & S_c \end{bmatrix}[V_p, V_c]^{\mathrm{T}}W\|_2 \\
&= \|U_pS_pV_p^{\mathrm{T}}W - (U_pS_pV_p^{\mathrm{T}} + U_cS_cV_c^{\mathrm{T}})W\|_2 \\
&= \|U_cS_cV_c^{\mathrm{T}}W\|_2 \\
&\leq \|U_c\|_2\|S_c\|_2\|V_c^{\mathrm{T}}W\|_2 \\
&\leq \sigma_{p+1}.
\end{aligned}$$

In the second equality, we split U, S and V into $[U_p, U_c]$, $\begin{bmatrix} S_p & \\ & S_c \end{bmatrix}$ and $[V_p, V_c]^{\mathrm{T}}W$, respectively

By perturbation theory of SVD [102], for $l = 1, \ldots, p$, it is the case that

$$|\tilde{\sigma}_l - \bar{\sigma}_l| \leq \sigma_{p+1}. \tag{4.31}$$

□

Before continuing the analysis, we review the Wedin's Theorem.

Definition 4.2. *Let $\mathcal{X}$ and $\mathcal{Y}$ be two real vector spaces of dimension k, and let X and Y be the orthonormal bases for $\mathcal{X}$ and $\mathcal{Y}$, respectively. Let $\gamma_1 \geq \gamma_2 \geq \ldots \geq \gamma_k$ be the singular values of $X^{\mathrm{T}}Y$. The canonical angles between $\mathcal{X}$ and $\mathcal{Y}$ are defined as,*

$$\theta_i = \cos^{-1}(\gamma_i).$$

Theorem 4.2 (Wedin [100])**.** *Let B and $\tilde{B}$ be two $m \times n$ matrices with SVDs*

$$B = U^b S^b (V^b)^{\mathrm{T}}$$

and

$$\tilde{B} = \tilde{U}^b \tilde{S}^b (\tilde{V}^b)^{\mathrm{T}}.$$

Let $\{\sigma_1^b, \ldots, \sigma_p^b, \sigma_{p+1}^b, \ldots, \sigma_n^b\}$ *and* $\{\tilde{\sigma}_1^b, \ldots, \tilde{\sigma}_p^b, \tilde{\sigma}_{p+1}^b, \ldots, \tilde{\sigma}_n^b\}$ *be the singular values of* B *and* $\tilde{B}$*, respectively. Suppose that* $\tilde{\sigma}_{p+1}^b < \sigma_p^b$ *and* $\sigma_{p+1}^b < \tilde{\sigma}_p^b$*. Let us write*

$$\delta = \max(|\tilde{\sigma}_{p+1}^b - \sigma_p^b|, |\sigma_{p+1}^b - \tilde{\sigma}_p^b|) \tag{4.32}$$

Let Φ^b *be the matrix of principal angles between* $\mathrm{range}(U_p^b)$ *and* $\mathrm{range}(\tilde{U}_p^b)$*, and* Θ^b *the matrix of principal angles between* $\mathrm{range}(V_p^b)$ *and* $\mathrm{range}(\tilde{V}_p^b)$*. For any unitarily invariant norm* $\|.\|$*, it is then true that*

$$\max\{\|\sin\Phi^b\|, \|\sin\Theta^b\|\} \leq \frac{\max\{\|R\|, \|S\|\}}{\delta}, \tag{4.33}$$

where $R = B\tilde{V}_p^b - \tilde{U}_p^b\tilde{S}_p^b$ *and* $S = B^{\mathrm{T}}\tilde{U}_p^b - \tilde{V}_p^b\tilde{S}_p^b$*.*

Corollary 4.1. *If* Θ *is the matrix of principal angles between* $\mathrm{range}(\tilde{V}_p)$ *and* $\mathrm{range}(\bar{V}_p)$*, and if* $\tilde{\sigma}_p > \bar{\sigma}_{p+1}$ *and* $\bar{\sigma}_p > \tilde{\sigma}_{p+1}$*, then,*

$$\|\sin\Theta\| \leq \frac{\sigma_{p+1}}{\tilde{\sigma}_p - \sigma_{p+1}}. \tag{4.34}$$

Proof. Let $B = AW$ and $\tilde{B} = A_pW$ in Wedin's theorm. Then,

$$\delta \geq \tilde{\sigma}_p - \bar{\sigma}_{p+1} \geq \tilde{\sigma}_p - \sigma_{p+1}. \tag{4.35}$$

The last inequality follows from Cauchy's interlacing theorem.

$$\begin{aligned}
\|R\|_2 &= \|AW\bar{V} - \bar{U}\bar{S}\|_2 \\
&= \|AW\bar{V} - A_pW\bar{V}\|_2 \\
&\leq \|(AW - A_pW)\|_2\|\bar{V}\|_2 \\
&= \|U_cS_cV_c^{\mathrm{T}}W\|_2\|\bar{V}\|_2 \\
&\leq \sigma_{p+1}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
\|S\|_2 &= \|(AW)^{\mathrm{T}}\bar{U} - \bar{V}\bar{S}\|_2 \\
&= \|(AW)^{\mathrm{T}}\bar{U} - (A_pW)^{\mathrm{T}})\bar{U}\|_2 \\
&\leq \|(AW)^{\mathrm{T}} - (A_pW)^{\mathrm{T}})\|_2\|\bar{U}\|_2 \\
&= \|(U_cS_cV_c^{\mathrm{T}}W)^{\mathrm{T}}\|_2\|\bar{U}\|_2 \\
&\leq \sigma_{p+1}.
\end{aligned}$$

Therefore,

$$\|\sin\Theta\|_2 \leq \frac{\max\{\|R\|_2, \|S\|_2\}}{\delta} \leq \frac{\sigma_{p+1}}{\tilde{\sigma}_p - \sigma_{p+1}}. \tag{4.36}$$

□

We have shown that $\|\sin\Theta\|$ is bounded by $\frac{\sigma_{p+1}}{\tilde{\sigma}_p - \sigma_{p+1}}$. In Assumption 4.2, we assumed that

$$\text{dist}(\text{range}(\hat{V}), \text{range}(\tilde{V})) \leq \epsilon. \tag{4.37}$$

Because

$$\text{dist}(\text{range}(\hat{V}), \text{range}(\tilde{V})) = \text{dist}(\text{range}(\bar{V}), \text{range}(\tilde{V})) = \sin(\theta_p)$$

is smaller than $\|\sin\Theta\|$, $\epsilon = \frac{\sigma_{p+1}}{\tilde{\sigma}_p - \sigma_{p+1}}$ is an upper bound of $\text{dist}(\text{range}(\hat{V}), \text{range}(\tilde{V}))$.

To complete this analysis, we need to show how small ϵ is. In the formula of ϵ, the $\tilde{\sigma}_p$ is the p-th singular value of AW, which depends on input W. In the following section, we will investigate $\tilde{\sigma}_p$ for the case where W is in random position. In practice, this is not the case, but we can show empirically that the local errors on any given level behave as though this assumption were true.

4.3.1 The local error at a node with random input matrix W

In our previous analysis, we have shown that the local error satisfies

$$\epsilon \leq \frac{\sigma_{p+1}}{\tilde{\sigma}_p - \sigma_{p+1}}$$

This poses a dilemma, since we do not know the value of $\tilde{\sigma}_p$. The following lemma points out a way to overcome this dilemma, via sufficient condition which, if satified, implies

$$\tilde{\sigma}_p \geq \sigma_p \cos\theta_p,$$

so that the upper bound

$$\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \cos\theta_p - \sigma_{p+1}} \tag{4.38}$$

applies, which is independent of $\tilde{V}$ and only depends on the distribution of singular value of A and the alignment of the space range(W) with the singular vectors.

Let θ_i be the angle between the singular vector v_i and the space range(W) for $i = 1, \ldots, n$. In the following lemma, we show that under the assumption $\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p$, for $i = 1, \ldots, p$, we have $\tilde{\sigma}_p \geq \sigma_p \cos\theta_p$.

Lemma 4.4. *Let W be an $n \times k$ Stiefel matrix, and $\theta_1, \theta_2, \ldots, \theta_p$ the angles between v_i and* range(W)*. If $\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p$, for $i = 1, \ldots, p$, then $\tilde{\sigma}_p \geq \sigma_p \cos\theta_p$.*

Proof. Let $z_i = W^{\mathrm{T}} v_i$, then $\|z_i\| = \cos\theta_i$. Let $S = \text{range}\{z_1, z_2, \ldots, z_p\}$, then $\dim(S) = p$. By Min-max Theorem for singular values,

$$\tilde{\sigma}_p \;=\; \sigma_p(AW) \geq \min_{y \in S, \|y\|=1} \|AWy\|.$$

Because $\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p$, for $i = 1, \ldots, p$, $\|AWy\| \geq \sigma_p \cos\theta_p$, for any $y \in S, \|y\| = 1$. Therefore,

$$\tilde{\sigma}_p \quad \geq \sigma_p \cos\theta_p.$$

□

The sufficient condition of Lemma 4.4 cannot be verified, but we will now show that for Stiefel matrices W chosen uniformly at random, it applies with high probability. The Stiefel matrices W that occur in the course of a sweep of the algorithm are not random and not independent of the W occurring at other nodes, but our experiments of Section 4.4 show that in practice the error bounds behave as though the W were chosen i.i.d. uniformly at random. Hence, an analysis on the basis of assuming random W has high explanatory power for why the local error can be assumed constant at all nodes.

We start with a useful concentration of measure inequality.

Lemma 4.5. *Let S be a random k-dimensional subspace of $\mathbb{R}^n$ with uniform distribution on the Grassmannian. Let $Y \in \mathbb{R}^n$ be a unit vector, and L the length of the projection of Y into S. Then, the following hold:*
If $\beta < 1$, then

$$P\left[L \leq \frac{\beta k}{n}\right] \leq \exp\left(\frac{k}{2}(1 - \beta + \ln\beta)\right). \tag{4.39}$$

If $\beta > 1$, then

$$P\left[L \geq \frac{\beta k}{n}\right] \leq \exp\left(\frac{k}{2}(1 - \beta + \ln\beta)\right). \tag{4.40}$$

Proof. See Lemma 2.2 in [28]. □

The previous lemma now implies that the random angles θ_i are all concentrated around their expected value:

Lemma 4.6. *Let $\tau \in (0, 1)$ and W an $n \times k$ Stiefel matrix chosen uniformly at random. Let p_i be the projection of v_i into* range(W). *Then the following inequality applies,*

$$\mathrm{P}\left[\left|\|p_i\| - \frac{k}{n}\right| \leq \tau\frac{k}{n}\right] \geq 1 - \exp\left\{\frac{k}{2}\left(-\tau + \ln(1+\tau)\right)\right\} - \exp\left\{\frac{k}{2}\left(\tau + \ln(1-\tau)\right)\right\}.$$

Proof. Setting $\beta = 1 - \tau$ and applying the first part of Lemma 4.5, we have

$$P\left[\|p_i\| \leq (1-\tau)\frac{k}{n}\right] \leq \exp\left(\frac{k}{2}(\tau + \ln(1-\tau))\right). \tag{4.41}$$

Similarly, setting $\beta = 1 + \tau$, we get

$$P\left[\|p_i\| \geq (1+\tau)\frac{k}{n}\right] \leq \exp\left(\frac{k}{2}(-\tau + \ln(1+\tau))\right). \tag{4.42}$$

Therefore, by the union bound,

$$P\left[\left|\|p_i\| - \frac{k}{n}\right| \leq \tau\frac{k}{n}\right] \geq 1 - \exp\left(\frac{k}{2}(-\tau + \ln(1+\tau))\right) - \exp\left(\frac{k}{2}(\tau + \ln(1-\tau))\right). \tag{4.43}$$

□

For the purposes of the remaining steps in our analysis, we need to condition on the value of θ_p and distinguish two different cases. Let us therefore define

$$\Theta := \left\{\theta_p : \left|\cos\theta_p - \frac{k}{n}\right| \leq \tau\frac{k}{n}\right\},$$

where τ is chosen as in Lemma 4.6. For convenience of notation, we also define the quantity

$$\Delta := \frac{n\cos\theta_p}{k} \times \min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}.$$

We note that when $\theta_p \in \Theta$, then the following bounds apply,

$$(1-\tau)\min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i} \leq \Delta \leq (1+\tau)\min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}. \tag{4.44}$$

For the purposes of the next lemma, we need to constrain the value of τ slightly further: let

$$0 < \tau < \frac{\min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}}{1 + \min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}} \tag{4.45}$$

and define

$$\xi = -\tau + (1-\tau) \times \min_{i=1,\ldots,p} \frac{\sigma_i - \sigma_p}{\sigma_i}.$$

Then (4.45) guarantees that $\xi > 0$.

Lemma 4.7. *Let τ satisfy* (4.45). *Then*

$$\mathrm{P}\left[\sigma_i\cos\theta_i \geq \sigma_p\cos\theta_p \ (i = 1,\ldots,p-1) \mid \theta_p \in \Theta\right] \geq 1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}.$$

Proof. For $i = 1, \ldots, p-1$ and $\theta_p \in \Theta$, we have

$$\begin{aligned}
\mathrm{P}\left[\sigma_i \cos\theta_i < \sigma_p \cos\theta_p \mid \theta_p\right] &= \mathrm{P}\left[\cos\theta_i - \cos\theta_p < -\cos\theta_p \times \frac{\sigma_i - \sigma_p}{\sigma_i} \mid \theta_p\right] \\
&\leq \mathrm{P}\left[\cos\theta_i - \cos\theta_p < -\Delta \times \frac{k}{n} \mid \theta_p\right] \\
&\leq \mathrm{P}\left[\|p_i\| - \frac{k}{n} < \frac{k}{n}(\tau - \Delta) \mid \theta_p\right] \\
&\leq \mathrm{P}\left[\|p_i\| - \frac{k}{n} < \frac{k}{n}\left(\tau - (1-\tau) \times \min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}\right) \mid \theta_p\right] \\
&= \mathrm{P}\left[\|p_i\| - \frac{k}{n} < \frac{k}{n}\left(\tau - (1-\tau) \times \min_{i=1,\ldots,p-1} \frac{\sigma_i - \sigma_p}{\sigma_i}\right)\right] \\
&\leq \exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}.
\end{aligned}$$

The second inequality holds because that $\theta_p \in \Theta$. The third inequality follows (4.44) and the last inequality follows (4.45) and (4.42).

Therefore, for all $\theta_p \in \Theta$ we have

$$\begin{aligned}
\mathrm{P}\left[\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p,\ (i = 1, \ldots, p-1) \mid \theta_p\right] &\geq 1 - \sum_{i=1}^{p-1} \mathrm{P}\left[\sigma_i \cos\theta_i < \sigma_p \cos\theta_p \mid \theta_p\right] \\
&\geq 1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}.
\end{aligned}$$

Finally, integrating out θ_p, and writing $f_\Theta(\theta_p)$ for the density of the conditional distribution of θ_p conditioned on the event $\{\theta_p \in \Theta\}$, we obtain

$$\begin{aligned}
&\mathrm{P}\left[\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p\ (i = 1, \ldots, p-1) \mid \theta_p \in \Theta\right] \\
&\qquad = \int_\Theta \mathrm{P}\left[\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p,\ (i = 1, \ldots, p-1) \mid \theta_p\right] \times f_\Theta(\theta_p)\,\mathrm{d}\,\theta_p \\
&\qquad \geq \left(1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}\right) \times \int_\Theta f_\Theta(\theta)\,\mathrm{d}\,\theta_p,
\end{aligned}$$

which proves the claim, since the integral in the last expression equals 1. □

Combining all the elements derived above, we finally obtain a probabilistic bound on the local error of the algorithm:

Theorem 4.3. *Let τ and ξ be chosen as above, and let ϵ denote the local error defined in (4.20) under the assumption that W is an $n \times k$ Stiefel matrix chosen uniformly*

at random. Then

$$\begin{aligned}
&\mathrm{P}\left[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \times \frac{k}{n}(1-\tau) - \sigma_{p+1}}\right] \geq \\
&\quad \geq \left(1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}\right) \\
&\quad \times \left(1 - \exp\left\{\frac{k}{2}\left(-\tau + \ln(1+\tau)\right)\right\} - \exp\left\{\frac{k}{2}\left(\tau + \ln(1-\tau)\right)\right\}\right).
\end{aligned}$$

Proof. By Lemmas 4.4 and 4.7, and by (4.38), we have

$$\begin{aligned}
&\mathrm{P}\Big[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \cos\theta_p - \sigma_{p+1}} \mid \Theta\Big] \geq \\
&\quad \geq \mathrm{P}\left[\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p \ (i = 1, \ldots, p-1) \mid \Theta\right] \\
&\quad \geq 1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}. \qquad (4.46)
\end{aligned}$$

Furthermore, Lemma 4.6 applied to p_p implies

$$\mathrm{P}\left[\theta_p \in \Theta\right] \geq 1 - \exp\left\{\frac{k}{2}\left(-\tau + \ln(1+\tau)\right)\right\} - \exp\left\{\frac{k}{2}\left(\tau + \ln(1-\tau)\right)\right\}. \qquad (4.47)$$

It follows that

$$\begin{aligned}
&\mathrm{P}\Big[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \times \frac{k}{n}(1-\tau) - \sigma_{p+1}}\Big] = \\
&\quad = \mathrm{P}\left[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \times \frac{k}{n}(1-\tau) - \sigma_{p+1}} \mid \Theta\right] \times \mathrm{P}[\Theta] \\
&\qquad + \mathrm{P}\left[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \times \frac{k}{n}(1-\tau) - \sigma_{p+1}} \mid \Theta^c\right] \times \mathrm{P}[\Theta^c] \\
&\quad \geq \mathrm{P}\left[\epsilon \leq \frac{\sigma_{p+1}}{\sigma_p \cos\theta_p - \sigma_{p+1}} \mid \Theta\right] \times \mathrm{P}[\Theta] + 0 \\
&\quad \geq \left(1 - (p-1)\exp\left\{\frac{k}{2}\left(\xi + \ln(1-\xi)\right)\right\}\right) \\
&\quad \times \left(1 - \exp\left\{\frac{k}{2}\left(-\tau + \ln(1+\tau)\right)\right\} - \exp\left\{\frac{k}{2}\left(\tau + \ln(1-\tau)\right)\right\}\right).
\end{aligned}$$

The last inequality follows (4.46) and (4.47). □

We note that for the error bound of Theorem 4.3 to apply, we need the spectral gap to be large enough for

$$\sigma_p \times \frac{k}{n}(1-\tau) > \sigma_{p+1}.$$

This also highlights the benefit of working with larger subspaces than dimension p, as $k >> p$ leads to a quantifiably smaller local error.

