論文 / 著書情報
Article / Book Information

題目(和文)	ランク構造固有値問題に対する階層的QR分解とLDL分解
Title(English)	Hierarchical QR and LDL Decomposition for Rank-Structured Eigenvalue Problems
著者(和文)	Apriansyah Muhammad Ridwan
Author(English)	Muhammad Ridwan Apriansyah
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12560号, 授与年月日:2023年9月22日, 学位の種別:課程博士, 審査員:横田 理央,宮崎 純,関嶋 政和,下坂 正倫,小野 峻佑
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12560号, Conferred date:2023/9/22, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Hierarchical QR and LDL Decomposition for Rank-Structured Eigenvalue Problems

Muhammad Ridwan Apriansyah
Department of Computer Science
School of Computing
Tokyo Institute of Technology

Advisor: Rio Yokota

A thesis submitted for the degree of
Doctor of Engineering

2023

Tokyo Tech

Contents

List of Figures

Abstract

Hierarchical low-rank approximation can significantly reduce the cost for dense matrix decomposition from $\mathcal{O}(n^3)$ down to $\mathcal{O}(n)$. This thesis studies two hierarchical matrix decompositions that play a crucial role in solving eigenvalue problems, which are fundamental in numerous scientific and engineering domains. We first present a novel and highly parallel Block Low-Rank (BLR) QR decomposition based on the numerically stable Householder reflection. We also propose a scalable linear time generalized LDL decomposition based on the ULV decomposition. Numerical results show that our methods are more than an order of magnitude faster than state-of-the-art dense matrix decomposition and eigensolver in LAPACK/ScaLAPACK. Furthermore, our methods exhibit remarkable parallel scalability, utilizing up to tens of thousands of CPU cores across multiple computing nodes.

Acknowledgements

I would like to take this opportunity to express my heartfelt gratitude to the individuals who have played pivotal roles in making my Ph.D. studies a truly enriching and rewarding experience.

First and foremost, I would like to express my sincere gratitude to Professor Rio Yokota for his unwavering guidance and encouragement throughout my Ph.D. studies. His continuous support has been instrumental in motivating me to persevere and reach the finish line. I truly have learned many valuable lessons from him and this thesis would not have been possible without his mentorship over the past five years. I extend my gratitude to Professor Yasuhiro Matsumoto for his constructive feedback and support in writing this thesis. I am also thankful to Professor Takeo Hoshi (Department of Applied Mathematics and Physics, Tottori University) for introducing me to the significance of eigenvalue problems for electronic structure calculations. I greatly appreciate the efforts that he has made to provide me with the necessary models and data to enhance the quality of my research.

No words can express the depth of my gratitude to my family for their unparalleled love and support. I am forever indebted to my parents for nurturing and shaping me to become the person I am today. Despite the distance that separated us, their continuous support has been a constant source of strength, enabling me to explore new horizons in my life. To my beloved wife Desy and our son Rei, I offer my deepest appreciation for being my main support system here in Japan. Your unwavering belief in me has been a driving force behind my achievements, and I owe the successful completion of this thesis to your constant encouragement and understanding.

I am also grateful to my colleagues Sameer Deshmukh, Thomas Spendlhofer, and Qianxiang Ma for the valuable insights and fruitful discussions that have significantly enriched my understanding of the research field. A special mention goes to Hiroyuki Ootomo, Yuichiro Ueno, Edgar Josafat Martinez Noriega, and all members of Rio Yokota Lab who have added pleasant colors to my life in Japan with their kind and sociable gestures. The wonderful experiences we shared have made this journey all the more memorable.

Chapter 1

Introduction

1.1 Background

Eigenvalue problems lie at the core of computational science and engineering. Their numerical solution plays a crucial role in many application areas such as studying dynamics of electromagnetic fields, electronic state calculations, vibration analysis, particle accelerator and neutron flow simulations, and many more [58]. In order to solve eigenvalue problems for general (symmetric) dense matrices, a typical solution begins with tridiagonal reduction using orthogonal transformations, followed by applying iterative methods based on QR, bisection, or divide-and-conquer algorithms [83]. This kind of traditional dense eigenvalue solver usually requires $\mathcal{O}(n^3)$ flops. Although this approach is efficient for small matrices, the cost quickly becomes prohibitive when the matrix size is large, e.g. $n \geq 10^6$. As the rapid advances in scientific computing demand solving for large matrices, fast and accurate eigenvalue algorithms are desired to tackle complex and large-scale problems.

Dense matrices arising from many applications have been shown to possess a structured low-rank property such that a large portion of their off-diagonal blocks have fast-decaying singular values, i.e. small numerical rank. This class of matrices is referred to as *structured low-rank* matrices. They often arise from the discretization of partial differential and integral equations that govern a wide range of problems [12, 51]. In particular, the eigenvalue problems arising from partial differential equations are known to be very important in many application areas [58], which has given rise to the term *rank-structured eigenvalue problems*. Many structured low-rank formats have been proposed, including Block Low-Rank (BLR) [5], BLR^2 [11], Hierarchically Off-Diagonal Low-Rank (HODLR) [4], Quasiseparable/Semiseparable [28, 85], Hierarchically Semiseparable (HSS) [29], $\mathcal{H}$ [50], and $\mathcal{H}^2$ [48] matrices. These formats can reduce the cost for matrix multiplication, decomposition, and inversion from $\mathcal{O}(n^3)$ down to $\mathcal{O}(n)$ flops. Since these matrix operations are the building blocks of eigenvalue solvers, they can be used to significantly reduce the cost of eigenvalue computation.

1.2 Related Works

In recent decades, there have been new algorithms that utilize structured low-rank formats to accelerate the computation of all eigenvalues. QR decomposition of quasi-separable matrices has been studied extensively to accelerate the QR algorithm, allowing the computation of all eigenvalues of companion matrices in $\mathcal{O}(n^2)$ flops [17, 35, 30, 84]. For more general structured low-rank matrices, the QR decomposition of $\mathcal{H}$-matrices in [13] has been used to accelerate the QR algorithm for HODLR matrices, leading to a cost of $\mathcal{O}(n^2 (\log_2 n)^2)$ flops to compute all eigenvalues [65]. Further, a more robust QR decomposition of HODLR matrices described in [59] has been employed to compute spectral projector for a divide-and-conquer eigensolver of [68], resulting in a cost of $\mathcal{O}(n (\log_2 n)^3)$ flops to compute all eigenvalues along with the corresponding eigenvectors [81]. Another divide-and-conquer eigensolver described in [34] has also been adapted to HSS matrices to compute all eigenvalues in $\mathcal{O}(n^2)$ flops [27]. This approach was further accelerated in [86, 70] using the Fast Multipole Method (FMM) [44], reducing the cost down to $\mathcal{O}(n (\log_2 n)^2)$ flops to compute the full eigendecomposition of HSS matrices. Although these methods require significantly less computational cost than their dense counterparts, they are difficult to parallelize due to complex dependencies between the operations that arise during the computation. This often limits the level of parallelization to the memory-bound low-level BLAS/LAPACK kernels. In an attempt to utilize a higher level parallelization, the QR decomposition was studied with the BLR format in [55], which is much easier to parallelize than the hierarchical formats. However, they still reported limited parallel scalability due to the coarse granularity of the tasks within the algorithm.

On the other hand, there have been several studies that focus on computing a subset of eigenvalues of structured low-rank matrices. This is particularly important in some problem classes [53, 82, 33] where the eigenvalues of interest are only a small subset located far enough in the middle of the spectrum so it is unnecessary to compute all of the eigenvalues [67]. A projection method based on $\mathcal{H}$-matrix product and inversion has been presented in [49] that allows the computation of eigenvalues within a specified interval (a, b) in almost linear complexity. Cholesky decomposition of $\mathcal{H}$-matrices has also been used for preconditioned inverse iteration to compute an eigenvalue in $\mathcal{O}(n (\log_2 n)^2)$ flops [15]. Further, LDL decomposition of HODLR [14] and $\mathcal{H}^2$ [16] matrices have been studied with the bisection eigenvalue algorithm, allowing the computation of the k-th smallest eigenvalue in $\mathcal{O}(n \log_2(n) \log_2((b-a)/\epsilon_{ev}))$ flops, where $[a, b]$ is the bisection starting interval containing the target eigenvalue and ϵ_{ev} is the desired eigenvalue accuracy. The work in [88] further reduces this cost to $\mathcal{O}(n \log_2((b-a)/\epsilon_{ev}))$ by employing a fast generalized LDL decomposition of HSS matrices. Even though this is optimal in terms of computational cost, the application to 3D problem often leads to suboptimal performance due to the limitation of the HSS format. Moreover, to the best of our knowledge, at the time of writing, there has been no existing work that studies the parallelization of these methods in computing a single eigenvalue.

1.3 Objectives and Contributions

This thesis aims to accelerate the eigenvalue computation of structured low-rank matrices by presenting novel and fast matrix decomposition techniques. Our main contributions can be summarized as follows.

1. We present two new algorithms for the QR decomposition of BLR matrices. Our first algorithm extends the blocked Householder method in [43] to BLR matrices in order to obtain a fast QR decomposition with a cost of $\mathcal{O}(mn)$ flops. Our second algorithm further employs the idea of updating QR factorization to obtain a highly parallel QR decomposition with a cost of $\mathcal{O}(mn^{1.5})$ flops. We then apply our algorithm to accelerate eigenvalue computations with the QR algorithm.

2. We further present two new algorithms for the generalized LDL decomposition of structured low-rank matrices. Our first algorithm extends the existing method in [88] to distributed memory systems, resulting in a scalable generalized LDL decomposition of HSS matrices with a cost of $\mathcal{O}(n)$ flops. Our second algorithm extends the $\mathcal{H}^2$-ULV decomposition in [63] to obtain a fast generalized LDL factorization of $\mathcal{H}^2$-matrices that achieves a cost of $\mathcal{O}(n)$ flops even for 3D problems. We then employ these algorithms to solve the k-th eigenvalue problems efficiently using the bisection method.

3. We provide efficient parallel implementations of our proposed algorithms on shared and distributed memory systems.

4. We compare our implementations against several existing methods, including the state-of-the-art dense matrix factorization and eigensolver routines in LAPACK [8], ScaLAPACK [18] and ELPA [66]. Numerical results show that our methods achieve significantly faster computation time and higher parallel scalability.

5. We apply our proposed algorithm to efficiently solve a practical k-th eigenvalue problem arising from the electronic structure calculations of carbon nanomaterials.

1.4 Organization of Thesis

This thesis is written to serve as an introduction to eigenvalue problems and efficient computation of their numerical solutions using hierarchical low-rank approximation techniques. In Chapter 2, we first lay out the basic foundation of eigenvalue, eigenvector, and their characteristics, followed by matrix decomposition techniques that build up the traditional eigenvalue algorithms. In Chapter 3, we introduce the concept of hierarchical low-rank approximations and discuss their benefits in accelerating dense matrix computations.

Then in Chapter 4, we describe our proposed QR decomposition algorithms based on the BLR matrix format and their application to the QR algorithm for eigenvalue computations. In Chapter 5 we present another class of matrix decomposition called generalized LDL decomposition based on the HSS and $\mathcal{H}^2$-matrices formats along with their application to solving the k-th eigenvalue problem using the bisection method. Chapter 6 concludes this thesis along with some remarks regarding future works.

1.5 Notation

Here we introduce the mathematical notations that are used throughout the thesis. We mainly follow the Householder notation, such that scalar variables are denoted by Greek letters (e.g. α, β, λ), column vectors are denoted by lowercase Roman letters (e.g. v, w, x), and matrices are denoted by capital Roman letters (e.g. A, B, M). We also adopt the colon notation that is commonly used in MATLAB to represent submatrix. The complete reference is shown as follows.

flops	Floating-point operations
$\mathbb{Z}$	Space of integers
$\mathbb{R}$	Space of real numbers
$\mathbb{R}^n$	Space of real vectors with n elements
$\mathbb{R}^{m\times n}$	Space of real matrices with m rows and n columns
α, β, λ	Lowercase Greek letters denote scalar variables
$\text{sign}(\alpha)$	Sign function that returns 1 if $\alpha \geq 0$, otherwise returns -1
v, w, x	Lowercase Roman letters denote column vectors
v_i, $v(i)$	i-th element of the vector v
$v_{i:p}$, $v(i{:}p)$	The i to p-th elements of the vector v, where $p \geq i$
v^T	Transpose of vector v
$\|v\|_p$	p-norm of vector v
e_i	i-th unit vector
$\mathbf{0}_n$	Vector of zeros with n elements. n may be omitted if obvious
$span\{v_1, v_2, \ldots, v_n\}$	Space of vectors formed by the linear combinations of $v_1, v_2, \ldots, v_n$
A, B, M	Uppercase Roman letters denote matrices
$A_{i,j}$	The (i,j)-th sub-block of matrix A
$A_{l;i,j}$	The (i,j)-th hierarchical sub-block of matrix A at level l
$A_{l;i,+}$	The concatenation of all admissible blocks in the entire row i at level l
$A_{l;+,j}$	The concatenation of all admissible blocks in the entire column j at level l
$a_{i,j}$, $A(i,j)$	The element at the i-th row and j-th column of matrix A
$a_{i:p,j:q}$, $A(i{:}p, j{:}q)$	Submatrix of A taken from rows $[i,p]$ and columns $[j,q]$, where $p \geq i$ and $q \geq j$. Colon alone denotes full range
$\boldsymbol{a}_j$	The j-th column vector of matrix A (denoted in boldface)
$\tilde{A}$	The low-rank approximation of matrix A
A^T	Transpose of matrix A
A^{-1}	Inverse of matrix A
$\|A\|_p$	p-norm of matrix A
I_n	Identity matrix of size $n \times n$. n may be omitted if obvious
$\mathbf{0}$	Matrix of zeros with appropriate size
$diag(\delta_1, \ldots, \delta_n)$	Diagonal matrix with $\delta_1, \cdots, \delta_n$ on the main diagonal
$diag(A_1, A_2, \ldots, A_n)$	Block diagonal matrix with the matrices $A_1, A_2, \ldots, A_n$ on the main diagonal

Chapter 2

Eigenvalue Problem

The eigenvalue problem is one of the most important subjects in numerical linear algebra. Generally speaking, eigenvalues are particularly useful for two reasons: algorithmically, eigenvalue analysis can simplify the solutions of certain problems by reducing them into a collection of scalar problems; physically, eigenvalue can give insight into the behavior of evolving systems governed by linear equations in the long run [83]. For these reasons, the solution to eigenvalue problems plays a crucial role in solving a wide range of problems, such as

- dynamics of electromagnetic fields;
- electronic structure calculations;
- particle accelerator simulations;
- vibrations and buckling in mechanics, structural dynamics;
- and many more [58].