Level	1		2	
	Average	Std dev	Average	Std dev
1	1.41E-03	1.13E-03	1.40E-03	1.14E-03
2	6.09E-05	5.27E-05	6.02E-05	5.42E-05
3	2.18E-07	2.23E-07	-	-
Level	3		4	
	Average	Std dev	Average	Std dev
1	1.53E-03	1.19E-03	1.52E-03	1.15E-03
2	-	-	-	-
3	-	-	-	-

Table 4.1: Average local error and the standard deviation of the nodes on each level of 3-level binary execution trees.

4.4 Numerical experiments

In the first set of numerical experiments, we looked at how big the local error is. The test matrices are of size 1280×1280 and constructed from the singular vectors and singular values that are randomly generated. Table 4.1 lists the average local errors, i.e. $\text{dist}(\tilde{V}, \hat{V})$, and its standard deviations of 100 tests in all of the nodes in each level of a 3-level binary execution tree. Table 4.2 is a list of average errors and its standard deviation of the nodes on each level of 6-level execution trees. Figure 4.2 is the histogram of local errors of all seed nodes on a 6-level binary execution tree over 100 tests. In Section 4.2.2, we assumed that all of the local errors are bounded by the same bound ϵ. However, in practice, they get much smaller on deeper levels. This makes the linear accumulation rate of local errors very conservative.

In the second set of experiments, we checked whether assumptions

$$\sigma_i \cos\theta_i \geq \sigma_p \cos\theta_p, (i = 1, \ldots, p) \tag{4.48}$$

jointly hold with high probability.

We report the empirical estimates of the probabilities that (4.48) jointly hold on each level of a 4-level binary execution tree. The results of the cases where decrease factor is $\alpha = 2$, 1.2, 1.1 and 1.01 is listed in Table 4.3. They are computed from 1000 tests.

The W that appears at the different nodes on a fixed level are not independent. When decrease rate of the singular values is not very close to 1, although the input matrices in these tests are not i.i.d., the assumptions still hold in most cases.

Level	3		4	
	Average	Std dev	Average	Std dev
1	1.46E-03	1.15E-03	2.27E-03	1.95E-03
2	6.05E-05	5.33E-05	1.76E-04	1.70E-04
3	2.18E-07	2.23E-07	6.70E-05	5.93E-05
4	-	-	3.42E-07	3.36E-07
5	-	-	-	-
6	-	-	-	-
Level	5		6	
	Average	Std dev	Average	Std dev
1	3.27E-03	2.63E-03	4.96E-03	4.15E-03
2	3.56E-04	3.12E-04	8.17E-04	7.43E-04
3	1.65E-04	1.48E-04	3.70E-04	3.37E-04
4	6.10E-05	5.35E-05	1.80E-04	1.57E-04
5	4.39E-07	4.65E-07	6.77E-05	6.96E-05
6	-	-	8.51E-07	8.61E-07

Table 4.2: Average local error and the standard deviation of the nodes on each level of 6-level binary execution trees.

level \ α	2	1.2	1.1	1.01
1	1	1	0.722	0.605
2	1	1	0.956	0.609
3	1	1	1	0.680
4	1	1	1	0.856

Table 4.3: Probabilities that inequalities (4.48) jointly hold on each level.

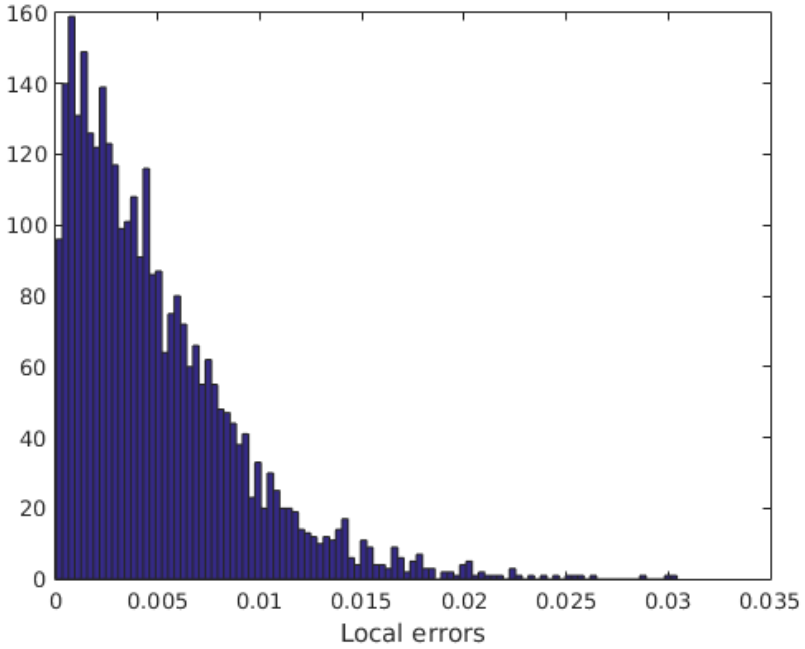

Figure 4.2: Histogram of the local errors of all seed nodes on a 6-level binary execution tree over 100 tests.

In conclusion, we proved that in each node, an approximate solution of an optimisation problem is found by local computations. The accumulating rate of the local errors is linear to the number of seed nodes of the execution tree. The local errors are bounded by $\frac{\sigma_{p+1}}{\sigma_p \times \frac{k}{n}(1-\tau)-\sigma_{p+1}}$ with high probability in the case where the input matrix W is in random general position.

Chapter 5

Numerical Experiments and Discussions of the Variants of the Algorithm

In this chapter, we discuss several practical issues and report numerical results on the parallel SVD algorithm. We will discuss the loose coupling properties and variants of the algorithm, along with numerical examples. We aim at giving the potential users basic ideas on how to choose the parameters and configurations of the algorithm, depending on their problems and computing resources.

5.1 Discussions of several practical issues

In previous chapters, we introduced the framework of our algorithm and presented its theoretical analysis. While the loosely coupled properties offer users flexibility to exploit computing resources depending on their availability, several practical issues arise.

5.1.1 Initialisation and warm-start

The easiest way to initialise the algorithm is to use column-partition and distribute groups of columns to the seed nodes, as described in Section 2.1.5. In this way, we do not have to form the input matrix W explicitly and can use the submatrix which is formed by a subset of columns of A as input AW in seed nodes.

If users find that the input matrices $(AQ_{\text{in}}, Q_{\text{in}})$ are still too large to be processed in seed nodes even after column-splitting, random column selection, such as the techniques stated in [34], can be used to reduce the dimensions of the input matrices for seed nodes, hence reduce the local computational costs in seed nodes.

If prior information is known to the users that could be used as an educated guess of the leading part SVD of A, the initial matrices in seed nodes can be customized by the users to warm start the algorithm. One of the possible circumstances in which warm-starting can be used is iterative methods where SVD is not applied once but multiple times to a sequence of related iterations. In many optimisation problems, the optimal solutions are approximated by iterative methods. For some methods, the matrices in consecutive iterations, for which we compute the leading part SVD, are closely related. Information about the SVDs of these matrices from previous iterations can be incorporated into the SVD computations of new iterations.

5.1.2 Sequential data

One of the advantages of the described algorithm is that it can update the approximation result without reprocessing all of the old data when new data is received or part of the old data is changed. If new data-sets arrive, we only need to add one or more seed nodes and combine the results with the previous results. If only part of the data is changed, only related nodes need updates.

This property is very useful in applications in which new data comes in while the previous data has been processed or is being processed. One of the examples is real-time analysis [31], [24], which is greatly used in e-commence and finance. Unlike offline analysis, data constantly changes in real-time analysis. Data analysis is required to be fast and results must be returned rapidly. The capacity to update results without processing old data is crucial for this type of analysis. This makes the described algorithm an ideal tool for such applications.

5.1.3 Multi-sweep

For applications with high precision requirements, the algorithm can be used in multi-sweep mode. An instantiation of a work flow with multiple sweeps over a binary combination tree is shown in Figure 5.1. The rectangles in the upper left-hand corner represent the seed data-sets that are re-injected at different levels when they get combined with the output from a sweep across the combination tree. The final output is represented by the rectangle in the right-hand bottom corner of the figure. In this case, the node on the third layer sends its output to a further set of combination nodes that combine this data with the initial seed node data and take another sweep. This process can be repeated until results meet a termination criterion, such as that

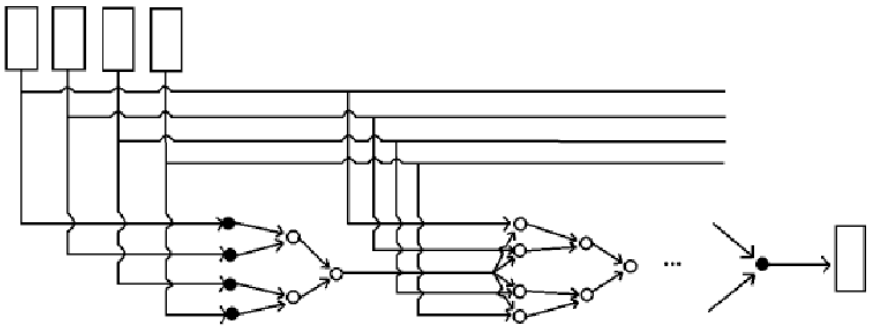

Figure 5.1: An instantiation of a work flow with multiple sweeps over a binary combinationn tree.

a minimum tolerance on the progress being satisfied. Numerical tests will follow in this chapter.

5.1.4 Sparsity and structure

Many matrix decomposition methods for large scale matrices ([46, 11, 67, 52]) rely on exploiting a sparsity structure. In this algorithm, we do not require the data matrix to be sparse. General matrices are considered. If the matrix is sparse, the local computations deal with sparse matrices, and all the local steps can be replaced by their sparsity-exploiting variants. In this situation, sparsity usually only occurs at the seed node level, but this is where it is most beneficial, since the number of columns considered in every seed data-set is much greater than $2p$, while in subsequent nodes it is $2p$.

Structured matrices are also an important class of matrices. Though not always sparse, specific methods have been designed to exploit special structure [18]. Similarly to sparsity, structured methods can be used to boost the local computations, especially in the seed nodes where the structure of the matrices are preserved well, compared with the subsequent nodes.

5.1.5 Oversampling in local computations

In low-rank approximation problems, we seek an $m \times k$ Stiefel matrix Q, with $k = k(\epsilon)$, such that,

$$\|A - QQ^{\mathrm{T}}A\|_2 \leq \epsilon. \tag{5.1}$$

This is the fixed-precision approximation problem. The number of sampling columns k is a function of the required precision ϵ.

A related problem is the fixed-rank approximation problem. Given a rank k and an oversampling parameter l, we want to find an $m \times (k+l)$ Stiefel matrix Q such that

$$\|A - QQ^{\mathrm{T}}A\|_2 \approx \min_{rank(X)\leq k} \|A - X\|_2. \tag{5.2}$$

In practice, a small level of additional sampling is enough ($l = 5$ or 10). We could take advantage of this property in local computations. As discussed in Chapter 4, the local errors decease as a function of l. Instead of computing exact p-leading SVD in the local SVD step, a small number of additional singular values and singular vectors are calculated. With this bit of extra computing efforts, this oversampling technique improves the performance of the local computations.

5.1.6 Pass-efficiency

Over the past decades, the amount of memory space in disks has increased fast due to hardware developments, while RAM and computing speeds have increased less rapidly. The power of traditional algorithms to process large data-sets lags far behind the amount of data that can be stored in disks. Data needs to be passed between fast memories and disks, which is both time and energy consuming. Therefore, the topic of pass-efficiency is important in modern matrix computing ([33], [82] and [25]).

In the pass-efficient model, we assume that data is stored on a disk. Algorithms access the data via a pass over the data. An algorithm is pass-efficient if it needs only a small number of passes and sublinear additional time and space in computing. In matrix computing, an algorithm is pass-efficient if it requires additional time and space that is $O(m+n)$ or $O(1)$. In its single sweep variant, the described method only needs to access the original matrix once in the seed nodes and this takes $O(\frac{kmn}{2^q})$ time in parallel, where $q+1$ is the number of levels. It is thus a one-pass method.

5.2 A random matrix model for test data generation

In the following sections, we will discuss the performance of our algorithm and its variants. We carried out a series of experiments in Matlab. The tests were conducted on a desktop with 4 AMD Phenom(tm) II X4 945 Processors and 3.6 GB of memory.

The first sets of tests were conducted with artificial input matrices, that were constructed by random generating the singular vectors and singular values separately. A random matrix $A \in \mathbb{R}^{m\times n}$ was constructed as follows,

$$A = U_k S_k V_k^{\mathrm{T}}, \tag{5.3}$$

where k is an integer greater than or equal to p but smaller than m and n. The Stiefel matrices $U_k \in \mathbb{R}^{m\times k}$ and $V_k \in \mathbb{R}^{n\times k}$ were generated from the orthogonalisation of i.i.d. standard Gaussian entries of the same dimensions. $S_k \in \mathbb{R}^{k\times k}$ was a nonnegative diagonal matrix with diagonal entries $\sigma_1, \ldots, \sigma_k$, that were determined by a set of parameters $\{\sigma_1, \alpha, \beta, \eta\}$, where $0 < \beta, \eta < 1$ and $\alpha > 1$: given σ_i, σ_{i+1} was determined as follows,

$$\sigma_{i+1} = \begin{cases} \sigma_i/\alpha & \text{with probability } \eta \\ \beta\sigma_i/\alpha & \text{with probability } 1-\eta. \end{cases} \tag{5.4}$$

Thus, α was the decrease factor of the singular values.

5.3 First empirical performance comparisons

In the first numerical experiment, we report empirical performance comparisons of the parallel SVD algorithm with the leading part SVD solver *svds* in Matlab which uses an implicitly restarted Lanczos method described in [5]. To make the comparisons fair, we used Matlab's *svds* for the local SVD computations of the parallel algorithm as well.

We tested on matrices with 3000 rows and different numbers of columns, ranging from 800 to 4000. The test matrices were randomly generated as stated in Section 5.2. The parameter settings were $\sigma_1 = 100$, $\alpha = 5$, $\beta = 0.7$, $\eta = 0.8$ and $k = 10$. We computed The first 5 singular values and vectors, using the binary combination tree for singular space aggregation.

Typical running times at individual nodes are listed in Table 5.1. The running times for seed nodes were relatively longer and more heterogeneous. The reason is that the input matrices in seed nodes were fatter than in other nodes. The column number of the matrix for which SVD is computed in the seed nodes is n divided by the number of seed nodes, while the same number in other nodes is 10. When computations in the first level are finished, the dimensions of the matrices reduce greatly. Therefore, the running times at the other levels are much shorter compared to the first level.

We tested Algorithm 10 with 8 seed nodes and simulated the parallel procedure with 1, 2, 4 and 8 cores. Figure 5.2 and Table 5.2 compare the running times of Matlab

Level	1	2	3	4	5	6	7	8
1	0.5124	0.2436	0.1432	0.1655	0.1158	0.2521	0.2349	0.0981
2	0.0148	0.0143	0.0146	0.0142	-	-	-	-
3	0.0144	0.0149	-	-	-	-	-	-
4	0.0158	-	-	-	-	-	-	-

Table 5.1: Typical running times at 8 individual nodes in Matlab simulation of the parallel algorithm with 3-level binary execution tree.

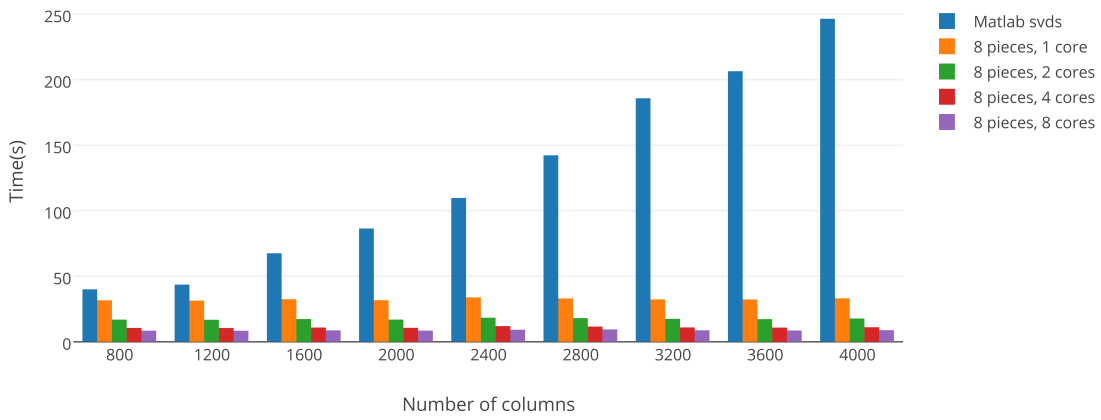

Figure 5.2: The running time comparison on matrices with increasing number of column vectors.

svds and the parallel algorithm with different numbers of cores. The speed-up rates are shown in Figure 5.3 and Table 5.3.

As is shown in Table 5.2, the computing time of Matlab *svds* grows with the increasing of the number of columns, while the computing time of the parallel algorithm stays at the same level. This is because that only small size SVD computation is conducted in the local computations of the parallel algorithm. The time used for SVD computations is only a small fraction of the total time, compared to the time used in data processing and warming up of the computation methods, which is similar for matrices of different dimensions. Therefore, the particular variant used does not have great impact on the running times.

No. of columns	svds	1 core	2 cores	4 cores	8 cores
800	40.00	31.66	16.91	10.58	8.47
1200	43.60	31.40	16.79	10.50	8.42
1600	67.53	32.46	17.35	10.87	8.70
2000	86.44	31.70	16.93	10.64	8.52
2400	109.73	33.87	18.35	11.99	9.18
2800	142.27	33.07	18.05	11.62	9.43
3200	185.83	32.34	17.48	10.94	8.75
3600	206.46	32.28	17.27	10.81	8.60
4000	246.45	33.13	17.75	11.11	8.86

Table 5.2: Running time comparison of Matlab *svds* and the parallel method with different numbers of cores.

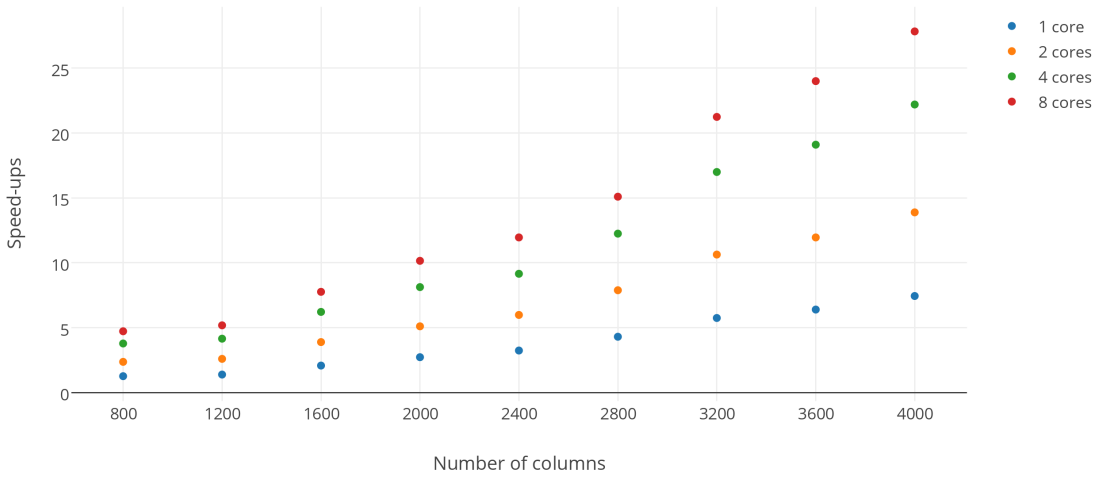

Figure 5.3: Graph showing the speed-ups on matrices with an increasing number of column vectors.

No. of columns	1 core	2 cores	4 cores	8 cores
800	1.26	2.37	3.78	4.72
1200	1.39	2.60	4.15	5.18
1600	2.08	3.89	6.21	7.76
2000	2.73	5.11	8.13	10.15
2400	3.24	5.98	9.15	11.95
2800	4.30	7.88	12.24	15.09
3200	5.75	10.63	16.99	21.24
3600	6.40	11.95	19.10	23.99
4000	7.44	13.88	22.19	27.82

Table 5.3: Speed-up factor by the parallel method based on the running time of Matlab *svds*.

5.4 μ_V-coherence

In Chapter 4, we presented an analysis that showed that the approximation of the leading right singular space is refined at each node of the execution tree. In this section, we will investigate the empirical quality of the approximation of the leading right singular space through numerical tests.

We introduce the concept of μ_V-coherence, which is inspired by the definition of matrix coherence. Matrix coherence plays an important role in several problems investigated in recent years, such as matrix completion [20] and low-rank approximation [92]. The standard mutual coherence of a matrix $A \in \mathbb{R}^{m\times n}$[32] is defined as,

$$\mu(A) = \max_{1\le i,j\le n} \frac{a_i^{\mathrm{T}} a_j}{\|a_i\|\|a_j\|}, \tag{5.5}$$

where a_i and a_j are the ith and jth columns of A, respectively. It is the maximum absolute value of the cross-correlations between the columns of A. The bigger $\mu(A)$ is, the more coherent A is.

For the purpose of measuring the approximation quality of the leading right singular space, we are interested in the difference between any Stiefel matrix W and the ground-truth matrix V_p. Inspired by the concept of matrix coherence, we define μ_V-coherence as follows.

Definition 5.1. *Given an $n \times p$ Stiefel matrix V, the μ_V-coherence of any Stiefel matrix $W \in \mathbb{R}^{n\times k}$ with respect to V is defined as*

$$\mu_V(W) = \max_{1\le i,j\le n} \frac{w_i^{\mathrm{T}} v_j}{\|w_i\|\|v_j\|}, \tag{5.6}$$

where w_i is the i-th columns of W and v_j are the jth columns of V.

Node	average of $\mu_{V_p}(W)$	variance of $\mu_{V_p}(W)$
(1,1)	0.5400	1.2589e-04
(1,2)	0.5398	1.2208e-04
(1,3)	0.5396	1.2419e-04
(1,4)	0.5399	1.2145e-04
(2,1)	0.7297	7.9078e-05
(2,2)	0.7395	7.9703e-05
(3,1)	1.0000	6.4164e-31

Table 5.4: The average μ_{V_p}-coherence of V_{out} at each node and their variance.

The definition μ_V-coherence describes how close the spaces range(W) and range(V) are. When W is orthogonal to V, $\mu_V(W) = 0$. When W generates the same subspace as V, $\mu_V(W) = 1$. If range(W) is a good approximation of range(V), $\mu_V(W)$ is close to 1. If range(W) is orthgonal to range(V), $\mu_V(W)$ is 0. We use μ_V-coherence of the outputs to observe how the quality of approximation of the leading right singular space is improved through the combinations.

In the following test, we looked at the coherence at the nodes in a 3-level binary execution tree. We tested 5000 randomly generated 1000×1000 matrices using the method in Section 5.2, with parameters $k = 10$, $\sigma_1 = 100$, $\alpha = 10$, $\beta = 0.7$ and $\eta = 0.8$. We computed the first 5 singular values and singular vectors.

The average μ_{V_p}-coherence of the output approximate leading right singular space at each level and their variance is shown in Table 5.4. Node (i, j) represents the j-th node on level i.

Figure 5.4 shows a graph with the histogram of the μ_{V_p}-coherence of output matrices at each of the three levels. The blue bars on the left of the graph show the histogram of the μ_{V_p}-coherence of the output matrices of the first level. The green bars that concentrate around 0.74 are the μ_{V_p}-coherence of the output matrices of the second level. The red bar shows that the μ_{V_p}-coherence of the output from the extraction node, i.e. the final results, are closely concentrated around 1.

The average of $\mu_{V_p}(W)$ keeps increasing from the top to the bottom of the execution tree. At the same time, the variance of $\mu_{V_p}(W)$ reduces to a very low level at the extraction node. This shows that the approximation of the leading right singular space improves in a very consistent manner through the levels of combination and computation.

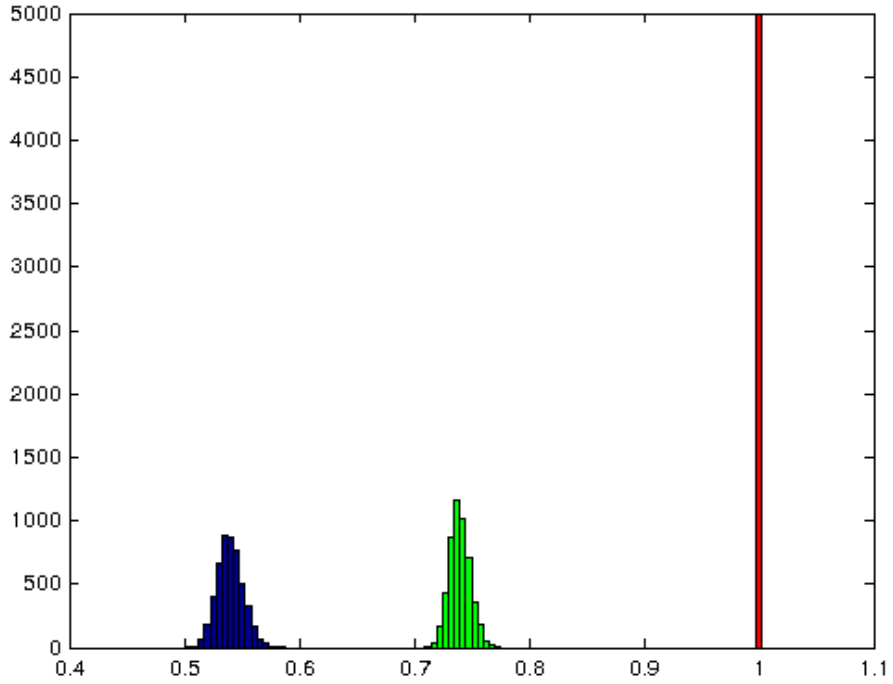

Figure 5.4: Histograms of the μ_{V_p}-coherence of output matrix V_{out} at each of the three levels

5.5 Tests on the loose coupling behaviours

In this section, we study the loose coupling behaviours of the algorithm by investigating how different combination tree topologies, node failure and communication costs affect its performance.

5.5.1 Tree topology

We first investigate the accuracy and the execution speed of the algorithm under two different combination tree topologies that represent opposite ends on the spectrum of possible orders, in which the information can be combined. The two topologies are shown in Figure 5.5 and 5.6. If the algorithm is able to ensure high accuracy under different tree topologies, then users can distribute computing tasks depending on the availability of their possibly heterogeneous machines and the next set of input data without worrying about effects on accuracy.

We study two extremes of tree topologies. In the first case, the execution tree follows binary combination as illustrated in Figure 5.5. This is the most balanced

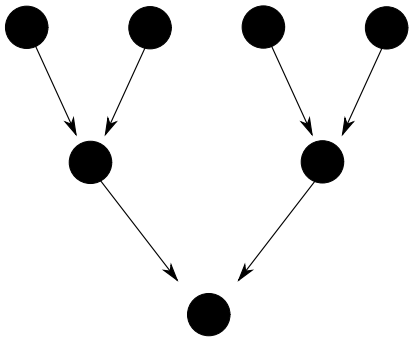

Figure 5.5: A binary execution tree.

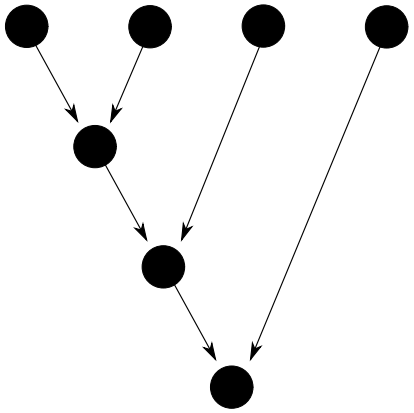

Figure 5.6: A comb execution tree.

order of data combination. In the second case, we follow a comb execution tree as shown in Figure 5.6. This is the least balanced order of data combination because the data from the leftmost seed node is used in every level computations, while the data from the rightmost seed node is used only once. In the binary tree, nodes on the same level do the same jobs only with different inputs. In the comb tree, the leftmost node in each level does all of the combinations of old outputs and new data. Other trees with the same number of seed nodes lie in-between these two cases. If these two type of trees can both guarantee the same level of accuracy, there is no need to bother choosing a specific tree topology for the purpose of accuracy.

We tested both topologies on the same set of randomly generated matrices of size 1000×4096. The parameters were set to $k = 10$, $p = 5$, $\sigma_1 = 100$, $\beta = 0.7$ $\eta = 0.2$. We carried out two sets of tests on $\alpha = 1.1$ and 5, respectively. The results shown in Tables 5.5 and 5.6 are averages of the μ_{V_p}-coherence obtained from 50 random samples, using the same number of seed nodes for both topologies. The first columns of both tables show the average μ_{V_p}-coherence of the outputs of seed nodes. The second columns show the results at nodes where the outputs from two seed nodes are

Number of seed nodes	1	2	4	8	16	32	64
Binary tree	0.1464	0.1975	0.2696	0.3709	0.5140	0.7164	0.9975
Comb tree	0.1478	0.1998	0.2706	0.3710	0.5148	0.7159	0.9976

Table 5.5: Comparison of average μ_{V_p}-coherence at levels corresponding to the specified number of seed nodes for the random matrix model with $\alpha = 1.1$.

Number of seed nodes	1	2	4	8	16	32	64
Binary tree	0.1472	0.1991	0.2717	0.3749	0.5196	0.7230	1.0000
Comb tree	0.1472	0.2004	0.2707	0.3754	0.5200	0.7232	1.0000

Table 5.6: Comparison of average μ_{V_p}-coherence at levels corresponding to the specified number of seed nodes for the random matrix model with $\alpha = 5$.

combined. For the binary combination tree, this is the average of the two combination nodes on the second level. For the comb tree, this is the leftmost node on level 2. We continue analogously to the point where 64 seed nodes are used, that is, the extraction nodes of both trees.

As indicated by Figure 5.7, the approximation quality of the outputs of the two trees are very close to each other when using the same amount of input data.

As is shown in these tests, the errors of different tree topologies are about the same in both cases. The sequential overall time used in the two trees are also similar: 122.70 seconds and 121.60 seconds respectively for $\alpha = 1.1$ and 113.80 seconds and 113.79 seconds respectively for $\alpha = 5$. With similar computing time, the two different trees achieve the same level of accuracy. This shows that the algorithm is well suited for heterogeneous computing environments.

5.5.2 Node failure

Unlike conventional supercomputers, modern computing architectures, like computer networks, have a higher probability to have node failure. Resistance to node failure is crucial for this type of computing. In the following tests, we investigated whether node failure affects the accuracy of the algorithm.

We randomly stopped some nodes in the execution tree and assumed that the local calculation failed in these nodes. The input data in these nodes were kept and reused in later nodes. If the extraction node failed, we restarted the node.

We tested on 1000×1024 random matrices with a 5-level binary execution tree. The parameters were set to $k = 10$, $p = 5$, $\sigma_1 = 100$, $\alpha = 5$, $\beta = 0.7$ and $\eta = 0.8$. We randomly forced parts of the nodes to fail with different probabilities as listed in

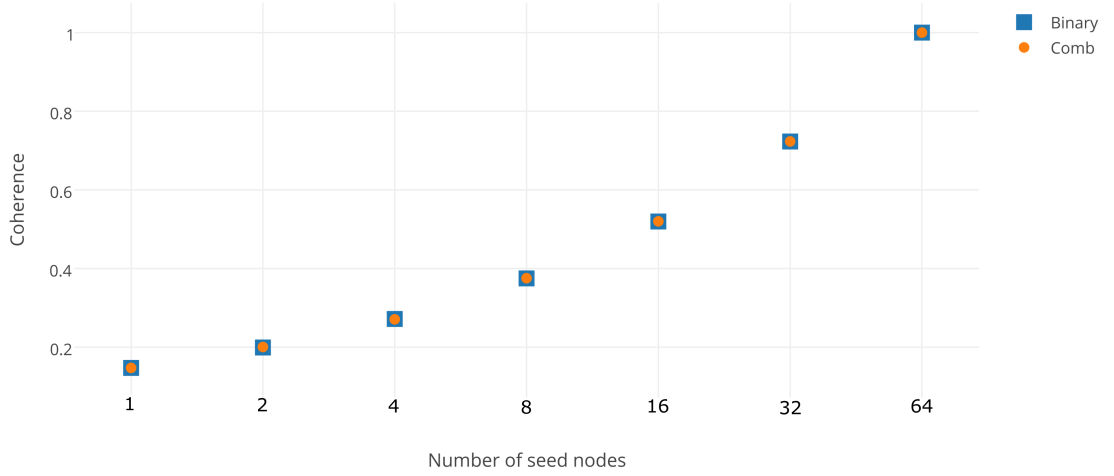

Figure 5.7: Comparison of μ_{V_p}-coherence of V_{out} of the nodes where the binary and comb trees use the same data ($\alpha = 5$).

Prob of failure	Uerr	Verr	$\sigma_1$err	$\sigma_5$err
0	6.3371e-05	4.5254e-03	1.5660e-15	1.3622e-04
0.1	5.8276e-05	4.1068e-03	1.7138e-15	1.1564e-04
0.2	7.3656e-05	4.2002e-03	1.4069e-15	1.2000e-04
0.5	1.0726e-04	3.5541e-03	1.5262e-15	8.6479e-05
0.8	8.7069e-05	4.2478e-03	1.2363e-15	1.1152e-04

Table 5.7: Errors of leading singular values and singular spaces when the nodes fail with different probabilities.

Table 5.7. We compared the errors of leading left and right singular spaces ('Uerr' and 'Verr'), i.e. the principal angles between the calucated spaces and the true ones, and the relative errors of the first and the fifth singular values ('$\sigma_1$err' and '$\sigma_5$err'). The results were the average values of 100 tests.