This chapter serves as an introduction to eigenvalue problems and the common ways to solve them. We begin by providing the definitions and some properties related to the eigenvalues and eigenvectors in Section 2.1. Then we explain the fundamentals of QR and LDL decomposition in Sections 2.2 and 2.3, which are widely used in modern eigenvalue algorithms. Finally, we introduce several well-known methods that have been commonly used to solve the eigenvalue problems in Section 2.4. The majority of the explanation within this chapter is based on the textbooks "Numerical Linear Algebra" by Trefethen and Bau [83] and "The Symmetric Eigenvalue Problem" by Parlett [71], which the author believes to contain intuitive and easy-to-understand elaborations. Therefore, we refer the reader to those references for more detailed discussions.

In this thesis, we focus on investigating the symmetric eigenvalue problem. Thus, from here onward we assume that the matrix $A \in \mathbb{R}^{n \times n}$ in the eigenvalue problems is symmetric.

2.1 Definitions

Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, the eigenvalue problem amounts to finding the solution of

$$Ax = \lambda x, \tag{2.1}$$

such that $\lambda \in \mathbb{R}$ is referred to as the *eigenvalue* of A and $x \in \mathbb{R}^n$ is the corresponding *eigenvector*. The set of all eigenvalues of A $\{\lambda_1, \lambda_2, ..., \lambda_n\}$ such that $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ is called the *spectrum* of A and it is usually denoted by $\Lambda(A)$. The pair (λ, x) composed of the eigenvalue λ and its corresponding eigenvector x is called an *eigenpair*. The eigenvalue problem where A has a structured low-rank property is called a *rank-structured eigenvalue problem*.

2.1.1 Rayleigh Quotient

The *Rayleigh quotient* of a vector $x \in \mathbb{R}^n$ is the scalar

$$\rho(x) = \frac{x^T Ax}{x^T x}. \tag{2.2}$$

One way to describe this formula is, given a vector x, what is the scalar $\hat{\lambda}$ that acts most like the corresponding eigenvalue of x in the sense of minimizing $\left\|Ax - \hat{\lambda}x\right\|_2$? This amounts to an $n \times 1$ least squares problem whose answer is $\rho(x)$ in Equation (2.2). This tells us that $\rho(x)$ is the natural eigenvalue estimate to choose if x is close to, but not necessarily equal to an eigenvector of A. If x is an eigenvector of A, then $\rho(x) = \lambda$ is the corresponding eigenvalue. This allows us to compute the corresponding eigenvalue of a given eigenvector.

2.1.2 Eigenvalue Decomposition

For any symmetric matrix A, there exists a factorization of the form

$$A = Z\Lambda Z^T, \tag{2.3}$$

where $\Lambda \in \mathbb{R}^{n\times n}$ is a diagonal matrix and $Z \in \mathbb{R}^{n\times n}$ is an orthogonal matrix [71, Fact 1.4]. This factorization is known as the *eigenvalue decomposition*. Let $\boldsymbol{z}_j \in \mathbb{R}^n$ be the j-th column vector of Z. Then, Equation (2.3) can be rewritten as

$$AZ = Z\Lambda$$

$$A\left[\begin{array}{c|c|c|c} & & & \\ \boldsymbol{z}_1 & \boldsymbol{z}_2 & \cdots & \boldsymbol{z}_n \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ \boldsymbol{z}_1 & \boldsymbol{z}_2 & \cdots & \boldsymbol{z}_n \\ & & & \end{array}\right] \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}.$$

It is clear that the j-th diagonal entry of Λ is the j-th eigenvalue of A, the j-th column of Z is the corresponding eigenvector. The columns of Z form an orthonormal set of the eigenvectors of A.

2.1.3 Characteristic Polynomial

From Equation (2.1), we can derive the *characteristic polynomial* of A, which is the degree n polynomial defined by

$$p_A(z) = \det(A - zI), \tag{2.4}$$

where $\det(A - zI)$ denotes the *determinant* of the matrix $A - zI$. It follows that λ is an eigenvalue of A *if and only if* $p_A(\lambda) = 0$ [83, Theorem 24.1]. Thus, finding the eigenvalues of A can also be seen as finding the roots of its characteristic polynomial. Therefore, Equation (2.4) can also be written as

$$p_A(z) = (z - \lambda_1)(z - \lambda_2) \cdots (z - \lambda_n), \tag{2.5}$$

where each λ_j $(j = 1, 2, \ldots, n)$ corresponds to an eigenvalue of A. In general, eigenvalues are not necessarily distinct, i.e. an eigenvalue might appear more than once in Equation (2.5). The *algebraic multiplicity* of an eigenvalue λ corresponds to its multiplicity as a root of p_A. An eigenvalue is *simple* if its algebraic multiplicity is 1.

2.1.4 Similarity Transformations

Let $X \in \mathbb{R}^{n \times n}$ be an invertible matrix. The transformation XAX^{-1} is called the *similarity transformation* of A. Two matrices A and B are *similar* if there exists a similarity transformation from one to the other such that $B = XAX^{-1}$. If two matrices are similar, they have the same eigenvalues, characteristic polynomial, and algebraic multiplicity [83, Theorem 24.3]. In other words, similarity transformation preserves the spectrum of the matrix, i.e.

$$\Lambda(A) = \Lambda(XAX^{-1}). \tag{2.6}$$

However, eigenvectors are generally not preserved. If (λ, z) is an eigenpair of A, then (λ, Xz) is the corresponding eigenpair of XAX^{-1}. This is easy to check since

$$\left(XAX^{-1}\right) Xz = XAz = \lambda Xz.$$

2.1.5 Determinant and Trace

Eigenvalues are also related to some important properties of a matrix. The determinant of A is equal to the product of its eigenvalues, i.e.

$$\det(A) = \prod_{j=1}^{n} \lambda_j. \tag{2.7}$$

The *trace* of A, which corresponds to the sum of its diagonal elements, is equal to the sum of its eigenvalues, that is

$$\operatorname{tr}(A) = \sum_{j=1}^{n} a_{j,j} = \sum_{j=1}^{n} \lambda_j. \tag{2.8}$$

2.2 QR Decomposition

QR Decomposition is one of the fundamental operations in linear algebra. It is an essential component to solve many problems in scientific computing. Notable examples are computing the solution of a system of linear equations, low-rank approximations, solving the least squares problems, and also eigenvalue problems. More importantly, QR decomposition provides the basis for one of the most widely used eigenvalue solvers nowadays, the QR algorithm, which we will explain in Section 2.4.5. In this section, we introduce the fundamentals of QR decomposition along with some well-established methods that are commonly used.

Given a matrix $M \in \mathbb{R}^{m \times n}$, the *full QR* decomposition produces a factorization of the form

$$M = QR, \tag{2.9}$$

where $Q \in \mathbb{R}^{m \times m}$ is an orthogonal matrix ($QQ^T = Q^TQ = I$) and $R \in \mathbb{R}^{m \times n}$ is an upper triangular matrix. When M is a tall-skinny matrix, i.e. $m > n$, the *reduced QR* decomposition produces compact factors $Q \in \mathbb{R}^{m \times n}$ and $R \in \mathbb{R}^{n \times n}$. Graphical illustrations are shown in Figures 2.1 and 2.2.

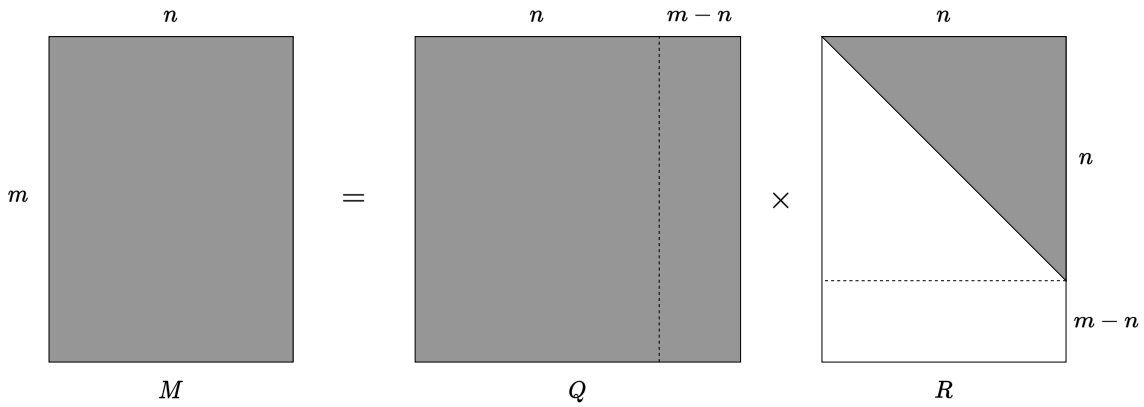


Figure 2.1: Full QR decomposition

Let $\boldsymbol{m}_j \in \mathbb{R}^m$ be the j-th column vector of M. Then we can rewrite Equation (2.9) as

$$\left[\begin{array}{c|c|c|c} & & & \\ \boldsymbol{m}_1 & \boldsymbol{m}_2 & \cdots & \boldsymbol{m}_n \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ \boldsymbol{q}_1 & \boldsymbol{q}_2 & \cdots & \boldsymbol{q}_n \\ & & & \end{array}\right] \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ & r_{2,2} & \cdots & r_{2,n} \\ & & \ddots & \vdots \\ & & & r_{n,n} \end{bmatrix}, \tag{2.10}$$

where $\boldsymbol{q}_j \in \mathbb{R}^m$ is the j-th orthonormal column of Q and $r_{i,j} \in \mathbb{R}$ is the (i,j)-th element of R. Equation (2.10) shows that the j-th column of M is represented as a linear combination of the first j columns of Q using the elements of R as the coefficients, that is

$$\boldsymbol{m}_j = r_{1,j}\,\boldsymbol{q}_1 + r_{2,j}\,\boldsymbol{q}_2 + \cdots + r_{j,j}\,\boldsymbol{q}_j, \tag{2.11}$$

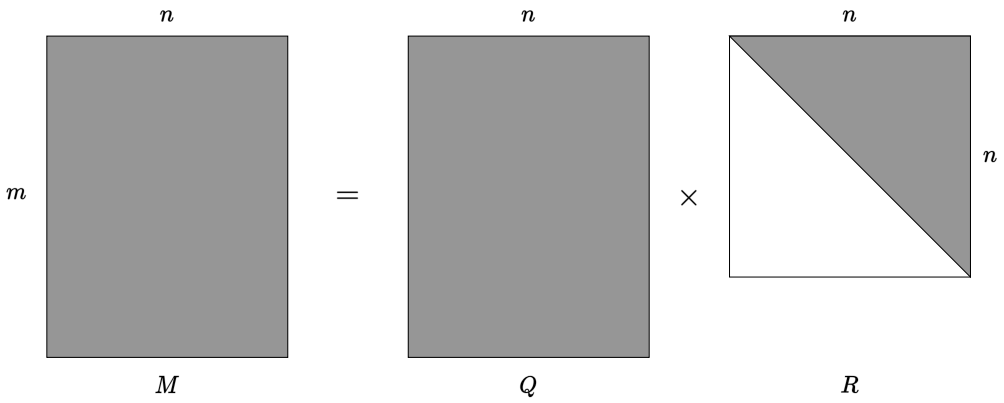


Figure 2.2: Reduced QR decomposition

for $j = 1, 2, \ldots, n$. This means that the first j columns of Q form an orthogonal basis for the first j-th columns of M. In other words, the QR decomposition aims to form the orthogonal basis Q that spans the successive column spaces of M.

In general, there are two common ways to compute the QR decomposition: *triangular orthogonalization* and *orthogonal triangularization*. The former corresponds to the Gram-Schmidt orthogonalization, where the input matrix M is transformed into an orthogonal matrix Q by multiplication with a set of triangular matrices. The latter corresponds to the Householder reflection and Givens rotation, where the input matrix M is transformed into an upper-triangular matrix R by multiplication with a set of orthogonal matrices. Typically, the method of choice depends on the condition number of M and how the factorization is going to be used. The Gram-Schmidt method directly produces the orthogonal factor Q and it is usually used when M is known to be well-conditioned since it suffers from loss of orthogonality when the condition number of M is high. On the other hand, Householder and Givens methods produce a set of orthogonal matrices that need to be multiplied further to obtain Q, but they are robust to ill-conditioning. The Householder method is typically the method of choice when $m \gg n$ and Q does not need to be explicitly computed [47].

2.2.1 Classical Gram-Schmidt QR

The Gram-Schmidt process was first introduced back in 1907 by Erhard Schmidt. It is one of the commonly used methods to produce an orthonormal basis for a set of linearly independent vectors. This method is used to compute the QR decomposition of the matrix M by forming the orthonormal basis for its columns. This basis corresponds to the orthogonal matrix Q that is obtained by multiplying M from the right with a sequence of elementary triangular matrices, which can be written as

$$M \underbrace{R_1 R_2 \cdots R_n}_{R^{-1}} = Q. \tag{2.12}$$

Because the product $R = R_n^{-1} \cdots R_2^{-1} R_1^{-1}$ is also an upper triangular matrix, this results in the reduced QR decomposition of M. This technique is also known as the *Gram-Schmidt triangular orthogonalization*.

The Gram-Schmidt QR proceeds directly from Equation (2.11), where it computes the successive orthonormal vectors $\boldsymbol{q}_j$ using the following formula:

$$\begin{aligned}
\boldsymbol{q}_1 &= \frac{\boldsymbol{m}_1}{r_{1,1}}, \\
\boldsymbol{q}_2 &= \frac{\boldsymbol{m}_2 - r_{1,2}\boldsymbol{q}_1}{r_{2,2}}, \\
\boldsymbol{q}_3 &= \frac{\boldsymbol{m}_3 - r_{1,3}\boldsymbol{q}_1 - r_{2,3}\boldsymbol{q}_2}{r_{3,3}}, \\
&\vdots \\
\boldsymbol{q}_n &= \frac{\boldsymbol{m}_n - \sum_{i=1}^{n-1} r_{i,n}\boldsymbol{q}_i}{r_{n,n}},
\end{aligned} \tag{2.13}$$

such that the entries $r_{i,j}$ are determined by

$$r_{i,j} = \begin{cases} \boldsymbol{q}_i^T \boldsymbol{m}_j, & (i \neq j) \\ \left\| \boldsymbol{m}_j - \sum_{k=1}^{j-1} r_{k,j}\boldsymbol{q}_k \right\|, & (i = j) \end{cases}.$$

This method is referred to as the *classical* Gram-Schmidt algorithm. It produces a reduced QR decomposition and the steps are summarized in Algorithm 1.