As illustrated in Table 5.7, the different probabilities of node failures did not affect the computing accuracy. As long as the data in the failed nodes was kept, the node failures had little impact on the results. This is in line with what we expect given our observations about the comb data flow in Section 5.5.1, as random node failure coupled with the re-use of data effectively corresponds to a random execution tree topology without node failure.

5.5.3 Communication costs

The avoidance of communication costs is an important motivation for distributed computing. In this section, we investigate the communication costs of a sequential implementation of the algorithm. The computations in nodes on a binary execution tree follows the computing order shown in Figure 5.8. We remark that for users with no access to proper parallel machines, they can still use the described algorithm by following this computing order, and there is a large speed-up to be gained in doing so, as our algorithm is a one-pass method and data can be read from the disk in chucks that are small enough to fit into CPU memory.

Under this execution order, we load a portion of columns of the original matrix into memory at points corresponding to seed nodes. At combination nodes and extraction nodes, we only transfer or store the outputs of a small number of other nodes. For example, if we compute the p leading SVD of a matrix of size $m \times n$ with a 4-level binary tree as shown in Figure 5.9, following the computing order of Figure 5.8, the data that needs to be stored at seed node $(1,4)$ is of size $\frac{mn}{s} + 2np$ where $s = 8$ is the total number of seed nodes in this instance. The term $\frac{mn}{s}$ is derives from the size of the inputt submatrix of the original matrix, and the $2np$ comes from the outputs of node $(2,1)$ and $(1,3)$. Similarly, the data to be held in memory of node $(1,8)$ is of size $\frac{mn}{s} + 3np$. The extra np comes from the outputs of node $(3,1)$ since we needed to keep it for the combination at node $(4,1)$. As a further example, in combination node $(2,2)$, $2np$ space was needed to store the outputs of node $(2,1)$, $(1,3)$ and $(1,4)$, while the space needed in node $(2,4)$ was $4np$. Generally speaking, the required memory size is $O(\frac{mn}{s})$ at a seed node and $O(np)$ at a combination node or extraction node.

We simulated the communications by transferring the data from memory to hard disk and loading them when needed. If the data-sets were small enough to be held in memory, we did not transfer them to the disk. For the tests we conducted, the only data transfer occurred at the seed nodes. The data needed in the combination nodes was small enough to be held in memory. We were interested in the proportion of time spent on data communication relative to the overall time. Algorithm 14 describes how the execution path is generated in a recursive way.

We fixed the number of rows to be 3000, the rank to 10 and the number of singular values computed to 5. Only the number of columns was changed, and the other parameters were chosen as $\sigma_1 = 100$, $\alpha = 5$, $\beta = 0.7$ and $\eta = 0.8$. We report the communication time ('com time'), the total time ('seq time') and the percentage of communication time out of total time ('percentage') in Table 5.8. Figure 5.10 illustrates the comparison between the communication time and the overall time.

Algorithm 14 A recursive implementation of the computing order in Figure 5.8

Input: $n_{level}+1$ is the number of levels
Call function Tree(n_{level},1)
function Tree(n_{level}, $n_{position}$)
  if ($n_{level}=1$)
    Seed node computations
  else
    Tree(n_{level}-1, $n_{position}\times$2-1)
    Tree(n_{level}-1, $n_{position}\times$2)
    Combination node computations
  end if

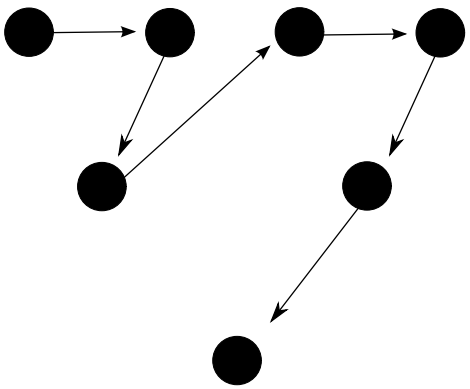

Figure 5.8: The sequential execution order in a binary tree.

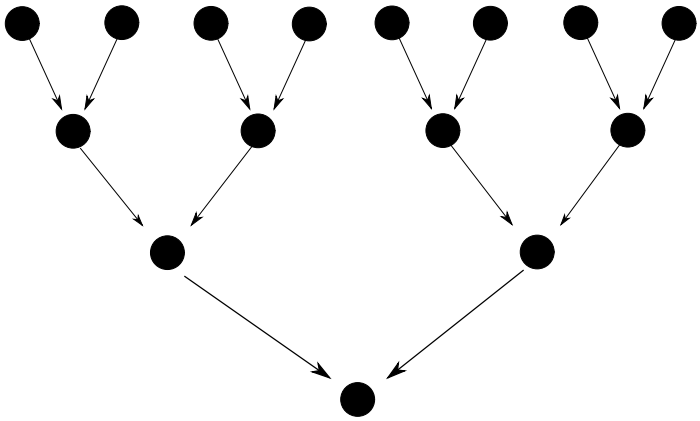

Figure 5.9: A 4-level binary tree.

5.6 Two-layer parallelisation

When the sizes of the matrices are extremely large compared to the computing capacity and memory size of the computer, the parallelisation can be carried out with both column-splitting and row-splitting. In each node of the standard implementation of our algorithm, we must compute the p-leading SVD of a skinny submatrix of

n	level	seq time	com time	percentage	Uerr	Verr	$\sigma_1$err
5000	4	6.54	0.89	13.65%	7.7468e-6	1.3324e-3	2.6716e-16
10000	4	13.50	1.77	13.10%	5.4587e-6	1.0911e-3	2.0464e-16
20000	4	26.98	3.63	13.44%	1.6289e-7	6.6463e-4	7.1054e-16
40000	5	54.19	7.28	13.43%	1.4745e-6	6.7151e-4	5.7696e-16
100000	5	152.79	27.07	17.71%	5.0952e-7	4.1662e-4	7.6170e-16

Table 5.8: Communication time and its percentage in total running time for matrices of different numbers of columns

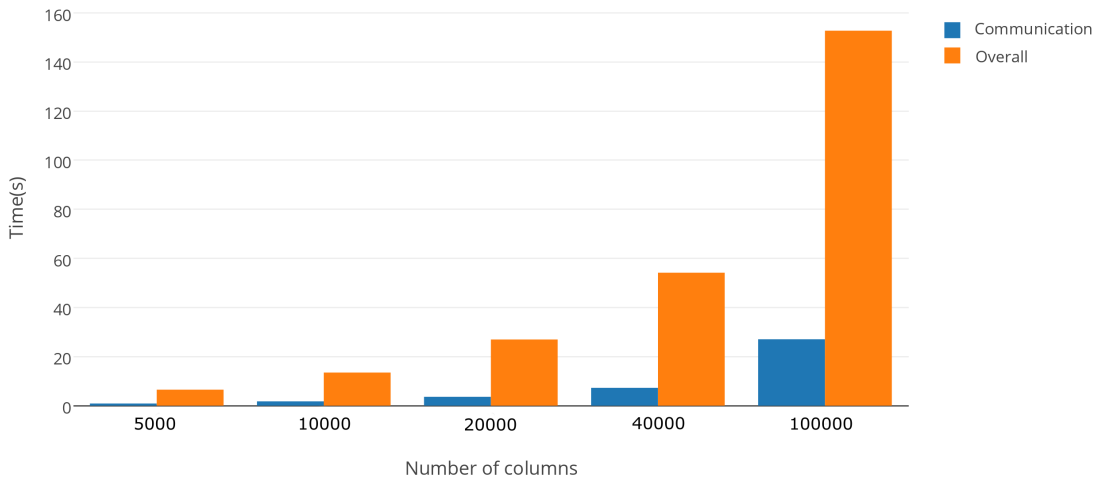

Figure 5.10: Comparison between communication time and the overall time. The algorithm was carried out with the sequential order illustrated in Figure 5.8.

A. This SVD computation could be done with any existing SVD method. To tackle the SVDs of huge matrices, we can therefore use the proposed parallel SVD algorithm to compute the leading SVD of such a skinny submatrix. Instead of column-splitting, we use row-splitting on the skinny matrix, or column splitting of its transpose, so that we can get almost square submatrices after the column and row splitting.

Figures 5.11, 5.12 and 5.13 show an example of such process. The matrix is firstly split into four pieces by column partitions. On a recursive second layer, one of the pieces is further split into four blocks by row-splitting. Local computations are conducted on these blocks that are small enough to compute their leading SVD directly. Each of the seed nodes accesses one of the submatrices, $A_{i,j}$, and does the local computations on $A_{i,j}^{\mathrm{T}}$. When combining the local results, we first used the parallel method to return the leading SVD of the first group of submatrices. For instance, when we receive the results of $A_{1,1}^{\mathrm{T}}$, $A_{2,1}^{\mathrm{T}}$, $A_{3,1}^{\mathrm{T}}$ and $A_{4,1}^{\mathrm{T}}$ in seed nodes, combination nodes combine them until an approximate leading part SVD can be extracted. This procedure is used in local computations for the seed nodes on the first level, i.e. for the pieces which after column-splitting contain $A_{1,1}$, $A_{2,1}$, $A_{3,1}$ and $A_{4,1}$. For the remaining layers, the data could be held in memory, and the second recursive partitioning is not necessary.

The two-layer parallelisation greatly increases the capacity of the algorithm in dealing with extremely large problems. We tested the method on randomly generated matrices with gradually increasing dimensions. The rank of the matrices was fixed to be 10 and we computed their 5-leading SVD. The parameters were chosen as $\sigma_1 = 100$, $\alpha = 10$, $\beta = 0.7$ and $\eta = 0.8$. The numbers of seed nodes were changed with the size of the matrices to reduce the matrices in local computations of moderate size, i.e. around 1000 by 1000. The results are listed in Table 5.9. The largest matrices we tested are 102400 by 102400 which finished in a few hours sequentially, including the loading time of the whole matrix.

These results show that the algorithm significantly improves the power of a single machine used by standard users in SVD computations. The desktop used in the tests can only solve the leading part SVD of matrices of size thousands by thousands using the Matlab function *svds*. By using the proposed method with two-layer parallelisation, the computing power of the desktop was increased to hundred thousands by hundred thousands, with the job done in only a few hours.

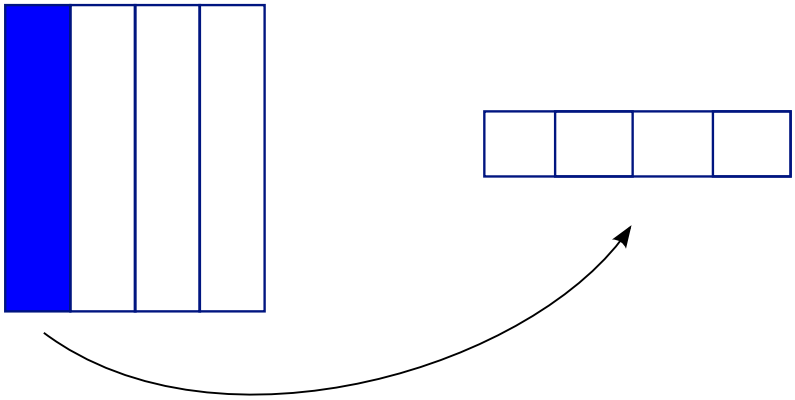

Figure 5.11: An example of parallelising the computations in two layers.

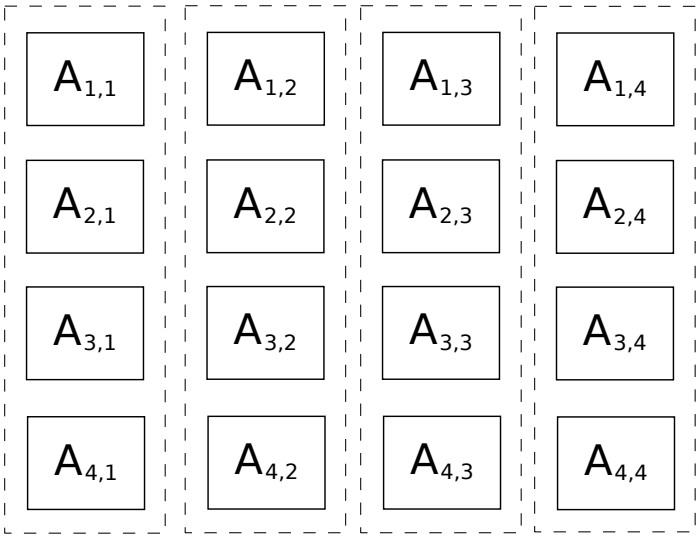

Figure 5.12: Column-splitting and row-splitting.

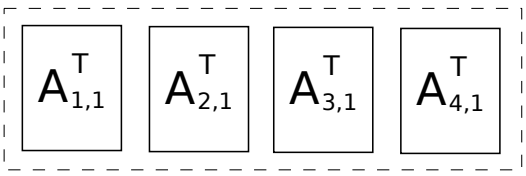

Figure 5.13: An example of row-splitting of local computations of seed nodes in the first level of parallelisation.

5.7 Multi-sweep

As discussed in Section 5.1, more than one sweep of the method can be used to improve the accuracy of the results. In the following experiment, we tested the multi-sweep version of the algorithm on randomly generated test matrices of size 1000×1024 with different α. The other parameters were kept constant at $k = 10$, $p = 5$, $\sigma_1 = 100$,

m, n	Outer level	Inner level	seq time	par time	Uerr	Verr	$\sigma_1$err
10000	4	4	47.05	6.16	9.90e-5	1.62e-4	4.41e-15
20000	4	4	229.47	30.14	2.24e-4	2.40e-4	1.56e-15
40000	5	5	849.17	59.52	1.09e-5	7.92e-5	4.26e-16
102400	7	7	5236.02	111.49	2.58e-4	2.66e-4	1.12e-14

Table 5.9: Numerical results of two layer parallelisation

		1 sweep	2 sweeps	3 sweeps
$\alpha = 5$	Uerr	6.6648e-06	1.0898e-14	1.0205e-14
	Verr	1.8974e-03	1.05326e-14	8.3298e-15
	$\sigma_1$err	4.2633e-16	9.9476e-16	4.2633e-16
	$\sigma_5$err	1.4111e-05	1.5266e-14	1.8041e-14
$\alpha = 1.1$	Uerr	4.5382e-02	3.4606e-14	1.5491e-13
	Verr	5.2667e-01	4.7428e-14	1.9826e-13
	$\sigma_1$err	4.8728e-03	1.1369e-15	7.1054e-16
	$\sigma_5$err	9.1338e+02	1.9327e-10	1.4396e-09
$\alpha = 1.01$	Uerr	1.5572e+00	1.9008e-14	1.6390e-14
	Verr	1.5625e+00	1.9053e-14	1.6396e-14
	$\sigma_1$err	8.5288e-03	3.9790e-15	1.4211e-16
	$\sigma_5$err	2.6479e+03	7.1054e-12	1.1369e-11

Table 5.10: Errors of leading singular values and singular spaces after multiple sweeps.

$\beta = 0.7$, $\eta = 0.8$. The errors of leading singular spaces and the relative errors of singular values in the first three sweeps are listed in Table 5.10.

The errors of both leading singular spaces and singular values reduce to a near machine accuracy after two sweeps. Thereafter, the errors are kept on the same order of magnitude. We tested up to 20 sweeps and the errors become stable after getting to a high accuracy in the second sweep. This suggests that two sweeps are usually enough.

5.8 Tests on parallel machine

In this section, we present the numerical results using a parallel machine, the COSMOS2 cluster [2] at the University of Cambridge. The Python code is attached in Appendices B and C.

5.8.1 Experiment settings

We have access to a maximum of 512 cores of the COSMOS2 cluster. COSMOS2 has 12.2TB of shared memory, but jobs are restricted to 7.3 GB per core.

We generated test matrices $A \in \mathbb{R}^{n\times n}$ by generating the singular vectors and values separately. The rank of the matrices was 10, and we used a parallel implementation of our leading SVD method to compute the first 5 singular values and the corresponding singular vectors. The decrease factor of the singular values was chosen to be 10. We further generated $n \times 10$ Stiefel matrices U and V through the QR factorisation of $n \times 10$ random matrices with $N(0, 1)$ i.i.d. coefficients.

The binary tree topology of Chapter 3 was used for the map reduce action. Let us denote the number of processor by $s = 2^l$ and the number of levels in the binary tree by $l + 1$. The test matrix A was equally split into s sub-matrices, $A_1, \ldots, A_s$, by column. Each sub-matrix A_i, $i = 1, \ldots, s$, was sent to a node i. We denote the time to transfer A_i by $t_{0,i}$. The seed node computations were carried out at nodes $1, \ldots, s$. The i-th node requiring computing time $T_{1,i}$. For even indices i, the outputs of node i are then transferred to node $i - 1$, where they are combined with the outputs in node $i - 1$. The time to transfer data from node i to $i - 1$ at level k is denoted by $t_{k,\frac{i}{2}}$, where $k = 1, \ldots, l$ and $i = 1, \ldots, \frac{s}{2^{(k-1)}}$. The computing time of node i at level k is denoted by $T_{k,i}$.

5.8.2 Experiment results

We tested our parallel code on matrices of size 4096×4096 to 262144×262144, using 32, 128 and 512 cores. We report the average results over 50 tests. Tables 5.11, 5.12 and 5.13 report the average running times and errors of the experiments.

We calculated the average running time of local computations at each level. The results are presented in Table 5.11. For node i at level k, the running time is $T_{k,i}$. The average of $T_{k,i}$, for $i = 1, \ldots, \frac{s}{2^{(k-1)}}$, is denoted by $\bar{T}_k$. Here we report the average of $\bar{T}_k$. The most costly computation was carried out on nodes at level 1, where the SVD of an $m \times \frac{n}{s}$ matrix was calculated. After the first level, the computing time was smaller by several orders of magnitude, due to the fixed size $n \times 2p$ of the data matrices for which a leading part SVD is computed. The reduction in complexity is particularly dramatic when n is large and the number of seed nodes is only moderately large. In these cases almost all of the computational cost arises at level 1, and subsequent levels are practically free. When the matrix size is 4096 and 8192, the difference was less than 100 times, while when the matrix size was more than 100000 by 100000, the

difference was more than 1000 times. We can expect that when an even larger input matrix is given, this effect will be even larger. Using more cores helps to reduce the parallel computing time. Let us take the experiments with matrices of size 65536 by 65536 as an example: When 32 cores were used, the ratio of the first level computing time to the second level computing time was more than 10^6 times. When 512 cores were used, this ratio reduced to about 4000. When using 512 cores, the parallel computing time at level 1 reduced to 4.0% of the time used for computations at level 1 when using 32 cores, with only a very small additional computing time due to more levels were used. Figure 5.14 is the Plot of average running time of local computation at level 1 versus number of cores used. If a user has more nodes for larger matrices, we expect the improvement to be even more significant.

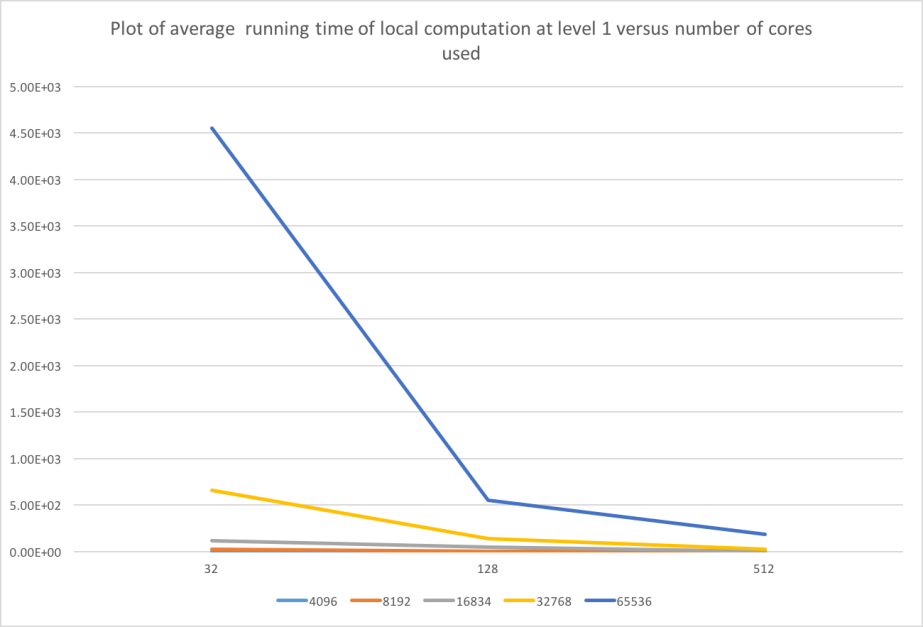

Figure 5.14: Plot of average running time of local computation at level 1 versus number of cores used.

Tables 5.12 shows the average time to transfer data between nodes. We count the time from the node starting to send the data to the time the data is transmitted. In column 'Level k', we present the time used to transfer data from outputs form level $k-1$ to the corresponding cores used for level k computations. For node i at level k, the communication time is $t_{k,i}$. The average of $t_{k,i}$, for $i = 1, \ldots, \frac{s}{2^{(k-1)}}$, is denoted by $\bar{t}_k$. Here we report the average of $\bar{t}_k$ over 50 tests. The communication time at level 1 is larger as it transfers matrices of larger size. This is effectively the set up cost for seed nodes, which is irrelevant in practical applications where the data-sets are

already in existence and distributed over multiple computers, rather than having to be generated, split, and distributed. When 128 cores were used, the communication time at the seed node level, i.e. the set up time, is consistently less than the same test on 32 cores, while parallel communication times from the levels 2 onwards are independent of the number of nodes used, due to the fixed size of the communicated data. This is more clear when the matrix size is larger, as the communication overheads take smaller percentage of the communication time. Again, let us take 65536 by 65536 as an example. The parallel communication time is reduced to 23.5 % with only a very small communication time is added due to more levels were used, when comparing using 128 cores to using 32 cores. We note that the communication time when 512 cores are used, is larger, while in theory, this time should be smaller since smaller matrices are being transferred. We think this is caused by the particular network topology of the cluster we have used. Our results up to 128 cores show that the communication time comes down with more cores and we expect this to be true for even higher number of cores.

Table 5.13 presents the errors of the singular values. Column 'σ_i err' is the relative error of singular value σ_i. The error increases when more seed nodes, and hence a larger number of levels, are used. We note that for the first and second singular values, the increase is very slow. For the smaller singular values, the increase is faster and the errors are larger, though the error growth is much smaller than the theoretical bound of the analysis we presented in Chapter 4. For applications that requires very high accuracy, the remedy to this problem is to compute a couple of more singular values as a trade-off.

We note that the maximal size of the matrix that we can test is limited by the memory accessible to each node. Larger matrices can be handled in a reasonable time if more memory is available.

Table 5.11: Comparison of average running times of local computations at nodes in different levels.

n	No. of levels	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
4096	6	4.48E+00	2.59E-03	1.87E-03	1.89E-03	2.01E-03	2.80E-03				
4096	8	2.58E-01	2.88E-03	2.03E-03	1.97E-03	2.00E-03	2.51E-03	2.66E-03	4.09E-03		
4096	10	5.32E-02	2.13E-03	1.85E-03	1.83E-03	1.83E-03	1.84E-03	1.87E-03	1.92E-03	2.03E-03	2.80E-03
8192	6	2.42E+01	4.66E-03	3.74E-03	3.66E-03	3.87E-03	5.45E-03				
8192	8	2.79E+00	4.68E-03	3.66E-03	3.50E-03	3.52E-03	3.63E-03	3.88E-03	5.48E-03		
8192	10	2.28E-01	5.31E-03	3.83E-03	3.57E-03	3.55E-03	3.66E-03	3.91E-03	4.10E-03	5.04E-03	9.64E-03
16834	6	1.16E+02	8.90E-03	8.27E-03	7.49E-03	7.78E-03	1.07E-02				
16834	8	4.74E+01	9.52E-03	8.38E-03	7.77E-03	7.54E-03	8.06E-03	8.91E-03	1.52E-02		
16834	10	1.95E+00	9.54E-03	8.47E-03	7.28E-03	7.14E-03	7.26E-03	7.68E-03	8.33E-03	9.40E-03	1.84E-02
32768	6	6.59E+02	1.80E-02	1.65E-02	1.62E-02	1.67E-02	2.35E-02				
32768	8	1.36E+02	1.80E-02	1.63E-02	1.55E-02	1.47E-02	1.53E-02	1.67E-02	2.36E-02		
32768	10	2.44E+01	1.91E-02	1.68E-02	1.60E-02	1.47E-02	1.53E-02	1.63E-02	1.90E-02	2.47E-02	4.05E-02
65536	6	4.56E+03	4.04E-02	3.33E-02	3.28E-02	3.58E-02	5.08E-02				
65536	8	5.52E+02	3.99E-02	3.24E-02	3.11E-02	3.16E-02	3.27E-02	3.56E-02	5.01E-02		
65536	10	1.81E+02	4.23E-02	3.39E-02	3.21E-02	3.18E-02	3.30E-02	3.57E-02	4.10E-02	5.27E-02	9.79E-02
131072	8	2.57E+03	1.63E-01	8.48E-02	7.24E-02	7.59E-02	8.57E-02	1.20E-01	2.18E-01		
131072	10	5.04E+02	1.30E-01	7.24E-02	5.75E-02	5.48E-02	5.46E-02	6.62E-02	9.46E-02	1.64E-01	3.53E-01
262144	10	5.87E+02	7.24E-01	3.22E-01	2.47E-01	2.23E-01	2.45E-01	3.04E-01	3.87E-01	6.62E-01	1.40E+00

Table 5.12: Comparison of average communication times of local computations at nodes in different levels.

n	No. of levels	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10
4096	6	5.03E-02	2.77E-04	5.46E-04	5.76E-04	8.90E-04	1.32E-03				
4096	8	8.57E-02	5.24E-04	5.76E-04	4.51E-04	6.67E-04	5.62E-04	3.52E-04	1.80E-04		
4096	10	1.04E-02	3.52E-03	3.39E-02	1.22E-03	2.03E-03	6.62E-04	4.56E-04	6.86E-04	1.67E-04	1.60E-04
8192	6	2.03E-01	4.11E-04	5.64E-04	5.72E-04	9.73E-04	1.19E-03				
8192	8	6.33E-02	5.77E-04	1.06E-03	4.84E-04	3.98E-04	2.67E-04	2.40E-04	2.69E-04		
8192	10	8.96E-02	1.84E-02	2.14E-02	8.87E-03	2.75E-03	3.27E-03	7.19E-04	4.02E-04	2.51E-04	4.26E-03
16834	6	8.76E-01	1.76E-02	8.33E-04	8.90E-04	1.10E-03	4.51E-04				
16834	8	7.70E-01	8.40E-03	3.46E-03	2.84E-03	2.15E-03	2.83E-03	3.47E-03	7.21E-03		
16834	10	1.60E-01	1.98E-02	2.31E-02	1.24E-02	2.71E-03	1.66E-03	1.54E-03	1.89E-03	3.35E-03	5.43E-03
32768	6	3.37E+00	2.73E-01	1.44E-03	1.64E-03	2.29E-03	2.93E-03				
32768	8	9.51E-01	4.44E-02	2.19E-03	1.58E-03	1.37E-03	1.23E-03	1.26E-03	1.32E-03		
32768	10	5.59E-01	2.29E-02	1.84E-02	8.96E-03	3.63E-03	3.09E-03	3.73E-03	5.25E-03	1.02E-02	2.50E-02
65536	6	1.53E+01	1.38E+00	2.57E-03	2.47E-03	3.21E-03	4.07E-03				
65536	8	3.59E+00	1.16E-01	3.00E-03	2.22E-03	2.64E-03	2.82E-03	3.43E-03	2.75E-03		
65536	10	2.26E+00	3.78E-02	9.76E-03	6.81E-03	4.60E-03	4.99E-03	7.13E-03	1.13E-02	2.21E-02	4.61E-02
131072	8	2.95E+01	1.80E+00	1.61E-02	1.33E-02	1.13E-02	1.56E-02	4.65E-02	1.11E-01		
131072	10	1.41E+01	6.23E-01	2.18E-02	1.14E-02	1.43E-02	1.31E-02	2.43E-02	4.80E-02	1.07E-01	2.50E-01
262144	10	1.66E+01	4.69E+00	1.93E-01	9.13E-02	9.38E-02	7.63E-02	1.34E-01	1.44E-01	4.88E-01	1.06E+00

Table 5.13: Relative numerical error in singular values as a function of matrix size and number of processors used.

n	No. of levels	σ_1 err	σ_2 err	σ_3 err	σ_4 err	σ_5 err
4096	6	6.11E-16	7.60E-16	4.25E-15	3.70E-11	3.76E-07
4096	8	6.65E-16	1.06E-15	1.65E-14	1.58E-10	1.61E-06
4096	10	7.05E-16	7.82E-16	1.06E-13	1.07E-09	1.04E-05
8192	6	7.45E-16	1.02E-15	2.26E-15	1.80E-11	1.86E-07
8192	8	7.90E-16	9.56E-16	8.19E-15	7.65E-11	7.66E-07
8192	10	8.36E-16	9.73E-16	3.47E-14	3.47E-10	3.49E-06
16834	6	1.22E-15	1.18E-15	1.61E-15	8.96E-12	9.64E-08
16834	8	1.24E-15	1.36E-15	4.80E-15	3.72E-11	3.94E-07
16834	10	1.11E-15	1.33E-15	1.64E-14	1.59E-10	1.62E-06
32768	6	1.35E-15	1.62E-15	1.53E-15	4.91E-12	4.44E-08
32768	8	1.57E-15	1.67E-15	2.48E-15	1.92E-11	1.91E-07
32768	10	1.60E-15	1.58E-15	8.48E-15	7.83E-11	7.91E-07
65536	6	4.55E-16	3.13E-16	7.99E-16	5.28E-13	5.28E-09
65536	8	4.09E-16	4.19E-16	1.06E-15	2.08E-12	2.16E-08
65536	10	4.09E-16	3.80E-16	4.19E-15	8.36E-12	8.73E-08
131072	8	6.37E-16	4.44E-16	9.50E-16	1.08E-12	1.04E-08
131072	10	6.65E-16	5.04E-16	1.71E-15	4.17E-12	4.32E-08
262144	10	2.66E-15	3.65E-15	3.18E-15	9.73E-12	9.89E-08

Chapter 6

Numerical Experiments: Sparsity-inducing Optimization

The leading part SVD is ubiquitous in scientific and engineering problems, for example, the leading part SVD of large matrices plays an important role as a dimension-reduction tool in big data applications. We are interested in the accuracy of our parallel SVD algorithm in such applications, as they provide a rich source of *non-random* input data, and the numerical experiments we generated so far were all based on random matrices.

One of the important areas that often uses leading part SVD are matrix optimisation problems. SVDs may be used in every iteration of algorithms for solving such problems, dominating the computational cost. Our parallel SVD algorithm may help improve the speed and scale of these methods. In this chapter, we chose low-rank matrix completion and sparse principal component analysis as examples to study the accuracy of the algorithm on matrices that arise in practical problems.

6.1 Sparsity inducing optimisation problems

The problem of learning a function from a small set of examples is central to machine learning and data mining. A typical technique is to cast the problem as an optimisation problem which minimizes an objective function combining of two terms over a large set of parameters that are intrinsic to the learning problem. One term is an error term which measures the fit to the data; the other one is a regularization term which favours certain parameter configurations. Here in particular, we are interested in the problems where the parameters are organized into a large matrix. Such problems that can be cast as an optimization problem of the form,

$$\min\{\varepsilon(W, y) + \lambda\Omega(W) | W \in \mathcal{W}\}, \tag{6.1}$$

where W is an $n \times d$ parameter matrix, $\mathcal{W}$ is a set of such matrices and y is an m-dimensional real vector of observations. $\varepsilon(W, y)$ is the error term and $\Omega(W)$ is the regularization term. The positive constant λ controls the trade-off between the two terms. A typical choice of $\varepsilon(W, y)$ is $\varepsilon(W, y) = \|I(W) - y\|_2^2$ where $I : \mathbb{R}^{n \times d} \to \mathbb{R}^m$ is a linear operator. Here we assume that $y = I(W^*) + \xi$ where W^* is the matrix we wish to learn and ξ is a zero mean random noise vector. The 2-norm $\|\cdot\|_2$ could be replaced by any convex loss function.

In sparsity inducing application methods, we focus on the case where Ω favours solutions with sparsity properties. In this context, sparsity means that many of the entries of the parameter matrix are zero, or that many of the components of a decomposition of the matrix are zero. For example, the matrix may have many zero rows or columns, many zero eigenvalues or singular values, sparse eigenvectors or singular vectors, banded structure, etc.. The solutions are expected to be low-rank matrices in many applications. Since the rank function is combinatorial in nature, the problems will be intractable if we choose Ω as $\text{rank}(W)$. In analogy to the cardinality function $\|\cdot\|_0$ of vectors, it is customary to replace $\text{rank}(\cdot)$ by the *nuclear norm*:

$$\|X\|_* := \sum_{i=1}^{r} \sigma_i(X), \tag{6.2}$$

where $\sigma_1(X) \geq \sigma_2(X) \geq \ldots \geq \sigma_r(X) \geq 0$ are the singular values of X. The nuclear norm is the best convex approximation of the rank function over the 2-norm unit ball of matrices [37].

The idea of sparsity inducing models first came to be popular because of the development of compressed sensing. Compressed sensing focuses on the problem of reconstruction of under-sampled signals. The Nyquist-Shannon sampling theorem gives a sufficient condition for reconstruction. However, given an additional assumption of sparsity, this condition can be relaxed. A typical result of the theory of compressed sensing is that a signal x of length N with T non-zero components can be reconstructed exactly in most cases if the number of measurements K is at least $T \log N$ [21]. When N is large and the signal is sufficiently sparse, this may imply that the signal can still be reconstructed perfectly even if under-sampled significantly.

The success of compressed sensing encouraged research into other sparsity notions and associated sparse optimisation models. Sparsity inducing optimisation models occur in many areas of machine learning, ranging from compressed sensing, sparse estimation, matrix completion to kernel learning. The motivations that drive the

pursuit of sparse solutions are manifold. Firstly, when the number of model parameters is much larger than the number of observations, a sparse parameter matrix is strongly desirable for fast and accurate learning. Secondly, because the parameters have their own physical meaning in practical problems, as Occam's Razor says, simpler explanations are generally better than more complex ones. It is often easier to conform a simpler solution to prior knowledge. Thirdly, sparse solutions are more robust to noisy and inaccurate data because they only extract salient information from the data.

The interplay between machine learning and optimization not only leads to more efficient algorithms, but also to the discovery of new learning models and custom-designed algorithms. In the following sections, we will detail two examples of sparse matrix learning problems, specifically, low-rank matrix completion and sparse PCA. Both problems are amenable to being solved by algorithms that rely on computing leading part SVDs in inner loops. They thus can be an ideal test bed for applying our SVD algorithm to a set of non-randomly generated matrices.