Algorithm 1: Classical Gram-Schmidt QR decomposition

Input: $M \in \mathbb{R}^{m \times n}$
Output: orthogonal $Q \in \mathbb{R}^{m \times n}$, upper-triangular $R \in \mathbb{R}^{n \times n}$

```
for j = 1 to n do
    q_{:,j} = m_{:,j}
    for i = 1 to j − 1 do
        r_{i,j} = q_{:,i}^T m_{:,j}
        q_{:,j} ← q_{:,j} − q_{:,i} r_{i,j}
    end
    r_{j,j} = ‖q_{:,j}‖_2
    q_{:,j} ← q_{:,j} / r_{j,j}
end
```

2.2.2 Modified Gram-Schmidt QR

The sequence of calculation in Algorithm 1 is known to be numerically unstable when performed on a computer due to rounding errors [83]. As a remedy, a modified version has been proposed where

the sequence of calculation in Equation (2.13) is reordered as follows.

$$
\begin{aligned}
\boldsymbol{q}_1 &= \frac{\boldsymbol{m}_1^{(1)}}{r_{1,1}};\\
\boldsymbol{m}_2^{(2)} &= \boldsymbol{m}_2^{(1)} - r_{1,2}\boldsymbol{q}_1,\ \boldsymbol{m}_3^{(2)} = \boldsymbol{m}_3^{(1)} - r_{1,3}\boldsymbol{q}_1,\ \cdots,\ \boldsymbol{m}_n^{(2)} = \boldsymbol{m}_n^{(1)} - r_{1,n}\boldsymbol{q}_1;\\
\boldsymbol{q}_2 &= \frac{\boldsymbol{m}_2^{(2)}}{r_{2,2}};\\
\boldsymbol{m}_3^{(3)} &= \boldsymbol{m}_3^{(2)} - r_{2,3}\boldsymbol{q}_2,\ \boldsymbol{m}_4^{(3)} = \boldsymbol{m}_4^{(2)} - r_{2,4}\boldsymbol{q}_2,\ \cdots,\ \boldsymbol{m}_n^{(3)} = \boldsymbol{m}_n^{(2)} - r_{2,n}\boldsymbol{q}_2;\\
&\vdots\\
\boldsymbol{m}_n^{(n)} &= \boldsymbol{m}_n^{(n-1)} - r_{n-1,n}\boldsymbol{q}_{n-1};\\
\boldsymbol{q}_n &= \frac{\boldsymbol{m}_n^{(n)}}{r_{n,n}},
\end{aligned}
\tag{2.14}
$$

such that the entries $r_{i,j}$ are determined by

$$
r_{i,j} = \begin{cases} \boldsymbol{q}_i^T \boldsymbol{m}_j^{(i)}, & (i \neq j) \\ \left\| \boldsymbol{q}_j^{(j)} \right\|, & (i = j) \end{cases}.
$$

Although this is mathematically equivalent to Equation (2.13), this order of computation introduces smaller errors on a computer. In practice, this approach is commonly used instead of the classical one. Algorithm 2 summarizes the steps, which requires $\mathcal{O}(mn^2)$ flops.

Algorithm 2: Modified Gram-Schmidt QR decomposition

Input: $M \in \mathbb{R}^{m \times n}$
Output: orthogonal $Q \in \mathbb{R}^{m \times n}$, upper-triangular $R \in \mathbb{R}^{n \times n}$

```
for i = 1 to n do
    r_{i,i} = ||m_{:,i}||_2
    q_{:,i} = m_{:,i} / r_{i,i}
    for j = i + 1 to n do
        r_{i,j} = q_{:,i}^T m_{:,j}
        m_{:,j} ← m_{:,j} − q_{:,i} r_{i,j}
    end
end
```

2.2.3 Householder QR

The Householder QR decomposition is based on the Householder *reflector*, which is a transformation that reflects a given vector $x \in \mathbb{R}^m$ across a plane that intersects the origin. The Householder reflection of x over a plane orthogonal to a normal vector v is expressed as the symmetric orthonormal Householder matrix

$$
H = I - \beta v v^T, \tag{2.15}
$$

such that $\beta = 2$ if v is a unit vector, otherwise $\beta = \frac{2}{v^T v}$. Then by setting v as

$$v = \begin{pmatrix} 1 \\ \frac{x_{2:m}}{\xi}, \end{pmatrix} \tag{2.16}$$

such that $\xi = x_1 + \text{sign}(x_1) \|x\|_2$ and $x_1 \in \mathbb{R}$ is the first element of x, we obtain a transformation that annihilates all but x_1. Therefore, we can repeatedly obtain and apply similar transformations to the columns of M in order to transform it into an upper triangular matrix. This process is called *triangularization*. Overall, this process can be viewed as multiplication from the left with a sequence of orthogonal matrices, that is

$$\underbrace{H_n \cdots H_2 H_1}_{Q^{-1}} M = R. \tag{2.17}$$

The product $Q = H_1^T H_2^T \cdots H_n^T$ is orthogonal too. This method produces a full QR decomposition and it is known as the Householder *orthogonal triangularization*. Figure 2.3 shows a simple example for a 4×3 matrix. In the matrices in Figure 2.3 and some other examples that we provide throughout the thesis, the symbol $\times$ denotes an entry that is not necessarily zero and the boldfaced symbol denotes an entry that has just been modified.

$$\underset{M}{\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}} \overset{H_1}{\rightarrow} \underset{H_1 M}{\begin{bmatrix} \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \times & \times \\ \mathbf{0} & \times & \times \\ \mathbf{0} & \times & \times \end{bmatrix}} \overset{H_2}{\rightarrow} \underset{H_2 H_1 M}{\begin{bmatrix} \times & \times & \times \\ 0 & \mathbf{\times} & \mathbf{\times} \\ 0 & \mathbf{0} & \times \\ 0 & \mathbf{0} & \times \end{bmatrix}} \overset{H_3}{\rightarrow} \underset{H_3 H_2 H_1 M}{\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \mathbf{\times} \\ 0 & 0 & \mathbf{0} \end{bmatrix}} = R.$$

Figure 2.3: Householder triangularization of a 4×3 matrix

Let us define the function $house(x) = [v, \beta]$ as the computation of the v and β in Equations (2.15) and (2.16). The Householder QR decomposition is shown in Algorithm 3 and it requires $\mathcal{O}(mn^2)$ flops. Upon completion, the upper triangular part of M is overwritten with the upper triangular

Algorithm 3: Householder QR decomposition

Input: $M \in \mathbb{R}^{m \times n}$
Output: Householder vectors $v_j \in \mathbb{R}^{m-j+1}$ for $j = 1, 2, \ldots, n$ and upper-triangular $R \in \mathbb{R}^{m \times n}$

```
for j = 1 to n do
    [v, β] = house(M(j:m, j))
    M(j:m, j:n) ← (I − βvv^T)M(j:m, j:n)
    if j < m then
        M(j + 1:m, j) ← v(2:m − j + 1)
    end
end
```

factor R. Note that since the resulting Householder vectors v_j always have 1 as their first element,

we can completely store them in the lower triangular part of M such that $v_j(2:m-j+1)$ is stored in $M(j+1{:}m, j)$. To clarify how M is overwritten, Equation (2.18) shows the elements of $M \in \mathbb{R}^{4\times 3}$ upon the completion of Algorithm 3.

$$M = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ v_1(2) & r_{2,2} & r_{2,3} \\ v_1(3) & v_2(2) & r_{3,3} \\ v_1(4) & v_2(3) & v_3(2) \end{bmatrix}. \tag{2.18}$$

Also note that there is no need to store the resulting β from each iteration since we can easily retrieve them from the Householder vectors using the equation

$$\beta_j = \frac{2}{1 + \|M(j+1{:}m, j)\|}. \tag{2.19}$$

In contrary to Algorithm 2, Algorithm 3 does not explicitly form the orthogonal factor Q. When Q needs to be explicitly formed, we can construct it by the following steps. Let $u_j \in \mathbb{R}^{m-j+1}$ be

$$u_j = \begin{pmatrix} 1 \\ M(j+1{:}m, j) \end{pmatrix}. \tag{2.20}$$

Then for each u_j, we construct the orthogonal matrix corresponding to the Householder reflection

$$H_j = \begin{bmatrix} I_{j-1} & \mathbf{0} \\ \mathbf{0} & I - \beta_j u_j u_j^T \end{bmatrix}. \tag{2.21}$$

Finally, we compute Q from

$$Q = \prod_{j=1}^{n} H_j^T.$$

This explicit formation of Q requires an additional cost of $\mathcal{O}(m^2 n + n^3)$ flops.

Schreiber and van Loan in [75] has introduced another way of accumulating Householder transformations that is tailored to better utilize modern CPU architectures, known as the *compact WY representation*. Using their method, the orthogonal factor Q is stored implicitly as

$$Q = I_m - YTY^T,$$

where $Y \in \mathbb{R}^{m\times n}$ is a lower trapezoidal matrix containing the householder vectors and $T \in \mathbb{R}^{n\times n}$ is an upper triangular matrix that is additionally generated during the QR decomposition.

The QR decomposition that contains the generation of Y and T matrices is shown in Algorithm 4. Note that upon completion R still overwrites the upper triangular part of M and Y overwrites the remaining lower trapezoidal part, but additional storage is needed to store T. This algorithm requires $\mathcal{O}(mn^2 + n^3)$ flops. The n^3 term comes from the generation of the T matrix, which can be further removed by employing an inner blocking strategy described in [73, 22].

Algorithm 4: Householder QR decomposition with compact WY representation

Input: $M \in \mathbb{R}^{m \times n}$

Output: $M = \{Y \backslash R\}$, $Y \in \mathbb{R}^{m \times n}$ is lower trapezoidal, $R \in \mathbb{R}^{m \times n}$ and $T \in \mathbb{R}^{n \times n}$ are upper triangular

for $j = 1$ **to** n **do**
  $[v, \beta] = house(M(j{:}m, j))$
  $M(j{:}m, j{:}n) \leftarrow (I - \beta v v^T) M(j{:}m, j{:}n)$
  if $j = 1$ **then**
    $Y = \begin{bmatrix} v \end{bmatrix}$
    $T = \begin{bmatrix} \beta \end{bmatrix}$
  end
  else
    $y = \begin{bmatrix} \mathbf{0}_{j-1} \\ v \end{bmatrix}$
    $z = -\beta \cdot T Y^T y$
    $Y \leftarrow \begin{bmatrix} Y & y \end{bmatrix}$
    $T \leftarrow \begin{bmatrix} T & z \\ \mathbf{0} & \beta \end{bmatrix}$
  end
end
Overwrite zeros below diagonal of M with lower trapezoidal part of Y

2.2.4 Givens QR

Givens rotation is a transformation to selectively introduce zero in a vector. This idea was first introduced by Wallace Givens in [39]. Unlike Householder reflection that zeroes everything but the first element of a vector, a Givens rotation only affects 2 elements of the vector, such that one of them will become zero.

A Givens rotation is essentially a rotation of θ radians in the (i, k) coordinate plane, which can be expressed as the orthogonal matrix

$$G(i,k,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & i & & k & & \end{matrix}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. For example, let $x \in \mathbb{R}^m$. By applying $G(i, k, \theta)^T$ $(1 \leq i < k \leq m)$ to x, that is

$$y = G(i,k,\theta)^T x,$$

we have the transformed vector

$$y_j = \begin{cases} cx_i - sx_k, & (j = i), \\ sx_i + cx_k, & (j = k), \\ x_j, & (j \neq i \text{ and } j \neq k). \end{cases}$$

Therefore, we can make the entry y_k zero by setting the variables

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}}, \quad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}}. \tag{2.22}$$

With this way of introducing zero to one element at a time, we can repeatedly apply Givens rotation to the columns of M in order to transform it to form an upper triangular matrix, thus producing a full QR decomposition. An example for $M \in \mathbb{R}^{4\times 3}$ is shown in Figure 2.4.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \overset{G(3,4)}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \mathbf{0} & \times & \times \end{bmatrix} \overset{G(2,3)}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \mathbf{0} & \times & \times \\ 0 & \times & \times \end{bmatrix} \overset{G(1,2)}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ \mathbf{0} & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \overset{G(3,4)}{\rightarrow}$$
$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \mathbf{0} & \times \end{bmatrix} \overset{G(2,3)}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \mathbf{0} & \times \\ 0 & 0 & \times \end{bmatrix} \overset{G(3,4)}{\rightarrow} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \mathbf{0} \end{bmatrix} = R.$$

Figure 2.4: Orthogonal triangularization using Givens rotation of a 4×3 matrix

Let us define the function $[c, s] = givens(x_i, x_k)$ to be the calculation of c and s from Equation (2.22). Algorithm 5 shows the QR decomposition using Givens rotation, which requires a cost of $\mathcal{O}(mn^2)$ flops. This method actually requires more flops compared to the Householder method due to the larger number of transformations. However, Givens rotation provides a significant advantage over Householder reflection when many elements of M are already zero, notably when M is a large sparse matrix [38].

Algorithm 5: Givens QR decomposition

Input: $M \in \mathbb{R}^{m\times n}$
Output: $M = \{\backslash R\}$, R is upper triangular

```
for j = 1 to n do
    for i = m downto j + 1 do
        [c, s] = givens(M(i − 1, j), M(i, j))
        M(i − 1 : i, j : n) ← [ c  s ; −s  c ]^T M(i − 1 : i, j : n)
    end
end
```

Similar to the Householder method, Givens QR does not explicitly compute the orthogonal factor Q. In case when Q needs to be explicitly formed, it can be constructed from the Givens rotations

that were used to form R. A compact way to represent Givens rotation has been shown in [78], where one rotation is associated with a single floating point number. These numbers can then be stored in the lower triangular part of M and used to recover the Givens rotations when needed.

2.3 LDL Decomposition

The LDL decomposition factorizes a symmetric matrix $A \in \mathbb{R}^{n\times n}$ to a form

$$A = LDL^T, \tag{2.23}$$

where $L \in \mathbb{R}^{n\times n}$ is a unit lower-triangular matrix (i.e. lower triangular matrix whose diagonal entries are all 1), and $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix. This matrix decomposition is notably important for the bisection-based eigenvalue solver, which we will cover in Section 2.4.7. This factorization is based on the *Gaussian elimination* and it is closely related to the *Cholesky factorization*. Here we start our explanation by providing a brief overview of the Gaussian elimination, followed by a description of the Cholesky decomposition. Then finally we explain how to compute the LDL decomposition.

2.3.1 Gaussian Elimination

Gaussian elimination is one of the basic techniques that are typically taught in linear algebra courses. It is the simplest method to solve a system of linear equations by hand. It also has become the standard method to solve them on a computer. The main idea is to transform a full linear system into an upper triangular one. It is quite similar to the Householder QR described in Section 2.2.3, in the sense that both methods transform a full matrix into an upper triangular matrix. However, the transformations that are used in Gaussian elimination are not orthogonal matrices, but unit lower triangular matrices.

Given a square matrix $M \in \mathbb{R}^{n\times n}$ (not necessarily symmetric), the Gaussian elimination amounts to multiplication from the left with a sequence of lower triangular matrices, that is,

$$\underbrace{L_{n-1}\cdots L_2 L_1}_{L^{-1}} M = U, \tag{2.24}$$

where $L_k \in \mathbb{R}^{n\times n}$ for $k = 1, 2, \ldots, n-1$ are unit lower triangular matrices and $U \in \mathbb{R}^{n\times n}$ is an upper triangular matrix. In other words, this process performs a *triangular triangularization*. Setting $L = L_1^{-1}L_2^{-1}\cdots L_{n-1}^{-1}$ leads to the LU decomposition

$$M = LU, \tag{2.25}$$

as illustrated in Figure 2.5. The elimination at the k-th step, which corresponds to the matrix L_k, introduces zeros below the diagonal in column k by subtracting multiples of row k from rows $k+1, k+2, \ldots, n$. An example of this process for a 4×4 matrix is given in Figure 2.6.