6.2 Low-rank matrix completion

The theme of compressed sensing is to recover a signal by solving underdetermined linear systems. A signal vector of length N that is k-sparse (i.e., has at most k nonzero entries), can be reconstructed from less than N linear measurements. An extension of these ideas is the low-rank matrix completion problem. Rather than recovering a sparse vector, we wish to recover a low-rank matrix $X \in \mathbb{R}^{m\times n}$.

The low-rank matrix completion problem can be formulated as follows,

$$\begin{aligned} \min \quad & \operatorname{rank}(X) \\ \text{s.t.} \quad & X_{ij} = M_{ij}, \quad (i,j) \in \Gamma, \end{aligned} \tag{6.3}$$

where $X \in \mathbb{R}^{m\times n}$ is the matrix which we want to complete. Γ is a subset of index pairs (i,j) and the (i,j)-th entries of M, i.e. $M_{ij}, (i,j) \in \Gamma$ is a given sample of X.

A famous instance of matrix completion is the Netflix problem. The data matrix M consists of ratings of movies by users, the (i,j)-th entry corresponding to the rating of movie j by user i. Usually, users only rate very few movies so we only know a small portion of the entries of M. We wish to complete M so that we can recommend movies that a particular user might also like. The data matrix is assumed to be low-rank because people believe that only a few factors contribute to an individual's tastes and preferences. One of the difficulties in this problem is the large data size. Another

example arises in image completion. Sometimes, many pixels are missing in an image because of occlusion. Recovery of the image is a matrix completion problem where the underlying scene is inherently low-rank.

The matrix completion problem is generally NP-hard because of the combinatorial nature of rank(X). However, the results of Candes-Recht [20] and Candes-Tao [22] show that, under some reasonable conditions, a solution of (6.3) can be found by solving the convex approximation:

$$\begin{aligned} \min \quad & \|X\|_* \\ \text{s.t.} \quad & X_{ij} = M_{ij}, \quad (i,j) \in \Gamma. \end{aligned} \tag{6.4}$$

Based on these results, various algorithms have been proposed, see [71], [93] and [13]. Because of the use of nuclear norm, many of them bear the computational cost required by SVD. Efficient SVD algorithms are important to improve the speed and scaling of such methods. We will take a FPCA scheme proposed by Ma, Goldfarb and Chen [71] as an example. The FPCA scheme is detailed in Algorithm 15, where $A(X)$ is the linear mapping from $\mathbb{R}^{m\times p}$ to $\mathbb{R}^{m\times n}$ and $s_{\tau\mu}$ is shrinkage operator:

$$s_{\tau\mu} = \bar{x} \quad \text{with} \quad \bar{x_i} = \begin{cases} x_i - \tau\mu, = & \text{if } x_i - \tau\mu > 0 \\ 0, & \text{o.w.} \end{cases} \tag{6.5}$$

Algorithm 15 FPCA scheme

Initialize: Given X_0 and $\bar{\mu} > 0$, select $\mu_1 > \mu_2 > \ldots > \mu_L = \bar{\mu} > 0$ and let $X = X_0$.
For $\mu = \mu_1, \mu_2, \ldots, \mu_L$ do
  While not converged, do
    1. select $\tau > 0$
    2. compute $Y = X - \tau A^*(A(X) - b)$, and SVD $Y = U\text{Diag}(\sigma)V^T$
    3. compute $Y = U\text{Diag}(s_{\tau\mu}(\sigma))V^T$
  End while
End for

The leading part SVD computations of large matrices Y in the inner loops dominate the computational costs of this method. A fast Monte Carlo method by Drineas et al. [34] was used in the literature to improve the performance of this scheme. In the following section, we will apply our parallel SVD algorithm in the context of Algorithm 15 and compare it with the case where the fast Monte Carlo method was used.

method	iter	rel err
FPC	33	2.33e-07
FPCA	60	2.49e-07
FPCP	33	2.33e-07

Table 6.1: Comparison of FPCA with Matlab *svds* (FPC), the fast Monte Carlo method (FPCA) and our parallel SVD (FPCP) (m=1000, n=1000, r=5, sr=0.5, freedom=0.02, mu=1.0000e-04, xtol=1.0000e-04, maxinneriter=10, tau=2).

method	iter	rel err
FPC	31	8.00e-08
FPCA	60	9.38e-08
FPCP	31	8.00e-08

Table 6.2: Comparison of FPCA with Matlab *svds* (FPC), the fast Monte Carlo method (FPCA) and our parallel SVD (FPCP). (m=3000, n=3000, r=10, sr=0.5, freedom=0.01, mu=1.0000e-04, xtol=1.0000e-04, maxinneriter=10, tau=2).

6.2.1 Numerical tests

In the first test, we started from matrices of moderate size and compared the performance of the FPCA approach with Matlab built-in function *svds* (FPC), the fast Monte Carlo method by Drineas et al. (FPCA) and the parallel SVD (FPCP). Two sets of tests on matrices size of 1000×1000 and 3000×3000 are reported in Table 6.1 and 6.2 respectively. 'iter' is the number of iterations used in the FPCA scheme and 'rel err' is the relative error. We used the binary combination verstion of the parallel algorithm with 4 levels.

As is shown in Table 6.1 and 6.2, all of the three approaches achieved the same level of accuracy. The FPC and FPCP approaches use the same number of iterations, while the FPCA approach uses nearly twice of number of iterations to achieve the same level of accuracy. Although the fast Monte Carlo method is fast in SVD computing, the larger number of iterations makes the FPCA approach under-perform the other two approaches.

When the matrices are too large to be held in memory, we can integrate the parallel SVD with the FPCA scheme. In step 2 of Algorithm 15, a full matrix Y does not need to be formed when initialising the seed nodes in the SVD computation, only a block of the columns is needed. In step 3, the new Y can be updated block by block. Hence, we do not need to form a full matrix throughout the computing. We only hold a subset of the columns of Y in CPU memory at any given time. The

m	n	iter	seq time	par time	err
1024	1024	35	69.66	24.63	4.43e-4
1024	2048	35	136.91	49.24	3.66e-4
1024	4096	34	277.48	87.92	3.76e-4
1024	8192	34	528.98	150.72	3.77e-4
1024	16384	34	1047.55	280.32	3.94e-4
1024	32768	35	2125.56	562.17	3.56e-4

Table 6.3: Recovery of matrices of different dimensions by FPCP. The number of levels of the execution trees is 3.

m	n	iter	seq time	par time	err
1024	1024	35	87.07	18.04	4.33e-4
1024	2048	35	162.08	30.05	3.62e-4
1024	4096	34	332.69	56.21	3.69e-4
1024	8192	34	648.86	99.18	3.69e-4
1024	16384	34	1297.87	186.41	3.90e-4
1024	32768	35	3075.27	448.05	3.58e-4

Table 6.4: Recovery of matrices of different dimensions by FPCP. The number of levels of the execution trees is 4.

blocks are saved after each update and loaded when needed. The power of a single computer is greatly increased in this way. We tested problems of sizes from 1024×1024 to 1024×32768. Numerical results are presented in Table 6.3 - 6.6, including the running times ('seq time') and the simulated parallel time ('par time'). Figure 6.1 shows the plot of running times for solving the same problem with different numbers of levels of parallelisation.

To illustrate the results of FPCP graphically, we show some images recovered by FPCP. We tested on both full rank and low rank matrices with random masks and

m	n	iter	seq time	par time	err
1024	1024	35	95.74	13.68	4.60e-4
1024	2048	35	170.04	22.04	3.65e-4
1024	4096	34	319.07	35.15	3.68e-4
1024	8192	34	635.05	59.81	3.75e-4
1024	16384	34	1312.20	110.56	3.91e-4
1024	32768	35	3209.12	285.76	3.58e-4

Table 6.5: Recovery of matrices of different dimensions by FPCP. The number of levels of the execution trees is 5.

m	n	iter	seq time	par time	err
1024	1024	35	74.24	11.74	4.56e-4
1024	2048	35	125.11	17.72	3.64e-4
1024	4096	34	255.71	28.89	3.73e-4
1024	8192	34	505.05	50.15	3.79e-4
1024	16384	34	1064.62	87.90	3.88e-4
1024	32768	35	2200.30	163.31	3.59e-4

Table 6.6: Recovery of matrices of different dimensions by FPCP. The number of levels of the execution trees is 6.

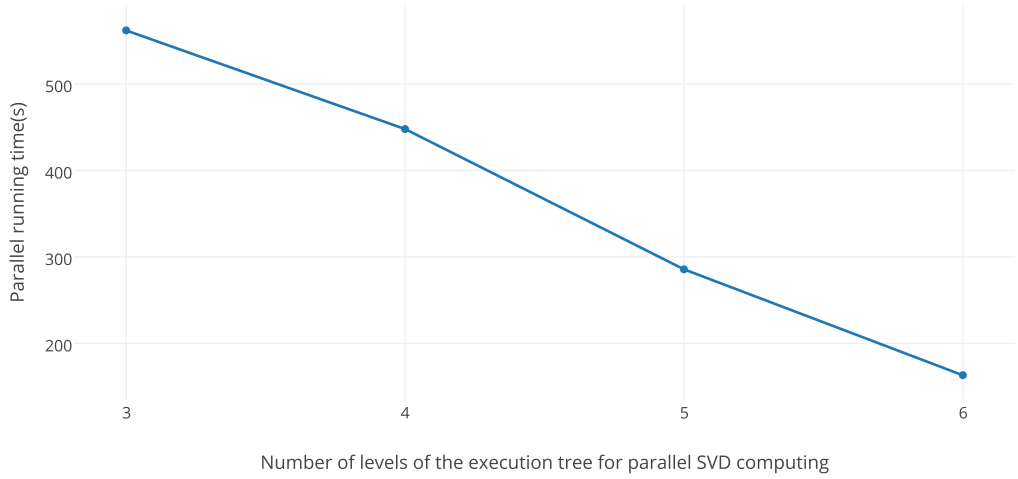

Figure 6.1: Comparison of running time with different numbers of levels of the execution tree of the parallel SVD algorithm ($m = 1024$ and $n = 32758$).

deterministic masks. See Figures 6.2 - 6.5.

6.3 Sparse principal component analysis

Principal component analysis (PCA) [58] is an important tool in multivariate data analysis and dimension reduction. Applications of PCA lies in numerous areas, such as machine learning, image compression and genetics, where analysis of large datasets are needed. However, PCA suffers from an obvious shortcoming: PCs are linear combinations of all the input variables, i.e. the loadings do not have many zero coefficients. In many applications, the input variables have physical meaning. The PCA is usually more interpretable if the PCs are combinations of only a small number


Figure 6.2: Original 768×1024 image with full rank (left); Original image truncated to rank 40 (right).


Figure 6.3: 50% randomly masked original image (left); Recovered image by FPCP ($rel.err = 6.93e-2$) (right).


Figure 6.4: 50% randomly masked rank 40 image (left); Recovered image by FPCP ($rel.err = 8.47e-2$) (right).

Figure 6.5: Deterministically masked rank 40 image (left); Recovered image by FPCP ($rel.err = 7.04e-2$) (right).

of the input variables. Thus, it is of interest to study sparse principal component analysis which seeks the trade-off between statistical fidelity (explaining as much variance as possible) and interpretability (using as few variables as possible).

In sparse PCA problem, we maximize the variance of a vector $x \in \mathbb{R}^n$ with constraints on its cardinality,

$$\begin{aligned} \max \quad & x^T A x \\ \text{s.t.} \quad & \|x\|_2 = 1, \\ & \text{Card}(x) \leq k. \end{aligned} \tag{6.6}$$

where $\text{Card}(x)$ is the cardinality of x, i.e.the number of non zero coefficients of x and $k > 0$ is the parameter controlling sparsity.

Sparse PCA is a computational hard problem because of the combinatorial nature of $\text{Card}(x)$. A very first approach [19] is an ad-hoc method where the loadings with small absolute values are artificially thresholded to zero. Several more involved approaches arose in the last decade including nonconvex optimization methods (SCoTLASS [59], SLRA [105, 106] and SPCA [108]), a greedy algorithm (Moghaddam et al.[77]) and a generalized power method (Journee et al. [62]). A direct approach, called DSPCA, was proposed by d'Aspremont et al. [29] in 2004. They formulated a convex relaxation of the sparse PCA problem and solved it by Nesterov's first-order optimization algorithm [78]. In this method, an eigendecomposition or SVD is needed in every iteration.

6.3.1 A DSPCA approach

A DSPCA algorithm is designed for the sparse PCA problem (6.6). A semidefinite relaxation of the problem is formed,

$$\begin{aligned} \max \quad & \mathbf{Tr}(AX) - \mathbf{1}^T|X|\mathbf{1} \\ \text{s.t.} \quad & \mathbf{Tr}(X) = 1, \\ & X \succeq 0, \end{aligned} \tag{6.7}$$

where $\mathbf{Tr}(X)$ is the trace of X, $\mathbf{1}$ is a n-vector of ones and X is symmetric. We can rewrite this problem in the saddle-function format by duality

$$\min_{U \in Q_1} f(U), \tag{6.8}$$

where

$$Q_1 = \{U \in \mathbf{S}^n : |U_{ij}| \leq 1, i, j = 1, \ldots, n\}, \tag{6.9}$$

$$Q_2 = \{X \in \mathbf{S}^n : \mathbf{Tr}(X) = 1, X \succeq 0\}, \tag{6.10}$$

$$f(U) = \max_{X \in Q_2} \langle TU, X \rangle - \hat{\phi}(X) \tag{6.11}$$

with

$$T = I_{n^2}, \quad \hat{\phi}(X) = -\mathbf{Tr}(AX). \tag{6.12}$$

The DSPCA approach uses Nesterov's smoothing technique to solve this problem, as stated in Algorithm 16.

Algorithm 16 DSPCA

Input: semidefinite $n \times n$ matrix U_0

Repeat:

1. Compute $f_\mu(U_k)$ and $\nabla f_\mu(U_k)$
2. Find $Y_k = \arg\min_{Y \in Q_1} \langle \nabla f_\mu(U_k), Y \rangle + \frac{1}{2} L \|U_k - Y\|_F^2$
3. Find $W_k = \arg\min_{W \in Q_1} \{\frac{L d_1(W)}{\sigma_1} + \sum_{i=0}^{k} \frac{i+1}{2} (f_\mu(U_i) + \langle \nabla f_\mu(U_i), W - U_i \rangle)\}$
4. Set $U_{k+1} = \frac{2}{k+3} W_k + \frac{k+1}{k+3} Y_k$

Until: duality gap $\leq \varepsilon$

In Algorithm 16, $f_\mu(U)$ is

$$f_\mu(U) = \max_{X \in Q_2} \langle TU, X \rangle - \hat{\phi}(X) - \mu d_2(X), \tag{6.13}$$

where μ is a regularisation parameter and $d_2(X) = \mathbf{Tr}(X \log X) + \log(n)$. In practice, $f_\mu(U)$ can be computed as:

$$f_\mu(U) = \sigma_1 + \mu \log(\sum_{i=1}^{n} y_i) - \mu \log n, \tag{6.14}$$

	Iter	Gap	oval
DSPCA with Matlab *svds*	1600	0.08378	1.022 $\times 10^4$
DSPCA with our algorithm	1600	0.08378	1.022 $\times 10^4$

Table 6.7: Numerical results for sparse PCA (n=200).

	Iter	Gap	oval
DSPCA with Matlab *svds*	1000	0.03948	1.009$\times 10^4$
DSPCA with our algorithm	1000	0.03947	1.009$\times 10^4$

Table 6.8: Numerical results for sparse PCA (n=500).

where $y_i = \exp(\frac{\sigma_i - \sigma_1}{\mu})$ and σ_i is i-th singular value of $Z = A + U_k$.

The gradient $\nabla f_\mu(U_k)$ is computed as

$$\nabla f_\mu(U_k) = V \mathrm{Diag}(h_i) V^T, \tag{6.15}$$

where $h_i = \frac{\exp(\frac{\sigma_i - \sigma_1}{\mu})}{\sum_{j=1}^n \exp(\frac{\sigma_i - \sigma_1}{\mu})}$.

Besides, L is the Lipschitz continuous gradient of f_μ and $d_1(W) = \frac{1}{2} W^{\mathrm{T}} W$.

The duality gap of iteration k is

$$\mathrm{gap}_k = \lambda_{\max}(A + U_k) - \mathbf{Tr}(AX_k) + \mathbf{1}^{\mathrm{T}} |X_k| \mathbf{1}. \tag{6.16}$$

In each iteration, we need to compute the spectral decomposition of Z. Computing the full spectral decomposition is costly but unnecessary. In fact, it is sufficient to compute the l largest eigenvalues of Z for some small number l and use approximate function values

$$\tilde{f}_\mu(U_k) = \sigma_1 + \mu \log(\sum_{i=1}^{l} y_i) - \mu \log n, \tag{6.17}$$

and gradients

$$\tilde{\nabla} f_\mu(U_k) = \sum_{i=1}^{l} h_i v_i v_i^T \tag{6.18}$$

Therefore, we can use our algorithm to compute the spectral decomposition of Z in this method, as Z is symmetric.

6.3.2 Numerical results

We tested our method on artificial matrices, $A = U^T U + \sigma v v^T$, where $U \in \mathbb{R}^{n \times n}$ is randomly generated, $\sigma = 20$ is a signal-to-noise ratio and $v \in \mathbb{R}^n$ is a sparse vector.

In our tests, we used $n = 200, 500$ respectively and compared the accuracy and performance of the variant of Algorithm 16 that uses our SVD algorithm with the vairant based on the Matlab built-in function *svds*. In the first case, we used the combination tree stated in Algorithm 10 with 8 seed nodes, computing the leading 5 singular values and vectors.

Table 4.1 and 4.2 list the results. 'Iter' is the number of iterations used and 'Gap' is the duality gap after termination. To check the convergence, we computed the duality gap after every 100 iterations. We also computed the objective value $oval = Tr(AX)$ to check the correctness. The numbers of iterations, the duality gap and the objective values of the variants of Algorithm 16 that uses our SVD algorithm and the Matlab built-in function *svds* are very close in both tests. This implies that the parallel SVD algorithm achieved a reasonable high level of accuracy.

Chapter 7

Conclusion and Future Directions

In this thesis, we presented a loosely coupled distributed algorithm for the computation of the leading part SVDs of large matrices, as well as its theoretical analysis and numerical experiments.

The development of this algorithm is motivated by the need of scientific computing on large-scale problems and the increasingly distributed nature of data and hardware. In the final comments, we discuss potential research directions in numerical linear algebra and sparse optimisation, based on similar approaches to our algorithm.

7.1 Future directions

7.1.1 Distributed methods in numerical linear algebra

The parallel methodology of the described SVD algorithm is to divide the large-scale problem into smaller ones by subspace splitting. The method computes the leading part SVDs of submatrices or outputs from other nodes at each computational node and combines the outputs of two or more of such calculations to produce the inputs of new nodes. The approximations of the leading part SVD are successively improved through this type of combination. The communication costs are reduced to a very low level because only thin and tall matrices are transferred between different nodes. Apart from communication-minimising, the method has very low synchronicity requirements and is robust to node failure.

This parallel methodology has potential to be used in other matrix computation problems, such as eigenproblem that shares similar properties with SVD. The method can perhaps be extended to these problems. Another possible application is tensor decomposition. Many tensor problems are solved through matricisation. SVD forms the

basis of many tensor decomposition methods. This naturally leads to the extension of the method to tensor decomposition problems.

7.1.2 Sparse optimisation

Most of the existing sparse optimisation algorithms do not lend themselves naturally to distributed implementation, which is necessary when dealing with data of extreme scale and those collected in a distributed manner. In Chapter 6, our parallisation of the FPCA scheme by combining it with our parallel SVD algorithm gives encouraging results. In many existing and emerging applications of SVD, SVD is not applied once but multiple times to a sequence of related iterations. Sometimes, SVD is applied to different but very similar inputs. For these applications, efforts have been made by researchers to explore parallel and distributed computing for solving large-scale instances [83, 84, 16, 17], independent of our effort to make the SVD subroutine parallel and distributed. Skillfully integrating theirs and our approach will lead to significant improvement in terms of efficiency and scalability, and may also motivate new directions to be pursued. Let me take low-rank matrix/tensor recovery and SDP as examples.

a. Tensor recovery. A tensor is a generalization of a vector and a matrix. Higher-order tensors naturally arise in many applications such as 3D image reconstruction [88], video inpainting [99] and higher-order web link analysis [66]. The software package Low Rank Tensor Completion (LRTC) recovers low-rank tensors by minimising the sum of the nuclear-norms of multiple 'projected matrices' of the unknown tensor. The scope of application of these solvers is seriously limited by SVD computations. Novel algorithms are needed to overcome this curse of dimensionality. It is possible to achieve this goal by integrating our distributed SVD algorithm into optimisation algorithms and make them all loosely coupled. We expect that this will significantly improve the scalability of low-rank tensor recovery.

b. Semidefinite Programming (SDP). Many optimisation problems can be formulated and solved as SDP problems, which are optimisation problems in which the unknowns are symmetric semidefinite matrices. Although powerful, current SDP algorithms can only solve problems of moderate sizes. There are two important classes of SDP algorithms. One is interior point methods [94, 98]. The drawback of interior point methods is that they solve a large linear system at every iteration, which limits the sizes of the problems to $n \times n$ matrices, where n^2 is approximately the memory size. The other class of SDP algorithm are first-order methods, such as alternating direction methods of multipliers (ADMM) [101], semismooth Newton

CG method [107] and regularization methods [73]. These first-order methods share the same bottleneck that a (partial) eigenvalue-decomposition or SVD is required at each iteration. The efficiency of SVD determines the efficiency of these methods. However, computing SVD from scratch is not necessary. The structure of the SDP problems induces the inputs of the SVD steps. To explore this by using the parallel methodology in the distributed SVD algorithm may be able to improve the scalability of SPD solutions.

Appendix A

Matlab code for the parallel SVD method

```
function [U,S,V,time] = pasvd(A, p, level, err)
Q = A(1:2*p, :)';
time = zeros(level, 2^(level-1));
[R, S, V, time] = tree(A, Q, p, err, level, 1, time);
V=V(:,1:p)';
S=S(1:p,1:p);
U=R(:,1:p)*inv(S);

function [R, S, V, time] = tree(A, Q, p, err, level, position, time)
if level == 1
    tic
    [S, V] = localCalculation(A, Q, p, err);
    R = A*V;
    time(level, position) = toc;
else
    [m, n] = size(A);
    d = floor(n/2);
    A1 = A(:,1:d);
    Q1 = Q(1:d,:);
    A2 = A(:,d+1:n);
    Q2 = Q(d+1:n,:);
    [R1, S1, V1, time] = tree(A1, Q1, p, err, level-1, position*2-1, time);
    [R2, S2, V2, time] = tree(A2, Q2, p, err, level-1, position*2 , time);
    tic
```

```
    V1 = [V1', zeros(p,n-d)]';
    V2 = [zeros(p,d), V2']';
    VT = [V1,V2];
    A = [R1,R2];
    Q = A(1:2*p, :)';
    [S, V] = localCalculation(A, Q, p, err);
    R = A*V;
    V = VT*V;
    time(level, position) = toc;
end

function [S, Q] = localCalculation(A, Q, p, err)
[qm,qn]=size(Q);
Qprev = Q;
[S, Q] = iteration_SVD(A, Q, p);
while (norm(Q(:,1:p)-Qprev(:,1:p))) > err
    Qprev = Q;
    [S, Q] = iteration_SVD(A, Q, p);
    for i=1:qn
        factor=(Q(:,i)'*Qprev(:,i))/abs(Q(:,i)'*Qprev(:,i));
        Q(:,i)=factor*Q(:,i);
    end
end
Q = Q(:,1:p);

function [S, Q] = iteration_SVD(A, Q, p)
    [U,S,V] = svd(A*Q);
    X = Q * V(:,1:p);
    S = S(1:p,1:p); %Set S to the diagonal entries
    [Qp, Rp] = qr(A'*U(:,1:p)); %Calculate the new value of Q
    [Q, R] = qr([X, Qp(:,1:p)]);
    Q=Q(:,1:2*p);
```

Appendix B

Sequential Python code for the parallel SVD method

```
import numpy as np
import sys
from mpi4py import MPI

comm = None
nproc = None
rank = None
nlevel = None
runtimes = None

def seedNode(At, p):
    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Aseed = Ut*np.diag(St)
    Vseed = Vt

    return Aseed, Vseed


def combinationNode(At1, At2, Vt1, Vt2):
    At  = np.concatenate((At1, At2), axis=1)
    Vtt1 = np.concatenate((Vt1, np.zeros(Vt1.shape)), axis =1)
```

```
    Vtt2 = np.concatenate((np.zeros(Vt2.shape), Vt2),axis =1)
    Vtt = np.concatenate((Vtt1,Vtt2))
    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Qt, Rt = np.linalg.qr(np.dot(Vtt, Vt))
    Acomb = np.dot(Ut, np.dot(np.diag(St), np.linalg.inv(Rt)))
    Vcomb = Qt

    return Acomb, Vcomb


def extractionNode(At1, At2, Vt1, Vt2):
    At  = np.concatenate((At1, At2), axis=1)
    Vtt1 = np.concatenate((Vt1, np.zeros(Vt1.shape)), axis =1)
    Vtt2 = np.concatenate(( np.zeros(Vt2.shape), Vt2),axis =1)

    Vtt = np.concatenate((Vtt1,Vtt2))

    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Qt,Rt = np.linalg.qr(np.dot(Vtt, Vt))
    Abreve = np.dot(Ut, np.dot(np.diag(St), np.linalg.inv(Rt)))
    U, S, Vbar= np.linalg.svd(Abreve, full_matrices=False)

    V = np.dot(Qt, Vbar)

    S = np.diag(S)

    return U,S,V


if __name__ == "__main__":
```

```
m = int(sys.argv[1])
n = int(sys.argv[2])

comm = MPI.COMM_WORLD
nproc = comm.Get_size()
rank = comm.Get_rank()
nlevel = int(np.floor(np.log(nproc)/np.log(2E0) + 1E0))
runtimes = np.zeros((nlevel, nproc, 3), dtype=np.double)

p = 5
r = 10
alpha = 10

for index in range(50):

    A = None
    nSeedNodes = 2 ** (nlevel - 1)
    Adict = None
    Bdict = None
    ALocal = np.empty((m, int(n / nSeedNodes)), dtype=np.double)
    U = None
    S = None
    V = None
    Uout = None
    Sout = None
    Vout = None

    Adict = np.empty((2, m, p), dtype=np.double)
    Vdict = np.empty((2, int(n / nSeedNodes), p), dtype=np.double)

    if (rank == 0):
        np.random.seed(1)

        Q = np.random.random((m, r))
        U, R = np.linalg.qr(Q)
```

```
    Q = np.random.random((n, r))
    V, R = np.linalg.qr(Q)

    Q = R = None

    S = np.zeros((r, r))
    S[0,0]=100
    for i in range(1,r):
        S[i,i] = S[i-1,i-1]/alpha

    U = np.mat(U)
    S = np.mat(S)
    V = np.mat(V)
    A = U * S * V.T

    for i in range(1, nSeedNodes):
        ALocal = A[:, int(n / nSeedNodes * i):int(n /
                                                  nSeedNodes * (i + 1))]
        t0 = MPI.Wtime()
        comm.Send(ALocal.flatten(), dest=i, tag=i)
        runtimes[0, rank, 1] += MPI.Wtime() - t0
    ALocal = A[:, 0:int(n / nSeedNodes)].copy()
    A = None
else:
    t0 = MPI.Wtime()
    comm.Recv(ALocal, source=0, tag=rank)
    runtimes[0, rank, 2] = MPI.Wtime() - t0
    ALocal = np.asmatrix(ALocal.reshape((m, int(n / nSeedNodes))))
comm.Barrier()

telapsed = 0
if (rank < nSeedNodes):
    t0 = MPI.Wtime()
    Aseed, Vseed = seedNode(ALocal, p)
    AdictLocal = Aseed
    VdictLocal = Vseed
```

```
    runtimes[0, rank, 0] += MPI.Wtime() - t0
    if (rank % 2 != 0):
        t00 = MPI.Wtime()
        comm.Send([AdictLocal.flatten(), MPI.DOUBLE],
                  dest=rank - 1, tag=rank)
        comm.Send([VdictLocal.flatten(), MPI.DOUBLE],
                  dest=rank - 1, tag=rank + nproc)
        runtimes[1, rank, 1] = MPI.Wtime() - t00
    else:
        Adict[0] = AdictLocal
        Vdict[0] = VdictLocal
        AdictLocal = np.empty((m, p))
        VdictLocal = np.empty((int(n / nSeedNodes), p))
        t00 = MPI.Wtime()
        comm.Recv([AdictLocal, MPI.DOUBLE], source=rank + 1,
                  tag=rank + 1)
        comm.Recv([VdictLocal, MPI.DOUBLE], source=rank + 1,
                  tag=rank + 1 + nproc)
        runtimes[1, rank, 2] = MPI.Wtime() - t00
        Adict[1] = AdictLocal.reshape((m, p))
        Vdict[1] = VdictLocal.reshape((int(n / nSeedNodes), p))
    telapsed += MPI.Wtime() - t0
comm.Barrier()
ALocal = None

At1 = Vt1 = At2 = Vt2 = None
for i in range(nlevel - 2):
    nCombNodes = 2 ** (nlevel - i - 2)
    if ((rank % (2**(i+1))== 0) and (rank < nSeedNodes)):
        t0 = MPI.Wtime()

        Adictnew = np.copy(Adict)
        Vdictnew = np.copy(Vdict)

        Adict = np.empty((2, m, p))
        Vdict = np.empty((2, int(n / nCombNodes), p))
```

```
            At1 = Adictnew[0]
            Vt1 = Vdictnew[0]
            At2 = Adictnew[1]
            Vt2 = Vdictnew[1]

            Acomb, Vcomb = combinationNode(At1, At2, Vt1, Vt2)
            AdictLocal = Acomb
            VdictLocal = Vcomb

            runtimes[i + 1, rank, 0] = MPI.Wtime() - t0
            if (rank % (2**(i+2)) != 0):
                t00 = MPI.Wtime()
                comm.Send([AdictLocal.flatten(), MPI.DOUBLE],
                          dest=rank - 2**(i+1), tag=rank)
                comm.Send([VdictLocal.flatten(), MPI.DOUBLE],
                          dest=rank - 2**(i+1), tag=rank+nproc)
                runtimes[i + 2, rank, 1] = MPI.Wtime() - t00
            else:
                Adict[0] = AdictLocal
                Vdict[0] = VdictLocal
                AdictLocal = np.empty((m, p))
                VdictLocal = np.empty((int(n / nCombNodes), p))
                t00 = MPI.Wtime()
                comm.Recv([AdictLocal, MPI.DOUBLE], source=rank + 2**(i+1),
                          tag=rank + 2**(i+1))
                comm.Recv([VdictLocal, MPI.DOUBLE], source=rank + 2**(i+1),
                          tag=rank + 2**(i+1) + nproc)
                runtimes[i + 2, rank, 2] = MPI.Wtime() - t00
                Adict[1] = AdictLocal.reshape((m, p))
                Vdict[1] = VdictLocal.reshape((int(n / nCombNodes), p))
            telapsed += MPI.Wtime() - t0
        comm.Barrier()
    comm.Barrier()

    t0 = MPI.Wtime()
```

```
    AdictLocal = VdictLocal = None
    if (rank == 0):
        Adictnew = Vdictnew = None
        At1 = Adict[0]
        Vt1 = Vdict[0]
        At2 = Adict[1]
        Vt2 = Vdict[1]
        Uout, Sout, Vout = extractionNode(At1, At2, Vt1, Vt2)
    runtimes[nlevel - 1, 0, 0] = MPI.Wtime() - t0

    Adict = Vdict = None
    At1 = Vt1 = At2 = Vt2 = None
    telapsed += MPI.Wtime() - t0

    fout = None
    if (rank == 0):
        U = V = S = None
    comm.Barrier()

    for j in range(1, nproc):
        if (rank == j):
            comm.send(telapsed, dest=0, tag=rank)
        elif (rank == 0):
            telapsed += comm.recv(source=j, tag=j)
    comm.Barrier()
comm.Barrier()
```

Appendix C

Parallel Python code for the parallel SVD method

```
import numpy as np


def computeSVD(A, nlevel,m,n):
    print  "computing SVD"

    nSeedNodes  = 2**(nlevel-1)

    Adict = {}
    Vdict = {}
    for i in range(nSeedNodes):
        Aseed,Vseed = seedNode(A[:,n/nSeedNodes*(i):n/nSeedNodes*(i+1)])
        Adict[i] = Aseed
        Vdict[i] =Vseed

    for i in range(nlevel-2):
        Adictnew = Adict
        Vdictnew = Vdict
        for j in range(2**(nlevel-i-2)):
            At1 = Adictnew[j*2]
            Vt1  = Vdictnew[j*2]
            At2 = Adictnew[j*2+1]
            Vt2 = Vdictnew[j*2+1]
            Acomb, Vcomb = combinationNode(At1,At2,Vt1, Vt2)
```

```
            Adict[j] = Acomb
            Vdict[j] = Vcomb

    At1 = Adict[0]
    Vt1 = Vdict[0]
    At2 = Adict[1]
    Vt2 = Vdict[1]

    U,S,V = extractionNode(At1,At2,Vt1, Vt2)

    return U, S, V




def seedNode(At):
    print  "Seed node"

    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Aseed = Ut*np.diag(St)
    Vseed = Vt

    return Aseed, Vseed

def combinationNode(At1,At2,Vt1, Vt2):
    print  "Combination node"

    At  = np.concatenate((At1, At2), axis=1)
    Vtt1 = np.concatenate((Vt1, np.zeros(Vt1.shape)), axis =1)
    Vtt2 = np.concatenate(( np.zeros(Vt2.shape), Vt2),axis =1)
    Vtt = np.concatenate((Vtt1,Vtt2))
    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
```

```
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Qt,Rt = np.linalg.qr(Vtt*Vt)
    Acomb = Ut*np.diag(St)*np.linalg.inv(Rt)
    Vcomb = Qt

    return Acomb, Vcomb

def extractionNode(At1,At2,Vt1, Vt2):
    print  "Extraction node"

    At  = np.concatenate((At1, At2), axis=1)
    Vtt1 = np.concatenate((Vt1, np.zeros(Vt1.shape)), axis =1)
    Vtt2 = np.concatenate(( np.zeros(Vt2.shape), Vt2),axis =1)

    Vtt = np.concatenate((Vtt1,Vtt2))

    Ut, St, Vt = np.linalg.svd(At, full_matrices=False)
    Ut = Ut[:,:p]
    St = St[:p]
    Vt = Vt[:,:p]
    Qt,Rt = np.linalg.qr(Vtt*Vt)
    Abreve = Ut*np.diag(St)*np.linalg.inv(Rt)
    U, S, Vbar= np.linalg.svd(Abreve, full_matrices=False)

    V = Qt*Vbar

    S = np.diag(S)

    return U,S,V


if __name__ == "__main__":

    m = 2000
```

```
n = 2000

nlevel = 5
p = 5
r = 10
alpha = 10

Q = np.random.random((m,m))
U, R = np.linalg.qr(Q)

Q = np.random.random((n,n))
V, R = np.linalg.qr(Q)

S = np.zeros((m,n))
S[0,0]=100
for i in range(1,r):
    S[i,i] = S[i-1,i-1]/alpha



U = np.mat(U)
S = np.mat(S)
V= np.mat(V)
A = U*S*V.T

Uout, Sout, Vout = computeSVD(A, nlevel,m,n)
Aout = Uout*Sout*Vout.T

Up = U[:, :p]
Vp = V[:, :p]
Sp = S[:p,:p]
Ap  = Up *Sp*Vp.T

Uerr = np.linalg.norm(Up*Up.T-Uout*Uout.T)
Serr = np.linalg.norm(Sp-Sout)
Verr = np.linalg.norm(Vp*Vp.T-Vout*Vout.T)
Aerr = np.linalg.norm(Ap-Aout)
```

```
print Uerr, Serr, Verr, Aerr

print Sout
```

Bibliography

[1] Climateprediction.net. *http://www.climateprediction.net*.