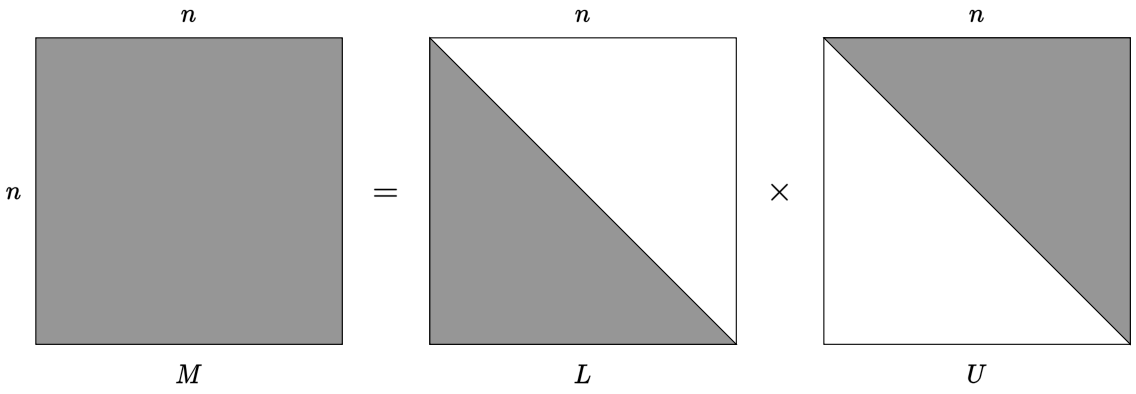


Figure 2.5: LU decomposition

$$\underset{M}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \overset{L_1}{\rightarrow} \underset{L_1M}{\begin{bmatrix} \times & \times & \times & \times \\ \mathbf{0} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \end{bmatrix}} \overset{L_2}{\rightarrow} \underset{L_2L_1M}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \mathbf{0} & \mathbf{\times} & \mathbf{\times} \\ 0 & \mathbf{0} & \mathbf{\times} & \mathbf{\times} \end{bmatrix}} \overset{L_3}{\rightarrow} \underset{L_3L_2L_1M}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \mathbf{0} & \mathbf{\times} \end{bmatrix}} = U.$$

Figure 2.6: Gaussian elimination of a 4×4 matrix

Let us see how the transformation L_k is chosen at each step. Let $\boldsymbol{m}_k \in \mathbb{R}^n$ denotes the k-th column of the matrix at the beginning of step k. Therefore, L_k must be chosen so that

$$\underset{\boldsymbol{m}_k}{\begin{bmatrix} m_{1,k} \\ m_{2,k} \\ \vdots \\ m_{k,k} \\ m_{k+1,k} \\ \vdots \\ m_{n,k} \end{bmatrix}} \xrightarrow{L_k} \underset{L_k\boldsymbol{m}_k}{\begin{bmatrix} m_{1,k} \\ m_{2,k} \\ \vdots \\ m_{k,k} \\ 0 \\ \vdots \\ 0 \end{bmatrix}}.$$

Therefore, we want to subtract $\ell_{j,k}$ times row k from row j such that

$$\ell_{j,k} = \frac{m_{j,k}}{m_{k,k}}, \tag{2.26}$$

for $k < j \leq n$. Thus, the matrix L_k is written as

$$L_k = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -\ell_{k+1,k} & 1 & \\ & & \vdots & & \ddots \\ & & -\ell_{n,k} & & 1 \end{bmatrix}. \tag{2.27}$$

Moreover, we can obtain the inverse of L_k by simply negating its subdiagonal entries, i.e.

$$L_k^{-1} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & \ell_{k+1,k} & 1 & \\ & & \vdots & & \ddots \\ & & \ell_{n,k} & & 1 \end{bmatrix}. \tag{2.28}$$

Further, the unit lower-triangular matrix $L = L_1^{-1} L_2^{-1} \cdots L_n^{-1}$ can be formed by collecting the entries $\ell_{j,k}$ into the appropriate places, as shown in Equation (2.29) below.

$$L = L_1^{-1} L_2^{-2} \cdots L_{n-1}^{-1} = \begin{bmatrix} 1 & & & & \\ \ell_{2,1} & 1 & & & \\ \ell_{3,1} & \ell_{3,2} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{n,1} & \ell_{n,2} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}. \tag{2.29}$$

Note that in practice the matrices L_k are never formed and multiplied explicitly. Instead, the entries $\ell_{j,k}$ are calculated and stored directly into $L_{j,k}$. The transformation L_k are then applied implicitly. The steps are summarized in Algorithm 6, which requires $\mathcal{O}(n^3)$ flops.

Algorithm 6: Gaussian elimination

Input: $M \in \mathbb{R}^{n \times n}$
Output: $L, U \in \mathbb{R}^{n \times n}$ where L is unit lower triangular, U is upper triangular

$U = M$, $L = I$
for $k = 1$ **to** $n - 1$ **do**
  for $j = k + 1$ **to** n **do**
    $L(j,k) \leftarrow \dfrac{U(j,k)}{U(k,k)}$
    $U(j, k{:}n) \leftarrow U(j, k{:}n) - L(j,k) U(k, k{:}n)$
  end
end

2.3.2 Cholesky Decomposition

If the matrix $M \in \mathbb{R}^{n \times n}$ is symmetric, then the Gaussian elimination can be performed in about 2 times faster than the general matrices case using the Cholesky decomposition, a variant of Gaussian elimination that operates on both left and right of the matrix at once to preserve symmetry. Additionally, if M is *positive definite*, i.e. $x^T M x > 0$ for all vectors $x \in \mathbb{R}^n$, it has a unique Cholesky decomposition [83, Theorem 23.1].

We start our explanation by considering the Gaussian elimination applied on a symmetric positive definite matrix with 1 in its top-left entry

$$M = \begin{bmatrix} m_{1,1} & m_{1,2:n} \\ m_{2:n,1} & m_{2:n,2:n} \end{bmatrix} = \begin{bmatrix} 1 & w^T \\ w & B \end{bmatrix}. \tag{2.30}$$

The first step is to introduce zero below the first diagonal element, which amounts to applying the lower triangular matrix from the left

$$\underset{M}{\begin{bmatrix} 1 & w^T \\ w & B \end{bmatrix}} = \underset{L_1^{-1}}{\begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix}} \underset{L_1 M}{\begin{bmatrix} 1 & w^T \\ 0 & B - ww^T \end{bmatrix}}. \tag{2.31}$$

The rightmost term of Equation (2.31) shows that zeros have been introduced to the first column by subtracting multiples of the first row from the subsequent rows. From here, instead of immediately introducing zeros to the second column, we first introduce zeros to the first row in order to maintain symmetry. This is easily done by applying L_1^T from the right, which gives us

$$\underset{M}{\begin{bmatrix} 1 & w^T \\ w & B \end{bmatrix}} = \underset{L_1^{-1}}{\begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix}} \underset{L_1 M L_1^T}{\begin{bmatrix} 1 & 0 \\ 0 & B - ww^T \end{bmatrix}} \underset{L_1^{-T}}{\begin{bmatrix} 1 & w^T \\ 0 & I \end{bmatrix}}. \tag{2.32}$$

The Cholesky decomposition repeats this process of introducing zeros to the column and row of M symmetrically until it is reduced to an identity matrix.

Let us generalize the process to a symmetric positive definite matrix as in Equation (2.30) but with any $m_{1,1} > 0$. Let $\alpha = m_{1,1}$ and $\beta = \sqrt{\alpha}$. The process in Equation (2.32) is generalized by adjusting some elements of L_1 by a factor of β so that

$$\underset{M}{\begin{bmatrix} \alpha & w^T \\ w & B \end{bmatrix}} = \underset{L_1^{-1}}{\begin{bmatrix} \beta & 0 \\ w/\beta & I \end{bmatrix}} \underset{L_1 M L_1^T}{\begin{bmatrix} 1 & 0 \\ 0 & B - (ww^T/\alpha) \end{bmatrix}} \underset{L_1^{-T}}{\begin{bmatrix} \beta & w^T/\beta \\ 0 & I \end{bmatrix}}. \tag{2.33}$$

The same process is then repeated to the bottom-right trailing submatrix $B - (ww^T/\alpha)$ until we reach the bottom-right corner, eventually reducing M into the identity matrix

$$M = \underbrace{L_1^{-1} L_2^{-1} \cdots L_n^{-1}}_{L} I \underbrace{L_1^{-T} L_2^{-T} \cdots L_n^{-T}}_{L^T}. \tag{2.34}$$

This results in the Cholesky decomposition of the form

$$M = LL^T, \tag{2.35}$$

such that $L \in \mathbb{R}^{n \times n}$ is a lower triangular matrix with positive diagonal entries. Notice that in contrast to the factorization in Equation (2.25, the matrix L that is produced here is not necessarily unit lower triangular. The steps for Cholesky decomposition are summarized in Algorithm 7, which involves $\mathcal{O}(n^3)$ flops. Note that due to symmetry, the typical implementation of Cholesky factorization only operates on half of the matrix, i.e. either only the lower triangular part or only the upper triangular part. This allows half of the arithmetic operations to be avoided, in contrast to the Gaussian elimination in Algorithm 6. Here we choose to operate on the lower triangular portion

Algorithm 7: Cholesky decomposition

Input: Symmetric positive definite matrix $M \in \mathbb{R}^{n \times n}$
Output: Lower triangular matrix $L \in \mathbb{R}^{n \times n}$

$L = M$
for $k = 1$ **to** n **do**
  for $i = k + 1$ **to** n **do**
    $L(i{:}n, i) \leftarrow L(i{:}n, i) - L(i{:}n, k)\dfrac{L(i,k)}{L(k,k)}$
  end
  $L(k{:}n, k) \leftarrow \dfrac{L(k{:}n, k)}{\sqrt{L(k,k)}}$
end

of the matrix so we can easily relate Algorithm 7 to the LDL decomposition that will be covered in the next section.

It is also important to note that the process described in Equation (2.33) only works if the top-left entry of the trailing submatrices, e.g. $B - (ww^T/\alpha)$, is positive since otherwise the whole algorithm would break down. However, since we assume M to be positive definite, then it is guaranteed that all trailing submatrices that appear during the Cholesky decomposition are positive definite, meaning that their top-left entry is always positive too [83, p. 174].

2.3.3 LDL Decomposition - Algorithm

Since we have the Gaussian elimination and Cholesky decomposition in place, we can easily introduce the LDL decomposition. Let us consider the basic step of Cholesky decomposition shown in Equation (2.33). The LDL decomposition also uses the same step except that it does not take a square root of the top-left entry α and instead performs the following

$$\underbrace{\begin{bmatrix} \alpha & w^T \\ w & B \end{bmatrix}}_{M} = \underbrace{\begin{bmatrix} 1 & 0 \\ w/\alpha & I \end{bmatrix}}_{L_1^{-1}} \underbrace{\begin{bmatrix} \alpha & 0 \\ 0 & B - (ww^T/\alpha) \end{bmatrix}}_{L_1 M L_1^T} \underbrace{\begin{bmatrix} 1 & w^T/\alpha \\ 0 & I \end{bmatrix}}_{L_1^{-T}}. \tag{2.36}$$

For this reason, the LDL decomposition is sometimes called *square-root-free Cholesky decomposition*. Note that now the top-left entry of $L_1 M L_1^T$ is not 1 anymore but α. The LDL decomposition repeats this process to the trailing bottom-right submatrix until the bottom-right corner is reached, reducing M into a diagonal matrix instead of identity matrix:

$$M = \underbrace{L_1^{-1} L_2^{-1} \cdots L_n^{-1}}_{L} D \underbrace{L_1^{-T} L_2^{-T} \cdots L_n^{-T}}_{L^T}. \tag{2.37}$$

This gives us the factorization as illustrated in Figure 2.7. The LDL decomposition is formulated in Algorithm 8, where a cost of $\mathcal{O}(n^3)$ is involved. Notice that the steps are very similar to the Cholesky decomposition in Algorithm 7.

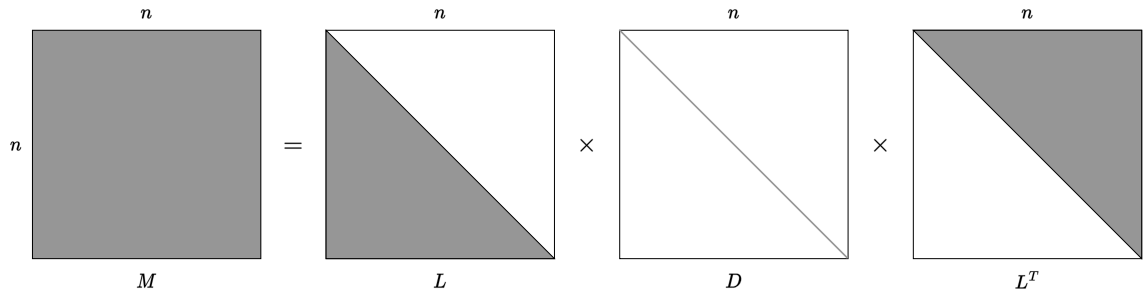


Figure 2.7: LDL decomposition

Algorithm 8: LDL decomposition

Input: Symmetric matrix $M \in \mathbb{R}^{n \times n}$
Output: $L, D \in \mathbb{R}^{n \times n}$ such that L is unit lower triangular and D is a diagonal matrix

$L = M$
for $k = 1$ **to** n **do**
  $L(k+1{:}n, k) \leftarrow \dfrac{L(k+1{:}n, k)}{L(k,k)}$
  for $i = k+1$ **to** n **do**
    $L(i{:}n, i) \leftarrow L(i{:}n, i) - L(i{:}n, k)\dfrac{L(i,k)}{L(k,k)}$
  end
end

The LDL decomposition is also closely related to the LU decomposition, as it shows a connection between the L and U when M is symmetric. If M is symmetric and has an LU decomposition $M = LU$, then U must be a row scaling of L^T using the diagonal entries of $D = UL^{-T}$. This brings us to the existence of the LDL decomposition, where according to [43, Theorem 4.1.3], if $M \in \mathbb{R}^{n \times n}$ is symmetric and its principal submatrix $m_{1:k,1:k}$ is nonsingular for $k = 1, 2, \ldots, n-1$, then M has a unique LDL decomposition as in Equation (2.23). This means that the LDL decomposition exists even though M is not positive definite, in contrast to the Cholesky decomposition that requires M to be positive definite. Some indefinite matrices can still have LDL decomposition with negative entries in D. This allows the LDL decomposition to be useful in finding eigenvalue(s) using bisection that will be covered in Section 2.4.7.