[2] Cosmos. *http://www.damtp.cam.ac.uk/cosmos/*.

[3] Nir Ailon and Bernard Chazelle. The fast Johnson-Lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322, 2009.

[4] Orly Alter, Patrick O Brown, and David Botstein. Singular value decomposition for genome-wide expression data processing and modeling. *Proceedings of the National Academy of Sciences*, 97(18):10101–6, 2000.

[5] James Baglama and Lothar Reichel. Augmented implicitly restarted Lanczos bidiagonalization methods. *SIAM Journal on Scientific Computing*, 27(1):19–42, 2005.

[6] Maria-Florina Balcan, Yingyu Liang, Le Song, David Woodruff, and Bo Xie. Communication efficient distributed kernel principal component analysis. *arXiv preprint arXiv:1503.06858*, 2015.

[7] Jayant Baliga, Robert WA Ayre, Kerry Hinton, and Rodney S Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, 2011.

[8] Grey Ballard, James Demmel, and Ioana Dumitriu. Minimizing communication for eigenproblems and the singular value decomposition. *arXiv preprint arXiv:1011.3077*, 2010.

[9] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[10] Eugenio Beltrami. Sulle funzioni bilineari. *Giornale di Matematiche ad Uso degli Studenti Delle Universita*, 11:98–106, 1873.

[11] Luca Bergamaschi and Mario Putti. Numerical comparison of iterative eigensolvers for large sparse symmetric positive definite matrices. *Computer Methods in Applied Mechanics and Engineering*, 191(45):5233–5247, 2002.

[12] Åke Björck and Gene H Golub. Numerical methods for computing angles between linear subspaces. *Mathematics of Computation*, 27(123):579–594, 1973.

[13] Jeffrey D Blanchard, Jared Tanner, and Ke Wei. Conjugate gradient iterative hard thresholding: observed noise stability for compressed sensing. *Signal Processing, IEEE Transactions on*, 63(2):528–537, 2015.

[14] Christos Boutsidis, Michael W Mahoney, and Petros Drineas. An improved approximation algorithm for the column subset selection problem. In *Proceedings of the twentieth Annual ACM-SIAM SODA*, pages 968–977. Society for Industrial and Applied Mathematics, 2009.

[15] Christos Boutsidis, David P Woodruff, and Peilin Zhong. Communication-optimal distributed principal component analysis in the column-partition model. *CoRR, abs/1504.06729*, 2015.

[16] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[17] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *In ICML, pages 321328*, 2011.

[18] Angelika Bunse-Gerstner, Ralph Byers, and Volker Mehrmann. A chart of numerical methods for structured eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 13(2):419–453, 1992.

[19] Jorge Cadima and Ian T Jolliffe. Loading and correlations in the interpretation of principle components. *Journal of Applied Statistics*, 22(2):203–214, 1995.

[20] Emmanuel J Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, 2009.

[21] Emmanuel J Candès, Justin Romberg, and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *Information Theory, IEEE Transactions on*, 52(2):489–509, 2006.

[22] Emmanuel J Candès and Terence Tao. The power of convex relaxation: Near-optimal matrix completion. *Information Theory, IEEE Transactions on*, 56(5):2053–2080, 2010.

[23] Michael B Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 163–172. ACM, 2015.

[24] Dean Croushore. Frontiers of real-time data analysis. *Journal of Economic Literature*, pages 72–100, 2011.

[25] Dominik Csiba and Peter Richtárik. Primal method for ERM with flexible mini-batching schemes and non-convex losses. *arXiv preprint arXiv:1506.02227*, 2015.

[26] Jane Cullum. The simultaneous computation of a few of the algebraically largest and smallest eigenvalues of a large, sparse, symmetric matrix. *BIT Numerical Mathematics*, 18(3):265–275, 1978.

[27] Jane Cullum and WE Donath. A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices. In *Decision and Control including the 13th Symposium on Adaptive Processes, 1974 IEEE Conference on*, pages 505–509. IEEE, 1974.

[28] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures & Algorithms*, 22(1):60–65, 2003.

[29] Alexandre d'Aspremont, Laurent El Ghaoui, Michael I Jordan, and Gert RG Lanckriet. A direct formulation for sparse PCA using semidefinite programming. *SIAM review*, 49(3):434–448, 2007.

[30] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[31] Francis X Diebold and Glenn D Rudebusch. Forecasting output with the composite leading index: A real-time analysis. *Journal of the American Statistical Association*, 86(415):603–610, 1991.

[32] David L Donoho and Xiaoming Huo. Uncertainty principles and ideal atomic decomposition. *IEEE Transactions on Information Theory*, 47(7):2845–2862, 2001.

[33] Petros Drineas and Ravi Kannan. Pass efficient algorithms for approximating large matrices. In *SODA*, volume 3, pages 223–232, 2003.

[34] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM Journal on Computing*, 36(1):158, 2006.

[35] Petros Drineas, Michael W Mahoney, and S Muthukrishnan. Relative-error CUR matrix decompositions. *SIAM Journal on Matrix Analysis and Applications*, 30(2):844–881, 2008.

[36] John C Duchi, Alekh Agarwal, and Martin J Wainwright. Dual averaging for distributed optimization: convergence analysis and network scaling. *IEEE Transactions on Automatic Control*, 57(3):592–606, 2012.

[37] Maryam Fazel. *Matrix rank minimization with applications*. PhD thesis, Stanford University, 2002.

[38] Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1434–1453. Society for Industrial and Applied Mathematics, 2013.

[39] Daniel J Fenn, Mason A Porter, Stacy Williams, Mark McDonald, Neil F Johnson, and Nick S Jones. Temporal evolution of financial-market correlations. *Physical Review E*, 84(2):026109, 2011.

[40] Peter Frankl and Hiroshi Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Series B*, 44(3):355–362, 1988.

[41] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[42] Gene H Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, pages 205–224, 1965.

[43] Gene H Golub, Franklin T Luk, and Michael L Overton. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Transactions on Mathematical Software (TOMS)*, 7(2):149–169, 1981.

[44] Gene H Golub and Richard Underwood. The block Lanczos method for computing eigenvalues. *Mathematical software*, 3:361–377, 1977.

[45] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. Johns Hopkins Univ. Pr., 1996.

[46] Gene H Golub and Qiang Ye. An inverse free preconditioned Krylov subspace method for symmetric generalized eigenvalue problems. *SIAM Journal on Scientific Computing*, 24(1):312–334, 2002.

[47] Daniel Goodman. *A Service-Oriented Architecture and Language for Abstracted Distributed Algorithms*. PhD thesis, Oxford University Computing Laboratory, 2007.

[48] Susan L Graham, Marc Snir, Cynthia A Patterson, et al. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.

[49] N. Halko, P.G. Martinsson, and J.A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *Arxiv preprint arXiv:0909.4061*, 2009.

[50] John Michael Hammersley and David Christopher Handscomb. *Monte Carlo Methods*, volume 1. Methuen London, 1964.

[51] George W Hart. Norms, adjoints, and singular value decomposition. In *Multidimensional Analysis*, pages 119–144. Springer, 1995.

[52] V Hernandez, JE Roman, A Tomas, and V Vidal. A survey of software for sparse eigenvalue problems. *Universitat Politecnica de Valencia, Tech. Rep. STR-6,[retrieved: May, 2013].[Online]. Available: http://www. grycap. upv. es/slepc*, 2005.

[53] Neal S Holter, Madhusmita Mitra, Amos Maritan, Marek Cieplak, Jayanth R Banavar, and Nina V Fedoroff. Fundamental patterns underlying gene expression profiles: simplicity from complexity. *Proceedings of the National Academy of Sciences*, 97(15):8409–8414, 2000.

[54] Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge University Presss, Cambridge, 1990.

[55] Roger A Horn and Charles R Johnson. Topics in matrix analysis. 1991.

[56] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM STOC*, pages 604–613. ACM, 1998.

[57] William B Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary Mathematics*, 26(189-206):1, 1984.

[58] Ian T Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[59] Ian T Jolliffe, Nickolay T Trendafilov, and Mudassir Uddin. A modified principal component technique based on the LASSO. *Journal of Computational and Graphical Statistics*, 12(3):531–547, 2003.

[60] Camille Jordan. Mémoire sur les formes bilinéaires. *Journal de Mathématiques Pures et Appliquées*, pages 35–54, 1874.

[61] Camille Jordan. Sur la réduction des formes bilinéaires. *Comptes Rendus de l'Académie des Sciences*, 78:614–617, 1874.

[62] Michel Journée, Yurii Nesterov, Peter Richtárik, and Rodolphe Sepulchre. Generalized power method for sparse principal component analysis. *The Journal of Machine Learning Research*, 11:517–553, 2010.

[63] Shmuel Kaniel. Estimates for some computational techniques in linear algebra. *Mathematics of Computation*, pages 369–378, 1966.

[64] Ravi Kannan, Santosh Vempala, and David P Woodruff. Principal component analysis and higher correlations for distributed data. In *COLT*, pages 1040–1057, 2014.

[65] Tomoko Kinoshita, Osamu Hino, and Rodney J Bartlett. Singular value decomposition approach for the approximate coupled-cluster method. *The Journal of Chemical Physics*, 119(15):7756–7762, 2003.

[66] Tamara G Kolda, Brett W Bader, and Joseph P Kenny. Higher-order web link analysis using multilinear algebra. In *Fifth IEEE International Conference on Data Mining*, pages 8–pp. IEEE, 2005.

[67] Rasmus M Larsen. Propack-software for large and sparse svd calculations. *Available online. URL http://sun. stanford. edu/rmunk/PROPACK*, 2004.

[68] Jason D Lee, Yuekai Sun, Qiang Liu, and Jonathan E Taylor. Communication-efficient sparse regression: a one-shot approach. *arXiv preprint arXiv:1503.04337*, 2015.

[69] John G Lewis. Algorithms for sparse matrix eigenvalue problems. Technical report, Calif. Univ. Stanford. Comput. Sci. Dept., 1977.

[70] Yingyu Liang, Maria-Florina F Balcan, Vandana Kanchanapally, and David Woodruff. Improved distributed principal component analysis. In *Advances in Neural Information Processing Systems*, pages 3113–3121, 2014.

[71] Shiqian Ma, Donald Goldfarb, and Lifeng Chen. Fixed point and Bregman iterative methods for matrix rank minimization. *Mathematical Programming*, 128(1):321–353, 2011.

[72] Michael W Mahoney and Petros Drineas. CUR matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

[73] Jérôme Malick, Janez Povh, Franz Rendl, and Angelika Wiegele. Regularization methods for semidefinite programming. *SIAM Journal on Optimization*, 20(1):336–356, 2009.

[74] Neil Massey, T Aina, M Allen, C Christensen, D Frame, D Goodman, J Kettleborough, A Martin, S Pascoe, and D Stainforth. Data access and analysis

with distributed federated data servers in climateprediction. net. *Advances in Geosciences*, 8:49–56, 2006.

[75] Jiří Matoušek. On variants of the Johnson–Lindenstrauss lemma. *Random Structures & Algorithms*, 33(2):142–156, 2008.

[76] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[77] Baback Moghaddam, Yair Weiss, and Shai Avidan. Spectral bounds for sparse PCA: Exact and greedy algorithms. In *Advances in Neural Information Processing Systems*, pages 915–922, 2005.

[78] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005.

[79] Christopher C Paige. *The computation of eigenvalues and eigenvectors of very large sparse matrices*. PhD thesis, University of London, 1971.

[80] Émile Picard. Sur un théorème général relatif aux équations intégrales de première espèce et sur quelques problèmes de physique mathématique. *Rendiconti del Circolo Matematico di Palermo*, 29(1):79–97, 1910.

[81] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[82] Zheng Qu, Peter Richtárik, and Tong Zhang. Randomized dual coordinate ascent with arbitrary sampling. *arXiv preprint arXiv:1411.5873*, 2014.

[83] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, pages 1–52, 2012.

[84] Peter Richtárik and Martin Takáč. Distributed coordinate descent method for learning with big data. *arXiv preprint arXiv:1310.2059*, 2013.

[85] Mark Rudelson and Roman Vershynin. Sampling from large matrices: An approach through geometric functional analysis. *Journal of the ACM (JACM)*, 54(4):21, 2007.

[86] Axel Ruhe. Implementation aspects of band Lanczos algorithms for computation of eigenvalues of large sparse symmetric matrices. *Mathematics of Computation*, 33(146):680–687, 1979.

[87] Yousef Saad. On the rates of convergence of the Lanczos and the block-Lanczos methods. *SIAM Journal on Numerical Analysis*, 17(5):687–706, 1980.

[88] Anne C Sauve, Alfred O Hero, W Leslie Rogers, Scott J Wilderman, and Neal H Clinthorne. 3D image reconstruction for a compton SPECT camera model. *Nuclear Science, IEEE Transactions on*, 46(6):2075–2084, 1999.

[89] Ohad Shamir, Nathan Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate Newton-type method. In *ICML*, volume 32, pages 1000–1008, 2014.

[90] Dave Stainforth, Jamie Kettleborough, Andrew Martin, Andrew Simpson, Richard Gillis, Ali Akkas, Richard Gault, Mat Collins, David Gavaghan, and Myles Allen. Climateprediction. net: Design principles for publicresource modeling research. In *IASTED PDCS*, pages 32–38. Citeseer, 2002.

[91] Gilbert W Stewart. Simultaneous iteration for computing invariant subspaces of non-Hermitian matrices. *Numerische Mathematik*, 25(2):123–136, 1976.

[92] Ameet Talwalkar and Afshin Rostamizadeh. Matrix coherence and the Nystrom method. *arXiv preprint arXiv:1004.2008*, 2010.

[93] Jared Tanner and Ke Wei. Normalized iterative hard thresholding for matrix completion. *SIAM Journal on Scientific Computing*, 35(5):104–125, 2013.

[94] Michael J Todd. Semidefinite optimization. *Acta Numerica 2001*, 10:515–560, 2001.

[95] Matthew A Turk and Alex P Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[96] Matthew A Turk and Alex P Pentland. Face recognition using eigenfaces. In *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*, pages 586–591. IEEE, 1991.

[97] Richard Underwood. Iterative block Lanczos method for the solution of large sparse symmetric eigenproblems.[driver program with principal subroutine minval, in FORTRAN IV]. Technical report, Stanford Univ., Calif.(USA). Dept. of Computer Science, 1975.

[98] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[99] Haomian Wang, Houqiang Li, and Baoxin Li. Video inpainting for largely occluded moving human. In *2007 IEEE International Conference on Multimedia and Expo*, pages 1719–1722. IEEE, 2007.

[100] Per-Åke Wedin. Perturbation bounds in connection with singular value decomposition. *BIT Numerical Mathematics*, 12(1):99–111, 1972.

[101] Zaiwen Wen, Donald Goldfarb, and Wotao Yin. Alternating direction augmented Lagrangian methods for semidefinite programming. *Mathematical Programming Computation*, 2(3-4):203–230, 2010.

[102] Hermann Weyl. Das asymptotische Verteilungsgesdz der Eigenwerte linearer partieller Differentialgleichungen (mit einer Anwendung auf die Theorie der Hohlraumstrahlung). *Mathematische Annalen*, 71(1):441–479, 1912.

[103] James H Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon Press Oxford, 1965.

[104] Yuchen Zhang, Martin J Wainwright, and John C Duchi. Communication-efficient algorithms for statistical optimization. In *Advances in Neural Information Processing Systems*, pages 1502–1510.

[105] Zhenyue Zhang, Hongyuan Zha, and Horst Simon. Low-rank approximations with sparse factors I: Basic algorithms and error analysis. *SIAM Journal on Matrix Analysis and Applications*, 23(3):706–727, 2002.

[106] Zhenyue Zhang, Hongyuan Zha, and Horst Simon. Low-rank approximations with sparse factors II: Penalized methods with discrete Newton-like iterations. *SIAM journal on matrix analysis and applications*, 25(4):901–920, 2004.

[107] Xin-Yuan Zhao, Defeng Sun, and Kim-Chuan Toh. A Newton-CG augmented Lagrangian method for semidefinite programming. *SIAM Journal on Optimization*, 20(4):1737–1765, 2010.

[108] Hui Zou, Trevor Hastie, and Robert Tibshirani. Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, 15(2):265–286, 2006.