2.4 Symmetric Dense Eigenvalue Solvers

Even though eigenvalues and eigenvectors have clear definitions and elegant properties as seen in Section 2.1, the best way to actually compute them is far from obvious. Looking at the characteristic polynomial described in Section 2.1.3, the first idea that might come to mind is to compute the coefficients of the characteristic polynomial and obtain its roots using an efficient polynomial rootfinder. Unfortunately, this is not a good idea since finding the roots of a polynomial is an ill-conditioned problem in general, even though the underlying symmetric eigenvalue problem is known

to be well-conditioned [43, 83]. Instead, the best methods are to compute the eigenvalue-revealing decomposition of A, so that the eigenvalues appear as entries of one of the factors. Moreover, these methods in general are iterative algorithms, since otherwise it would be a contradiction to the fundamental theorem of Abel-Ruffini that no such algorithm exists for the computation of the roots of a general polynomial of degree greater than 4 [42]. Therefore, the goal of an eigenvalue solver is to produce a sequence of numbers that converge rapidly toward the eigenvalues/eigenvectors.

Note that in this section we omit the detail of the termination condition for some iterative algorithms by expressing the loop as "**for** $k = 1, 2, \ldots$". While termination conditions are of course very important in practice, we believe that publicly available softwares such as LAPACK [8] and MATLAB contain well-optimized techniques along with the detailed discussions in their respective technical reports.

2.4.1 Reduction to Tridiagonal Form

We start by describing one of the most important operations in solving the symmetric eigenvalue problem, the reduction of a full matrix to its tridiagonal form using a sequence of unitary similarity transformations. A tridiagonal matrix is a matrix that has non-zero elements only on the main diagonal, subdiagonal, and superdiagonal. An example is shown on the right-hand side of Figure 2.8.

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}} \rightarrow \underset{QAQ^T}{\begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}}.$$

Figure 2.8: Example of reduction to tridiagonal form

We have seen similar techniques in Section 2.2 that introduce zero to a matrix using an orthogonal matrix. Since A is symmetric, the first idea that might come to mind is to apply the Householder matrices from both sides of A instead of only from the left, thus reducing A into a diagonal matrix using similarity transformation. This will not work because multiplying the Householder matrix from the right involves replacing each column with a linear combination of all the columns, thus destroying the zeros that were previously introduced, as shown in Figure 2.9 (recall that the symbol $\times$ denotes a not necessarily zero entry and boldfaced symbol denotes an entry that has just been modified). Moreover, we knew that this idea had to fail, because otherwise it would contradict the argument above that there is no finite process that can reveal the eigenvalues of A in an exact manner.

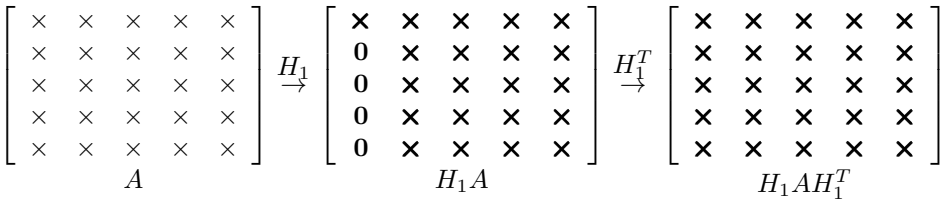


Figure 2.9: Multiplication of Householder matrix used for QR decomposition from both sides of a 5×5 matrix

The method that we discuss here still uses the Householder reflection to introduce zeros to the matrix and transform it into a tridiagonal form. However, unlike in the Householder QR where we choose a reflector that annihilates all but the first element of the vector x, here we choose the Householder reflector that leaves the first element untouched and annihilates all the others but the second element of x. This way, when it is multiplied from the right, it does not alter the zeros that have been previously introduced, as shown in Figure 2.10.

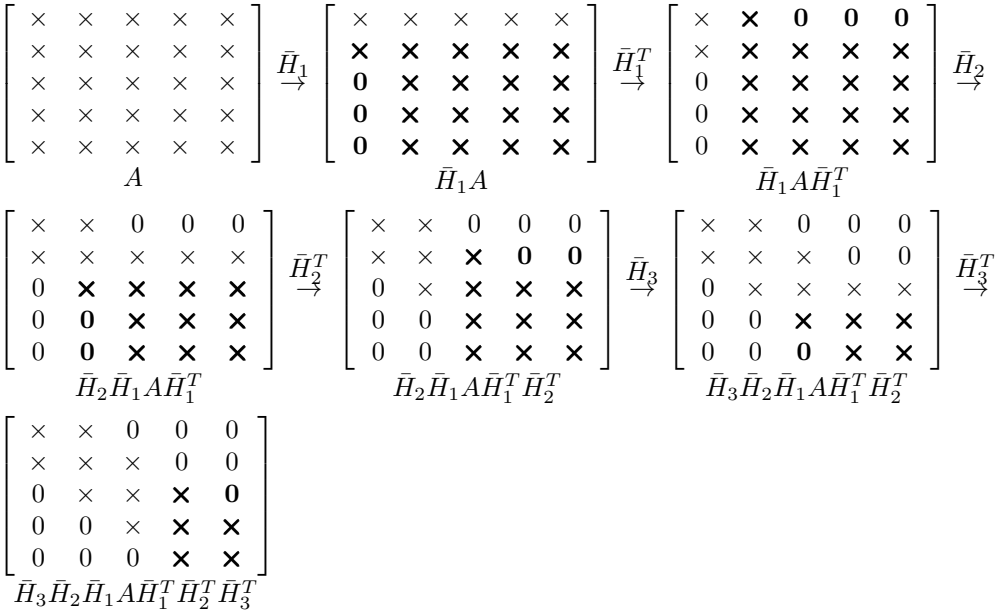


Figure 2.10: Tridiagonal reduction using Householder reflection of a 5×5 matrix

Using the same function $house(x)$ as the one in Section 2.2.3, the Householder tridiagonal reduction is formulated in Algorithm 9. Just as in Algorithm 3, the orthogonal matrix Q is not explicitly formed and the Householder vectors are stored in the zero parts in the bottom left part of the tridiagonal matrix. These can be used to reconstruct Q later if necessary.

The main reason why tridiagonal reduction is important is that it reduces the eigenvalue problem into a much simpler form. Moreover, since it uses similarity transformation, the eigenvalues are

Algorithm 9: Householder reduction to tridiagonal form

Input: $A \in \mathbb{R}^{n \times n}$
Output: Householder vectors $v_j \in \mathbb{R}^{n-j}$ for $j = 1, 2, \ldots, n-2$ and tridiagonal matrix $Q^T AQ$

for $j = 1$ **to** $n - 2$ **do**
  $[v, \beta] = house(a_{j+1:n,j})$
  $a_{j+1:n,j:n} \leftarrow (I - \beta v v^T) a_{j+1:n,j:n}$
  $a_{j:n,j+1:n} \leftarrow a_{j:n,j+1:n}(I - \beta v v^T)$
  $a_{j+2:n,j} \leftarrow v_{2:n-j}$
end

preserved and the eigenvectors can easily be recovered using the matrix Q. More importantly, although it requires an initial cost of $\mathcal{O}(n^3)$ to reduce the matrix A to a tridiagonal form, having A in its tridiagonal form greatly reduces the subsequent cost per iteration of the eigenvalue solvers that we will describe in the rest of this section. This is why most implementations of dense symmetric eigenvalue solvers start with reducing the matrix to the tridiagonal form.

2.4.2 Power Iteration

The *power iteration*, sometimes referred to as the *power method*, is the simplest method to compute eigenvalue and eigenvector. It is based on the fact that the sequence

$$\frac{v}{\|v\|}, \frac{Av}{\|Av\|}, \frac{A^2v}{\|A^2v\|}, \frac{A^3v}{\|A^3v\|}, \cdots,$$

under certain assumptions converges to the eigenvector corresponding to the largest eigenvalue of A (in absolute value). This provides the basic idea for a simple iterative method where we repeatedly multiply A to a chosen starting vector as shown in Algorithm 10. Each step of the power iteration requires two matrix-vector multiplications, leading to a cost of $\mathcal{O}(n^2)$ flops per iteration. If A is a tridiagonal matrix, the cost per iteration goes down to $\mathcal{O}(n)$ flops.

Algorithm 10: Power iteration

Input: $A \in \mathbb{R}^{n \times n}$
Output: Largest eigenpair $\left(\lambda^{(k)}, v^{(k)}\right)$

$v^{(0)}$ = some vector with $\left\|v^{(0)}\right\| = 1$
for $k = 1, 2, \ldots$ **do**
  $w = Av^{(k-1)}$
  $v^{(k)} = \dfrac{w}{\|w\|}$
  $\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$
end

We can analyze the algorithm as follows. Let $v^{(0)} \in \mathbb{R}^n$ with $\left\|v^{(0)}\right\| = 1$ be our starting vector and the set $\{z_1, z_2, \ldots, z_n\}$ be the orthonormal eigenvectors of A. We first write $v^{(0)}$ as the linear

combination

$$v^{(0)} = \alpha_1 z_1 + \alpha_2 z_2 + \cdots + \alpha_n z_n.$$

Then at the k-th iteration, $v^{(k)}$ is a multiple of $A^k v^{(0)}$. Thus for some constant γ_k we have

$$\begin{aligned} v^{(k)} &= \gamma_k A^k v^{(0)} \\ &= \gamma_k \left(\alpha_1 \lambda_1^k z_1 + \alpha_2 \lambda_2^k z_2 + \cdots + \alpha_n \lambda_n^k z_n \right) \\ &= \gamma_k \lambda_n^k \left(\alpha_1 \left(\lambda_1 / \lambda_n \right)^k z_1 + \alpha_2 \left(\lambda_2 / \lambda_n \right)^k z_2 + \cdots + \alpha_n z_n \right). \end{aligned} \tag{2.38}$$

From here we obtain the conclusion that if $z_n^T v^{(0)} \neq 0$, i.e. $\alpha_n \neq 0$, the iterates in Algorithm 10 satisfy the following properties

$$\left\| v^{(k)} - (\pm z_n) \right\| = \mathcal{O}\left(\left| \frac{\lambda_{n-1}}{\lambda_n} \right|^k \right), \quad \left| \lambda^{(k)} - \lambda_n \right| = \mathcal{O}\left(\left| \frac{\lambda_{n-1}}{\lambda_n} \right|^2 k \right) \tag{2.39}$$

as $k \to \infty$ [83, Theorem 27.1]. This implies that the convergence of the power iteration depends on the ratio between the largest two eigenvalues. If the largest two eigenvalues are significantly far from each other in magnitude, the convergence will be very fast. Otherwise, the convergence will be slow.

On its own, the power iteration has limited usage since it only computes the largest eigenpair. However, the idea of this method has provided the basis for many powerful methods. Several techniques that were suggested as improvements to the power method are also used to accelerate modern iterative methods [42].

2.4.3 Inverse Iteration

The inverse iteration is one of the most valuable tools of numerical linear algebra. The method takes its root in the power iteration. The key idea is that for any $\mu \in \mathbb{R}$ that is not an eigenvalue of A, the matrix $(A - \mu I)^{-1}$ has the same eigenvectors as A and the corresponding eigenvalues are $\{(\lambda_i - \mu)^{-1}\}$, where $\{\lambda_i\}$ are the eigenvalues of A. This suggests that if μ is close to an eigenvalue λ_i, then the value $(\lambda_i - \mu)^{-1}$ may have a much larger magnitude than the other $(\lambda_j - \mu)^{-1}$ for all $j \neq i$. Therefore, applying power iteration to the matrix $(A - \mu I)^{-1}$ should lead to a rapid convergence to the corresponding eigenvector z_i, thus giving the name *inverse iteration*. This method, particularly when $\mu \neq 0$, is also known as the *shifted inverse iteration*. The steps are summarized in Algorithm 11. Each iteration involves finding the solution of a linear system, which typically costs $\mathcal{O}(n^3)$ flops. However, one can factorize the matrix $A - \mu I$ in advance using LU or QR decomposition so that solving the linear system only costs $\mathcal{O}(n^2)$ flops. Further, if A is a tridiagonal matrix, the cost per iteration becomes $\mathcal{O}(n)$.

Unlike power iteration which only computes the largest eigenpair, in the inverse iteration we can choose which eigenpair to compute by giving the estimate μ that is close to the target eigenvalue.

Algorithm 11: Inverse iteration

Input: $A \in \mathbb{R}^{n \times n}$, $\mu \in \mathbb{R}$
Output: Eigenpair $\left(\lambda^{(k)}, v^{(k)}\right)$ such that $\lambda^{(k)}$ is the closest eigenvalue to μ

$v^{(0)}$ = some vector with $\left\|v^{(0)}\right\| = 1$
for $k = 1, 2, \ldots$ **do**
  Solve $(A - \mu I)\, w = v^{(k-1)}$ for w
  $v^{(k)} = \dfrac{w}{\|w\|}$
  $\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$
end

Moreover, the rate of convergence can also be controlled since it depends on the quality of μ. If μ is much closer to one eigenvalue than to the others, then the largest eigenvalue of the matrix $(A - \mu I)^{-1}$ would be significantly larger than the other eigenvalues, leading to very fast convergence.

The convergence of inverse iteration can be analyzed using a reasoning similar to the one used for power iteration in Section 2.4.2. Let λ_i be the closest eigenvalue to μ and λ_j is the second closest, that is

$$|\lambda_i - \mu| < |\lambda_j - \mu| \leq |\lambda_l - \mu|$$

for all $l \neq i$ and $l \neq k$. Let us also assume that $z_i^T v^{(0)} \neq 0$. Then the iterates in Algorithm 11 satisfy the following properties

$$\left\|v^{(k)} - (\pm z_i)\right\| = \mathcal{O}\left(\left|\frac{\lambda_i - \mu}{\lambda_j - \mu}\right|^k\right), \quad \left|\lambda^{(k)} - \lambda_i\right| = \mathcal{O}\left(\left|\frac{\lambda_i - \mu}{\lambda_j - \mu}\right|^2 k\right) \tag{2.40}$$

as $k \to \infty$ [83, Theorem 27.2].

Furthermore, if μ is an eigenvalue of A, the inverse iteration can compute the corresponding eigenvector in a single iteration. In fact, inverse iteration is the standard method to compute the corresponding eigenvectors when the eigenvalues are known. Even though the matrix $A - \mu I$ is singular or μ is extremely close to an eigenvalue so that the matrix $A - \mu I$ is very ill-conditioned, these do not cause any trouble to the inverse iteration [83, p. 206].

2.4.4 Rayleigh Quotient Iteration

The Rayleigh quotient in Equation (2.2) gives us a way to compute the eigenvalue estimate of a given eigenvector estimate. The inverse iteration in Section 2.4.3 gives us a way to compute the eigenvector estimate of a given eigenvalue estimate. The *Rayleigh quotient iteration* combines these two ideas to use continually improving eigenvalue estimates to increase the convergence rate of inverse iteration at each step, as shown in Algorithm 12. Each iteration involves solving a system of equations, leading to a cost of $\mathcal{O}(n^3)$ per iteration. Unlike the inverse iteration where we can pre-process the matrix $A - \mu I$ in advance, the shift value here is the $\lambda^{(k-1)}$ that may be different

Algorithm 12: Rayleigh quotient iteration

Input: $A \in \mathbb{R}^{n \times n}$

Output: Eigenpair $\left(\lambda^{(k)}, v^{(k)}\right)$ such that $v^{(k)}$ is the eigenvector close to $v^{(0)}$

$v^{(0)}$ = some vector with $\left\|v^{(0)}\right\| = 1$

$\lambda^{(0)} = \left(v^{(0)}\right)^T A v^{(0)}$

for $k = 1, 2, \ldots$ **do**

  Solve $\left(A - \lambda^{(k-1)} I\right) w = v^{(k-1)}$ for w

  $v^{(k)} = \dfrac{w}{\|w\|}$

  $\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$

end

on each iteration so an additional technique is required to reduce the cost per iteration. However, when A is a tridiagonal matrix, the cost per iteration is reduced to $\mathcal{O}(n)$ flops.

The convergence rate of this algorithm is dramatically higher than the power and inverse iteration that we have described above: each iteration triples the number of digits of accuracy. This method converges to an eigenpair of A for all except a set of measure zero of starting vectors $v^{(0)}$. If $v^{(0)}$ is sufficiently close to the eigenvector z_i and λ_i is the corresponding eigenvalue, then the iterates in Algorithm 12 satisfy the following:

$$\left\|v^{(k+1)} - (\pm z_i)\right\| = \mathcal{O}\left(\left\|v^{(k)} - (\pm z_i)\right\|^3\right), \qquad \left|\lambda^{(k+1)} - \lambda_i\right| = \mathcal{O}\left(\left|\lambda^{(k)} - \lambda_i\right|^3\right) \tag{2.41}$$

as $k \to \infty$ [83, Theorem 27.3].

2.4.5 QR Algorithm

QR algorithm was first introduced by Francis back in the early 1960s [36, 37]. Since then it has been the most widely used method to compute all eigenvalues of a matrix. This method relies heavily on the QR decomposition that we have covered in Section 2.2. The are several variants of the QR algorithm. The commonly used modern implementation in LAPACK [8] is based on the multi-shift variant with aggressive early deflation described in [20]. Here we discuss two basic variants that we believe is essential to understand the fundamental of QR algorithm.

2.4.5.1 QR Algorithm without Shifts

Algorithm 13: QR algorithm without shifts

Input: $A \in \mathbb{R}^{n \times n}$

Output: Diagonal matrix $A^{(k)}$ containing the eigenvalues of A

$A^{(0)} = A$

for $k = 1, 2, \ldots$ **do**

  $Q^{(k)} R^{(k)} = A^{(k-1)}$

  $A^{(k)} = R^{(k)} Q^{(k)}$

end

The QR algorithm without shifts is the most basic version of QR algorithm, as shown in Algorithm 13. This variant is rarely used in practice, but it is a good start to understand the QR algorithm in general. At each iteration, we perform QR decomposition of the iterate (line 3), and then form the next iterate by multiplying back the Q and R factors in reverse order (line 4). These simple steps are equivalent to repeatedly applying a similarity transformation to the matrix A. This is because the QR decomposition in line 3 amounts to forming the matrix

$$R^{(k)} = \left(Q^{(k)}\right)^T A^{(k-1)}.$$

The multiplication in line 4 can then be written as

$$A^{(k)} = \left(Q^{(k)}\right)^T A^{(k-1)} Q^{(k)}. \tag{2.42}$$

Since $Q^{(k)}$ is orthogonal, Equation (2.42) shows a similarity transformation being applied to the iterate $A^{(k-1)}$. This iterate, under certain assumption, converges to a diagonal matrix containing the eigenvalues of A. Each iteration involves a QR decomposition and a matrix-matrix multiplication, leading to cost of $\mathcal{O}(n^3)$ per iteration. If A is a tridiagonal matrix, the cost per iteration becomes $\mathcal{O}(n^2)$ flops.

Additionally, the QR algorithm can also be used to compute the corresponding eigenvectors. The key observation is that the process in Algorithm 13 generates a sequence of matrices defined by the QR decomposition of the k-th power of A. Let $\hat{Q}^{(k)} = Q^{(1)}Q^{(2)}\cdots Q^{(k)}$ and $\hat{R}^{(k)} = R^{(k)}R^{(k-1)}\cdots R^{(1)}$. Thus, the k-th power of A can be expressed as

$$A^k = \hat{Q}^{(k)}\hat{R}^{(k)}. \tag{2.43}$$

The iterate in Equation (2.42) can also be written as

$$A^{(k)} = \left(\hat{Q}^{(k)}\right)^T A \hat{Q}^{(k)}. \tag{2.44}$$

The proof for Equation (2.43) and (2.44) can be seen in [83, Theorem 28.3]. Therefore, from Equation (2.44) we can see that if $A^{(k)}$ is a diagonal matrix and $\hat{Q}^{(k)}$ is orthogonal, which is true since it is a product of orthogonal matrices, then $\hat{Q}^{(k)}$ contains the orthonormal eigenvectors of A.

Let $|\lambda_1| < |\lambda_2| < \cdots < |\lambda_n|$ be the eigenvalues of A and their corresponding eigenvectors $\{z_i\}$ are the columns of the matrix Z such that all leading principal minors of Z are invertible, i.e. all of its upper-left submatrices of dimensions $1 \times 1,\ 2 \times 2, \ldots,\ n \times n$ are nonsingular. Then, the iterate $A^{(k)}$ in Algorithm 13 converges linearly with the constant

$$\max_{1<k\leq n} \frac{|\lambda_{k-1}|}{|\lambda_k|}$$

to the diagonal matrix $diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$, and the matrix $\hat{Q}^{(k)}$ converges at the same rate to Z as $k \to \infty$ [83, Theorem 28.4].

2.4.5.2 QR Algorithm with Shifts

Here we introduce a more practical version of the QR algorithm. This algorithm is obtained by applying the following modifications to Algorithm 13 in order to improve the overall performance in practice:

1. The matrix A is first reduced to a tridiagonal form before the iteration begins.
2. A well chosen shift value $\mu^{(k)}$, which corresponds to an eigenvalue estimate, is introduced at each iteration so that $A^{(k)} - \mu^{(k)}$ is factorized instead of $A^{(k)}$.
3. Whenever possible, for example when an eigenvalue is found, the problem is "deflated" by subdividing A into submatrices.

The practical QR algorithm that has the modifications mentioned above is formulated in Algorithm 14.

Algorithm 14: QR algorithm with shifts

Input: $A \in \mathbb{R}^{n \times n}$
Output: Diagonal matrix $A^{(k)}$ containing the eigenvalues of A

1 $(Q^{(0)})^T A^{(0)} Q^{(0)} = A$
2 **for** $k = 1, 2, \ldots$ **do**
3 Choose a shift $\mu^{(k)}$
4 $Q^{(k)} R^{(k)} = A^{(k-1)} - \mu^{(k)} I$
5 $A^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I$
6 **if** *any off-diagonal element of* $A^{(k)}_{j,j+1}$ *is sufficiently close to zero* **then**
7 Set $A_{j,j+1} = A_{j+1,j} = 0$ to obtain $\begin{bmatrix} A_1 & \mathbf{0} \\ \mathbf{0} & A_2 \end{bmatrix} = A^{(k)}$
8 Apply the QR algorithm to the matrices A_1 and A_2.
9 **end**
10 **end**

Let us analyze Algorithm 14. The tridiagonal reduction in line 1 incurs a cost of $\mathcal{O}(n^3)$ at the beginning of the algorithm. However, the resulting tridiagonal structure of the iterate $A^{(k)}$ can then be exploited within each iteration: the QR decomposition of a tridiagonal matrix can be done in $\mathcal{O}(n^2)$ flops instead of $\mathcal{O}(n^3)$ since a Householder matrix of dimension 2×2 is enough for each step of the QR decomposition. Also, the RQ multiplication in line 5 again produces a tridiagonal matrix, so the tridiagonal structure is preserved along the iterations. This significantly reduces the overall cost, especially because the QR algorithm requires $\mathcal{O}(n)$ iterations to compute all eigenvalues.

Next, the introduction of a shift in line 3 is done to increase the convergence rate. This is actually similar to what we did in the Rayleigh quotient iteration. If the chosen shifts are good eigenvalue estimates, the last diagonal entry of $A^{(k)}$ converges quickly to an eigenvalue. There are two common ways to choose the shifts for QR algorithm: the *Rayleigh quotient shift* and the *Wilkinson shift*. Both

techniques lead to cubic convergence, in the sense that $a_{n,n}^{(k)}$ converges cubically to an eigenvalue as in the right part of Equation (2.41). However, the convergence of Rayleigh quotient shift is not guaranteed for all initial conditions. The Rayleigh quotient shift at each iteration is defined as

$$\mu^{(k)} = a_{n,n}^{(k)}, \tag{2.45}$$

i.e. the bottom-right entry of the iterate $A^{(k)}$. The Wilkinson shift corresponds to the eigenvalue of the bottom-right 2×2 submatrix of $A^{(k)}$ that is close to the last diagonal entry of $A^{(k)}$. Let $B^{(k)} \in \mathbb{R}^{2\times 2}$ be that particular submatrix

$$B^{(k)} = \begin{bmatrix} a_{n-1,n-1}^{(k)} & a_{n-1,n}^{(k)} \\ a_{n,n-1}^{(k)} & a_{n,n}^{(k)} \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix},$$

and $\delta = (\alpha - \gamma)/2$. Then, the Wilkinson shift is defined as

$$\mu^{(k)} = \gamma - \frac{\operatorname{sign}(\delta)\,\beta^2}{|\delta| + \sqrt{\delta^2 + \beta^2}}. \tag{2.46}$$

Finally, the steps in line 6-9 is done to deflate the problem whenever the value zero (or a sufficiently small number) appear in the subdiagonal and superdiagonal of $A^{(k)}$. In other words, if at some point $A^{(k)}$ becomes a block diagonal matrix

$$A = \begin{bmatrix} A_1 & \mathbf{0} \\ \mathbf{0} & A_2 \end{bmatrix}, \tag{2.47}$$

then the problem can be reduced to finding the eigenvalues of A_1 and A_2. Let (α, x) and (β, y) are the eigenpairs of A_1 and A_2, respectively. Then, it follows that

$$\begin{bmatrix} A_1 & \mathbf{0} \\ \mathbf{0} & A_2 \end{bmatrix} \begin{bmatrix} x \\ \mathbf{0} \end{bmatrix} = \alpha \begin{bmatrix} x \\ \mathbf{0} \end{bmatrix}, \text{ and } \begin{bmatrix} A_1 & \mathbf{0} \\ \mathbf{0} & A_2 \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ y \end{bmatrix} = \beta \begin{bmatrix} \mathbf{0} \\ y \end{bmatrix}.$$

Therefore, the eigenvalues of A are the union of the eigenvalues of A_1 and A_2, that is

$$\Lambda(A) = \Lambda(A_1) \cup \Lambda(A_2). \tag{2.48}$$

When the eigenvectors are also needed, they can be computed using a technique similar to that of the unshifted QR algorithm that we have described in the previous section.

2.4.6 Divide-and-Conquer

The divide-and-conquer algorithm was first introduced by Cuppen in the 1980s [34]. The method is based on a recursive subdivision of a symmetric tridiagonal eigenvalue problem into problems of smaller size, which can be viewed as a more general version of the one we just saw in Equation (2.47). Let $T \in \mathbb{R}^{n\times n}$ be a symmetric tridiagonal matrix and $p = \lfloor n/2 \rfloor$. The *divide* step is to split T into

$$T = \begin{bmatrix} T_1 & \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \alpha & \mathbf{0} \end{bmatrix} \\ \begin{bmatrix} \mathbf{0} & \alpha \\ \mathbf{0} & \mathbf{0} \end{bmatrix} & T_2 \end{bmatrix} = \begin{bmatrix} \hat{T}_1 & \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{0} \end{bmatrix} \\ \begin{bmatrix} \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} & \hat{T}_2 \end{bmatrix} + \begin{bmatrix} \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \alpha \end{bmatrix} & \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \alpha & \mathbf{0} \end{bmatrix} \\ \begin{bmatrix} \mathbf{0} & \alpha \\ \mathbf{0} & \mathbf{0} \end{bmatrix} & \begin{bmatrix} \alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \end{bmatrix}, \tag{2.49}$$

such that $T_1, \hat{T}_1 \in \mathbb{R}^{p\times p}$, $T_2, \hat{T}_2 \in \mathbb{R}^{(n-p)\times(n-p)}$, and $\alpha = t_{p+1,p} = t_{p,p+1} \neq 0$. Also, $\hat{T}_1$ has the same elements as T_1 except that its bottom-right element is replaced by $t_{p,p} - \alpha$, and $\hat{T}_2$ has the same elements as T_2 except that its top-left element is replaced by $t_{p+1,p+1} - \alpha$. In other words, we have just written the tridiagonal matrix T as a sum of a block-diagonal matrix with tridiagonal blocks and a rank-1 matrix. If we know the eigenvalues of $\hat{T}_1$ and $\hat{T}_2$, we can calculate the eigenvalues of T quickly since the rightmost term of Equation (2.49) is a rank-1 matrix. Thus, we are left with the problem of finding the eigenvalues of smaller tridiagonal matrices $\hat{T}_1$ and $\hat{T}_2$. We recursively apply the same split operation as in Equation (2.49) until the problem size becomes small enough so that we can directly compute the eigenvalues. For example, we can recurse until we reach 1×1 or 2×2 eigenvalue problems where we can quickly compute the eigenvalues using a simple formula. However, in practice one usually switch to another eigenvalue solver such as the QR algorithm when the problem size is sufficiently small to get a better performance.

Next, we consider the *conquer* step: calculate the eigenvalues of T from the obtained eigenvalues of $\hat{T}_1$ and $\hat{T}_2$. Suppose that we have obtained the eigendecomposition of $\hat{T}_1$ and $\hat{T}_2$, that is

$$\hat{T}_1 = Q_1 D_1 Q_1^T, \text{ and } \hat{T}_2 = Q_2 D_2 Q_2^T,$$

where $Q_1 \in \mathbb{R}^{p\times p}$ and $Q_2 \in \mathbb{R}^{(n-p)\times(n-p)}$ are orthogonal matrices. Then we can rewrite Equation (2.49) as

$$T = \begin{bmatrix} Q_1 & \mathbf{0} \\ \mathbf{0} & Q_2 \end{bmatrix} \left(\begin{bmatrix} D_1 & \mathbf{0} \\ \mathbf{0} & D_2 \end{bmatrix} + \alpha z z^T \right) \begin{bmatrix} Q_1^T & \mathbf{0} \\ \mathbf{0} & Q_2^T \end{bmatrix}, \tag{2.50}$$

where $z^T = [(Q_1)_{p,:}\ (Q_2)_{1,:}]$, i.e. z^T is the row vector made from concatenating the last row of Q_1 with the first row of Q_2. Because Equation (2.50) is a similarity transformation, we have just reduced the problem further to finding the eigenvalues of a diagonal matrix plus a rank-1 matrix. For the sake of simplicity, let us redefine the problem as finding the eigenvalues of the matrix $D + ww^T$, where $D \in \mathbb{R}^{p\times p} = diag(d_1, d_2, \ldots, d_p)$ and $w \in \mathbb{R}^p$. Then, the eigenvalues of $D + ww^T$ are the roots of the following rational function

$$f(\lambda) = 1 + \sum_{j=1}^{p} \frac{w_j^2}{d_j - \lambda}. \tag{2.51}$$

The equation $f(\lambda) = 0$ is also known as the *secular equation*. Newton's iteration is typically used to solve the secular equation, where each root requires $\mathcal{O}(p)$ flops. Once an eigenvalue λ is found, the corresponding eigenvector x can be obtained from

$$x = (D - \lambda I)^{-1} w. \tag{2.52}$$

This results in a cost of $\mathcal{O}(p^2)$ flops to find the eigenvalues of a $p \times p$ tridiagonal matrix at each conquer step. Note that the formula in Equation (2.52) is usually not directly used in practice due to stability concern [70]. The method described in [45] can be used instead to obtain the eigenvector in a more stable manner.

Assuming that the initial matrix size n is split in half at each recursion step, we have a total cost of

$$\mathcal{O}\left(n^2 + 2\left(\frac{n}{2}\right)^2 + 4\left(\frac{n}{4}\right)^2 + 8\left(\frac{n}{8}\right)^2 + \cdots + n\left(\frac{n}{n}\right)^2\right) = \mathcal{O}(n^2)$$

to find all eigenvalues of a tridiagonal matrix using the divide-and-conquer algorithm. This method is known to be about two times faster than the QR algorithm if both eigenvectors and eigenvalues are required. A rigorous analysis can be found in [83, p. 232].

2.4.7 Bisection

The algorithms that we have discussed so far allow us to compute the eigenpair that is close to a chosen value μ or vector v, and also all eigenvalues/eigenvectors. However, what if we only want a specific subset of the eigenvalues (or eigenvectors)? For example, the 10-th smallest eigenvalue, the largest 20 eigenvalues, 5 interior eigenvalues around the median eigenvalue, or even all eigenvalues within a specific interval [0, 1]. For such cases, the bisection method can be used to find only the target eigenvalues without computing the others. When the number of eigenvalues required is much smaller than the total number of eigenvalues, this method is more efficient than computing all the eigenvalues. If the corresponding eigenvectors are also required, they can also be found using, for example, one step of the inverse iteration.

The bisection eigenvalue solver relies on the ability to count the number of eigenvalues of A that are smaller than a value μ, i.e. the number of eigenvalues within the interval $(-\infty, \mu)$. Let us define this as the function

$$\nu(A, \mu) = |\{\lambda \in \Lambda(A) | \lambda < \mu\}| . \tag{2.53}$$

In order for the bisection method to work efficiently, we need an efficient way to evaluate the function in Equation (2.53) without actually computing the eigenvalues. According to [71], there are two ways to do that: one that is based on the *Sturm sequence*, and another one that is based on *Sylvester's inertia theorem*. Both methods determine the number of eigenvalues that are smaller than the value μ by counting the number of negative eigenvalues of the matrix $A - \mu I$. This is easy to verify because if $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ are the eigenvalues of A, then $\{\lambda_1 - \mu, \lambda_2 - \mu, \ldots, \lambda_n - \mu\}$ are the eigenvalues of $A - \mu I$. Therefore, all eigenvalues of A that are smaller than μ will become negative eigenvalues in $A - \mu I$. We briefly explain both methods as follows:

1. *Counting negative eigenvalues with Sturm's sequence*: let $A^{(j)}$ be the principal submatrix of A such that $A^{(j)} = A_{1:j,1:j}$. According to [87, 83], the number of negative eigenvalues of A is equal to the number of sign changes in the sequence

$$1, \det\left(A^{(1)}\right), \det\left(A^{(2)}\right), \ldots, \det\left(A^{(n)}\right), \tag{2.54}$$

   where a "sign change" denotes a transition from positive $(+)$ to negative $(-)$ or vice versa. This still works even if zero determinants are encountered along the way, if we assume the

"sign change" to be a transition from + or 0 to − or from − or 0 to +, but not from + or − to 0 [83]. The sequence in Equation (2.54) is known as the Sturm sequence. If A is a full matrix, the determinants of its principal submatrices can be computed using the Gaussian elimination with $\mathcal{O}(n^3)$ flops. Since the determinant of a lower (and upper) triangular matrix is the product of its diagonal entries, it is easy to obtain the determinants once the L and U factors are available. However, if A has been reduced to a tridiagonal matrix, the determinants can be computed using a three-term recurrence relation described in [83, p. 229] with $\mathcal{O}(n)$ flops.

2. *Counting negative eigenvalues with Sylvester's theorem*: for any invertible matrix $X \in \mathbb{R}^{n\times n}$, the transformation XAX^T is called a *congruence transformation* of A. The matrix B is said to be *congruent* to A if there exists an invertible matrix X such that $B = XAX^T$. Unlike the similarity transformation presented in Section 2.1.4 (notice that the rightmost term here is X^T instead of X^{-1}), the congruence transformation in general does not preserve eigenvalues. Nevertheless, it preserves the signs of the eigenvalues, as stated in Sylvester's inertia theorem that the number of positive, negative, and zero eigenvalues of a matrix is invariant under congruence transformation [71, Fact 1.6]. These three numbers are called the *inertia* of A. Evaluating the function $\nu(A, \mu)$ is equivalent to computing the inertia of a matrix A that has been shifted by a value μ. If $A - \mu I = LDL^T$ is the LDL decomposition as described in Section 2.3.3, then the number of negative eigenvalues of $A - \mu I$ is equal to the number of negative entries in the diagonal matrix D [71]. If A is a full-matrix, the LDL decomposition costs $\mathcal{O}(n^3)$ flops, whereas if A has been reduced to tridiagonal form, the cost goes down to $\mathcal{O}(n)$ per LDL decomposition. Although this has about the same cost as the one using Sturm's sequence, this method is more preferred in finite precision computations [71, p. 55].

Once we have the means to count negative eigenvalues, we can perform a binary search to find the target eigenvalue(s) from a given interval. Suppose we want to estimate the k-th ($1 \leq k \leq n$) smallest eigenvalue of A, i.e. $\tilde{\lambda}_k \in [\alpha, \beta]$. Let ϵ_{ev} be the minimum interval size to stop the binary search. The bisection eigenvalue solver that uses Sylvester's inertia theorem is shown in Algorithm 15. This algorithm is also referred to as *slicing-the-spectrum* [71].

We provide an example of the bisection process in Figure 2.11, where the search interval is halved after each iteration. Red dots denote the (exact) eigenvalues that are larger than the midpoint of the current interval. Blue dots denote the (exact) eigenvalues that are smaller than the midpoint of the current interval. Since the binary search is stopped when the size of the search interval becomes smaller than the prescribed threshold ϵ_{ev} and our estimate is the midpoint of that final interval, then the absolute error of the eigenvalue estimate is bounded by

$$\left|\tilde{\lambda_k} - \lambda_k\right| < \frac{1}{2}\,\epsilon_{ev}, \tag{2.55}$$

Algorithm 15: Slicing the spectrum

Input: $A \in \mathbb{R}^{n\times n}$, $k \in \mathbb{Z}$, and $\alpha, \beta, \epsilon_{ev} \in \mathbb{R}$
Output: The k-th smallest eigenvalue $\tilde{\lambda}_k$

while $\beta - \alpha \geq \epsilon_{ev}$ **do**
  $\mu = \dfrac{\alpha+\beta}{2}$
  $LDL^T = A - \mu I$
  $\nu(A,\mu) = |\{j | D_{j,j} < 0\}|$
  if $\nu(A,\mu) \geq k$ **then** $\beta \leftarrow \mu$
  else $\alpha \leftarrow \mu$
end
$\tilde{\lambda}_k = \dfrac{\alpha+\beta}{2}$

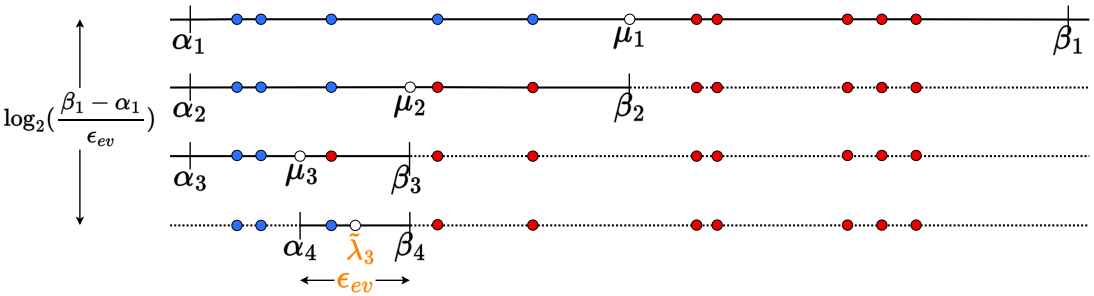


Figure 2.11: Example of binary search to find the 3rd smallest eigenvalue

where λ_k is the exact k-th smallest eigenvalue of A. Moreover, the number of binary search iterations is bounded by

$$\mathcal{O}(\log_2((\beta-\alpha)/\epsilon_{ev})).$$

This implies that we can control the eigenvalue accuracy by adjusting the depth of the binary search. When high accuracy is required, e.g. down to machine precision accuracy, the number of iterations required is $\mathcal{O}(\log_2(\epsilon_{machine}))$. However when such highly accurate computation is not necessarily needed, which is usually the case in many applications across many scientific and engineering domains, the number of iterations can be significantly reduced. This gives us a tool to compute, for example, a rough sketch of the eigenvalues with a relatively cheap cost.

Note that Algorithm 15 can be easily extended to compute some or all eigenvalues within a given interval. The basic idea is that one can calculate the number of eigenvalues within any interval $[\alpha, \beta)$ by the number of eigenvalues in $(-\infty, \beta)$ minus the number of eigenvalues in $(-\infty, \alpha)$, i.e. $\nu(A,\beta) - \nu(A,\alpha)$. Once the numbers are found, we know which eigenvalue indices k to be computed. Furthermore, when more than one eigenvalue is required, some of the computed $\nu(A,\mu)$ during the computation of one eigenvalue can be reused to accelerate the computation of other eigenvalues. For example in Figure 2.11, after the computation of $\tilde{\lambda}_3$ has been done, we know that we can use a smaller starting interval of $[\alpha_3, \mu_3]$ instead of $[\alpha_1, \beta_1]$ to compute the 1st and 2nd smallest eigenvalues. In

some cases, this may significantly reduce the total number of iterations compared to using the same starting interval to compute all target eigenvalues.

Chapter 3

Hierarchical Low-Rank Approximation

As we have seen in the previous chapter, matrix operations that are involved in solving eigenvalue problems typically cost $\mathcal{O}(n^3)$ flops. Even though these techniques are highly accurate, the cost quickly becomes very expensive when the problem size is large. As an example, computing all eigenvalues of a full matrix of order 131,072 using state-of-the-art LAPACK implementation on an Intel node with 72 cores (each running at 2.40 GHz), which is quite a powerful specification at the time of writing, requires about 128 GiB of storage and 3 hours to finish. If we further increase the matrix size by a factor of 10, i.e. to become an order of 1 million, the required time is expected to increase by a factor of $1,000$ due to the cubic complexity. As a result, this would be a major bottleneck in large-scale computation.

Hierarchical low-rank approximation emerges as a technique that can reduce both the storage and computational cost of matrix operations from $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively, down to $\mathcal{O}(n)$. It does so by exploiting the data sparsity that is often seen in dense matrices arising from many applications in scientific and engineering fields, notably from the discretization of integral and partial differential equations. It produces efficient and compact representations that utilize structured low-rank property of such matrices, hence they are often called *structured low-rank* formats. Due to the hierarchical nature that is exhibited in these formats, they are also well known as *hierarchical matrices*.

This chapter introduces the structured low-rank formats and their advantages over traditional techniques. We begin with introducing the concept of low-rank matrices and their approximations in Section 3.1. Then we explain matrix partitioning and admissibility in Sections 3.2 and 3.3, which are necessary to understand hierarchical low-rank approximation. After that, we finish by describing several well-known structured low-rank formats in Section 3.4.

3.1 Low-Rank Matrices

Let $M \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$. Multiplying M from the left of x produces a column vector that comes from the linear combination of the columns of M using the elements of x as the coefficient, as shown in Equation (3.1).

$$Mx = \left[\begin{array}{c|c|c|c} & & & \\ \boldsymbol{m}_1 & \boldsymbol{m}_2 & \cdots & \boldsymbol{m}_n \\ & & & \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \\ x_1\boldsymbol{m}_1 + x_2\boldsymbol{m}_2 + \cdots + x_n\boldsymbol{m}_n \\ \\ \end{bmatrix}. \tag{3.1}$$

The *range* of M is defined as the vector space formed by all possible linear combinations of its columns, that is

$$\text{range}(M) = \{Mx \mid x \in \mathbb{R}^n\}, \tag{3.2}$$

which sometimes is referred to as the *column space* of M. The *rank* of a matrix is equal to the dimension of its column space, which corresponds to the minimum number of linearly independent vectors that span its column space [12]. In other words, if M has a rank of k, then for any vector x, the product Mx can be written as a linear combination of k linearly independent vectors

$$Mx = \alpha_1 q_1 + \alpha_2 q_2 + \cdots + \alpha_k q_k$$

for some coefficients $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{R}$, where the vectors $q_1, q_2, \ldots, q_k \in \mathbb{R}^m$ forms the *column basis* of M. Thus, the rank can also be viewed as the number of linearly independent columns of M. Similar definitions also apply when considering the rows of M. When the number of rows is not equal to the number of columns, (i.e. $m \neq n$), the rank of M is taken as the minimum between the dimension of its column space and row space, that is

$$rank(M) \leq min(m, n). \tag{3.3}$$

When the number of rows is equal to the number of columns (i.e. $m = n$), the dimension of the column and row space are always equal [80]. Also, some properties related to the matrix rank according to [12] are listed as follows:

- $rank(X + Y) \leq rank(X) + rank(Y)$ for all $X, Y \in \mathbb{R}^{m \times n}$;
- $rank(XY) \leq min(rank(X), rank(Y))$ for all $X \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{n \times k}$.

A matrix $M \in \mathbb{R}^{m \times n}$ is said to be *rank-deficient* if it has a rank of $k < min(m, n)$. This kind of matrix is called *low-rank* matrix. On the contrary, if M has a rank of $k = min(m, n)$, it is called *full-rank* or *dense* matrix. For example, let $M \in \mathbb{R}^{3 \times 3}$ be

$$M = \left[\begin{array}{c|c|c} & & \\ \boldsymbol{m}_1 & \boldsymbol{m}_2 & \boldsymbol{m}_3 \\ & & \end{array}\right] = \begin{bmatrix} 2 & 4 & 10 \\ 2 & 2 & 6 \\ 0 & -2 & -4 \end{bmatrix}. \tag{3.4}$$

Here, it is easy to see that m_1 and m_2 are linearly independent since each one of them can not be expressed as a linear combination of the other. Moreover, it is clear that $m_3 = m_1 + 2m_2$. Therefore, M is a low-rank matrix with $rank(M) = 2$.

3.1.1 Representation

Consider a low-rank matrix $M \in \mathbb{R}^{m \times n}$ with a rank of $k < min(m, n)$. Since we know that among the n columns of M, only k of them are enough to represent the whole column space, there must be some matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$ such that

$$M = UV^T. \tag{3.5}$$

This representation is also called the *outer-product* form. Let $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k \in \mathbb{R}^m$ and $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k \in \mathbb{R}^n$ be the columns of U and V, respectively. Then we can rewrite Equation (3.5) as

$$M = \sum_{i=1}^{k} \boldsymbol{u}_i \boldsymbol{v}_i^T.$$

Thus, we can represent M by storing the vectors $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$, requiring $k(m + n)$ units of storage instead of mn units of storage when using the full matrix form. For example, the low-rank matrix in Equation (3.4) can also be written as

$$M = \underbrace{\begin{bmatrix} 2 & 4 \\ 2 & 2 \\ 0 & -2 \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}}_{V^T} = \begin{bmatrix} 2 & 4 & 10 \\ 2 & 2 & 6 \\ 0 & -2 & -4 \end{bmatrix}. \tag{3.6}$$

While in this case, the outer-product form does not seem to be beneficial in terms of storage, such representation will show its advantage when the rank k is significantly smaller than the matrix dimensions m and n. In such cases not only the storage is reduced, but the computational cost of matrix operations is also reduced, which we will discuss in Section 3.1.3.

3.1.2 Low-Rank Approximation

Although the promising properties of low-rank matrices that we have explained above, true low-rank matrix as in Equation (3.5) is rarely used in practical context. Instead, *numerically* low-rank matrices are often used, such that the product UV^T is not exactly equal to M, but approximately equal to M. *Low-rank approximation* is a common way to do it, where a matrix $M \in \mathbb{R}^{m \times n}$ is approximated by a matrix $\tilde{M} \in \mathbb{R}^{m \times n}$ with small rank $r < min(m, n)$ such that the error is bounded by a small value, that is

$$\left\| M - \tilde{M} \right\| \leq \epsilon \tag{3.7}$$

for some choice of norm, where $\epsilon \in \mathbb{R}$ is a prescribed *absolute error threshold*. Another commonly used formula is the relative error bound that is measured as

$$\frac{\left\|M - \tilde{M}\right\|}{\|M\|} \leq \epsilon, \tag{3.8}$$

such that the ϵ here is called the *relative error threshold*.

Low-rank approximation is a valuable tool in numerical linear algebra. It is closely related to Principal Component Analysis (PCA) and is often used in mathematical modeling and data compression. One representative example would be image compression in computers, where an image that is stored in the form of a matrix containing pixel values is compressed by removing certain components of the basis vectors that correspond to noises so that the image can still retain its quality with a smaller data size. In this section, we discuss several well-known techniques to perform low-rank approximation.

Note that in the context of this thesis, we assume a standard representation for the low-rank approximation of a numerically low-rank matrix M in the form of

$$\tilde{M} = USV^T, \tag{3.9}$$

such that the matrices U and V have orthonormal columns.

3.1.2.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is a well-known tool in scientific computations. The best low-rank approximation of a matrix by a rank r matrix is given by the SVD [51]. It is also a good start in mathematically understanding the concept of low-rank matrices. SVD factorizes a matrix $M \in \mathbb{R}^{m \times n}$ into

$$M = U\Sigma V^T \tag{3.10}$$

where $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix containing the basis of the column space of M, $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix containing the basis of the row space of M, and $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix containing the *singular values* of M in descending order.

The strength of SVD lies in the singular values that it produces. Let $k = min(m, n)$. Then the singular values are stored in the matrix $\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_k)$ such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k$. It turns out that the singular values reflect the rank of M. If M has a rank of s, then the values $[\sigma_1, \sigma_2, \ldots, \sigma_s]$ are greater than 0, and the rest $[\sigma_{s+1}, \ldots, \sigma_k]$ are all 0. Moreover, the singular values also reflect the quality of the low-rank approximation when using a certain rank r. Suppose that we have the SVD of M as in Equation (3.10). Then we can construct the r-rank approximation of M by taking only the first r singular values and the first r columns of U and V, that is:

$$\tilde{M} = U_r \Sigma_r V_r^T \tag{3.11}$$

where $U_r = u_{:,1:r}$, $V_r = v_{:,1:r}$, and $\Sigma_r = diag(\sigma_1, \sigma_2, \ldots, \sigma_r)$. This is also known as the *truncated SVD*, as shown in Figure 3.1. The approximation error is bounded by the $(r+1)$-th largest singular values of M such that

$$\left\| M - U_r \Sigma_r V_r^T \right\|_2 \leq \sigma_{r+1}.$$

Hence, by looking at how the singular values decay, we can choose the appropriate rank to obtain an approximation with the desired accuracy.

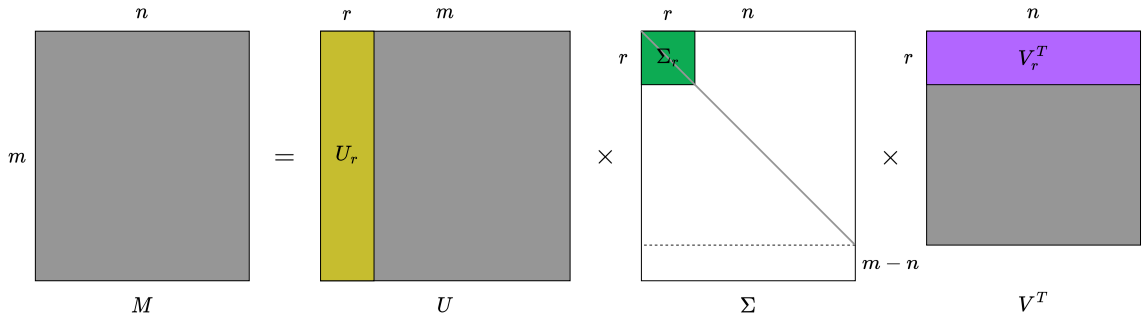


Figure 3.1: r-rank approximation using truncated SVD

Several methods have been proposed to perform the calculation of SVD efficiently. A well-known algorithm was developed by Golub and Kahan [41]. Their method relies on the creation of a bidiagonal matrix using the QR factorization technique (e.g. Householder reflection) and an iterative method to introduce zeros to the off-diagonal elements using Givens rotation. The latter is also referred to as Golub-Kahan SVD Step [43]. Computing the SVD of $M \in \mathbb{R}^{m \times n}$ requires a cost of $4m^2n + 8mn^2 + 9n^3$ flops if all factors U, Σ, V are computed. The cost may vary depending on the factors that will be computed. Detailed cost estimates for different cases of computed factors can be found in [43].

3.1.2.2 Rank-Revealing QR Decomposition

Consider a matrix $M \in \mathbb{R}^{m \times n}$. The basic idea of a rank-revealing QR decomposition is to permute the columns of M and perform a QR decomposition of

$$M\Pi = QR, \tag{3.12}$$

such that $Q \in \mathbb{R}^{m \times m}$ is an orthogonal matrix and $R \in \mathbb{R}^{m \times n}$ is an upper triangular matrix, which is similar to the regular full QR decomposition that we have explained in Section 2.2. However, here we have an appropriately chosen permutation matrix $\Pi \in \mathbb{R}^{n \times n}$ that reorders the columns of M so that the resulting QR decomposition reveals the rank of M.

Businger and Golub in [21] introduced the first algorithm to compute QR decomposition with column pivoting. The method is very similar to that of the regular Householder QR decomposition method. The main difference is at the beginning of each iteration, the columns of M are reordered

before the Householder reflection is generated. Before we reflect the j-th column of M to introduce zero below the diagonal, we first find the smallest number $p \in \mathbb{Z}$ ($j \leq p \leq n$) such that the p-th column of M has the highest magnitude, and then swap it with the j-th column. Then we generate the Householder reflection for the new j-th column and proceed like regular Householder QR. After that, we repeat the same process for the next columns until we reach the rightmost column.

Let $k < n$. At the end of the k-th iteration, after we reflected the k-th column of M, we are left with the matrix

$$H_k H_{k-1} \cdots H_1 \; M \; \Pi_1 \Pi_2 \cdots \Pi_k = \begin{bmatrix} R_{1,1} & R_{1,2} \\ \mathbf{0} & R_{2,2} \end{bmatrix}, \tag{3.13}$$

such that $H_j \in \mathbb{R}^{m \times m}$ is the j-th Householder reflection as defined in Equation (2.21), $\Pi_j \in \mathbb{R}^{n \times n}$ is the permutation matrix corresponding to the column swap at the j-th iteration, $R_{1,1} \in \mathbb{R}^{k \times k}$ is nonsingular upper triangular matrix, $R_{1,2} \in \mathbb{R}^{k \times (n-k)}$, and $R_{2,2} \in \mathbb{R}^{(m-k) \times (n-k)}$. At this point, depending on the value of $\|R_{2,2}\|$, we can choose to stop or continue the triangularization. If $\|R_{2,2}\| = 0$, then M must have a rank of k so we can stop the iteration, since we already obtained a triangular matrix anyway. But if $\|R_{2,2}\| \neq 0$, we usually continue the computation until we reach a state where $\|R_{2,2}\|$ is small enough. In practice, a common method is to stop the iteration when

$$\|R_{2,2}\| \leq \epsilon \, \|M\| \, , \tag{3.14}$$

for some prescribed error threshold $\epsilon \in \mathbb{R}$. The algorithm for computation of RRQR is presented in [43], which involves $4mnr - 2r^2(m+n) + 4r^3/3$ flops, where r is the numerical rank of M corresponding to the chosen threshold as in Equation (3.14).

Let the matrix $M \in \mathbb{R}^{m \times n}$ has a rank of k, in the sense that if we were to perform the RRQR on it, the process would stop after the k-th iteration because $\|R_{2,2}\| = 0$. Suppose that we have chosen a small value $\epsilon > 0$ such that the stopping criterion in Equation (3.14) is satisfied at the r-th ($r < k$) iteration. So we have $\hat{\Pi}_r = \Pi_1 \Pi_2 \cdots \Pi_r$, $Q = H_1 H_2 \cdots H_r$, and

$$\begin{aligned} M\hat{\Pi}_r &= Q_r R \\ &= \begin{bmatrix} Q_{1,1} & Q_{1,2} \\ Q_{2,1} & Q_{2,2} \end{bmatrix} \begin{bmatrix} R_{1,1} & R_{1,2} \\ \mathbf{0} & R_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} Q_{1,1} \\ Q_{2,1} \end{bmatrix} \begin{bmatrix} R_{1,1} & R_{1,2} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & Q_{1,2}R_{2,2} \\ \mathbf{0} & Q_{2,2}R_{2,2} \end{bmatrix}, \end{aligned} \tag{3.15}$$

such that $Q_{1,1} \in \mathbb{R}^{r \times r}$, $Q_{1,2} \in \mathbb{R}^{r \times (m-r)}$, $Q_{2,1} \in \mathbb{R}^{(m-r) \times r}$, and $Q_{2,2} \in \mathbb{R}^{(m-r) \times (m-r)}$. Then, if we take the approximation

$$M\hat{\Pi}_r \approx \begin{bmatrix} Q_{1,1} \\ Q_{2,1} \end{bmatrix} \begin{bmatrix} R_{1,1} & R_{1,2} \end{bmatrix}, \tag{3.16}$$

the error is proportional to the rightmost term in Equation (3.15).

Gu and Eisenstat in [46] have proposed a robust RRQR algorithm with better numerical properties, which they referred to as the Strong Rank-Revealing QR Decomposition. They showed that by using their method, the following property for the singular values of $R_{2,2}$ holds:

$$\sigma_i(R_{2,2}) \leq \sqrt{1 + r(n-r)}\, \sigma_{r+i}(M),$$

where $\sigma_i(X)$ denotes the i-th largest singular value of the matrix X. With this, we can express the approximation error as

$$\left\| M - \begin{bmatrix} Q_{1,1} \\ Q_{2,1} \end{bmatrix} \begin{bmatrix} R_{1,1} & R_{1,2} \end{bmatrix} \right\|_2 \leq \sqrt{1 + r(n-r)}\, \sigma_{r+1}(M). \tag{3.17}$$

3.1.2.3 Interpolative Decomposition

The Interpolative Decomposition (ID) is a low-rank approximation technique introduced by Cheng et al. in [32]. It approximates a numerically rank deficient matrix $M \in \mathbb{R}^{m \times n}$ using a rank $r < min(m, n)$ matrix such that

$$M \approx \hat{B}\hat{V}^T \tag{3.18}$$

where $\hat{B} \in \mathbb{R}^{m \times r}$ is a $m \times r$ submatrix of M formed by taking r of its columns and $\hat{V} \in \mathbb{R}^{n \times r}$ is a well-conditioned matrix that contains a $r \times r$ identity matrix as its submatrix and has a norm that is close to 1.

The main advantage of the ID lies in the fact that it reuses the columns of M in the approximation, i.e. the matrix $\hat{B}$ in Equation (3.18). This could lead to considerable storage savings in some cases, notably when factorizing large sparse matrices. Combined with the characteristic of the resulting $\hat{V}$, this factorization scheme can be used as an alternative in certain situations where the Singular Value Decomposition (SVD) cannot be used efficiently [32].

The calculation of ID is closely related to the rank-revealing QR decomposition that we have just described in Section 3.1.2.2. We start by computing the rank-revealing QR decomposition of M such that the stopping criteria is satisfied at the r-th iteration. Now consider the approximation in Equation (3.16). Since $R_{1,1}$ is upper triangular and nonsingular, we can compute its inverse and rewrite it as

$$M\hat{\Pi}_r \approx \begin{bmatrix} Q_{1,1} \\ Q_{2,1} \end{bmatrix} \begin{bmatrix} R_{1,1} & R_{1,2} \end{bmatrix} = \begin{bmatrix} Q_{1,1}R_{1,1} \\ Q_{2,1}R_{1,1} \end{bmatrix} \begin{bmatrix} I & R_{1,1}^{-1}R_{1,2} \end{bmatrix} = M_J\tilde{V}^T. \tag{3.19}$$

It is clear that the matrix $M_J \in \mathbb{R}^{m \times r}$ is formed by taking r columns of M, i.e. a submatrix of M. The matrix $\tilde{V}$ has $r \times r$ identity matrix as its submatrix, and its norm is reasonably close to 1 due to the nature of $R_{1,1}^{-1}$ and $R_{1,2}$ (see [32] for detailed explanation). Moving $\hat{\Pi}_r$ to the right hand side, we have the interpolative decomposition of M in the form of

$$M \approx \underbrace{M_J}_{\hat{B}} \underbrace{\tilde{V}^T\hat{\Pi}_r^T}_{\hat{V}^T}. \tag{3.20}$$

The method that we have just explained is known as the *one-sided* interpolative decomposition. If desired, we can continue and compute the *two-sided* interpolative decomposition, producing a factorization in the form of

$$M \approx UBV^T \tag{3.21}$$

where $B \in \mathbb{R}^{r \times r}$ is a $r \times r$ submatrix of M formed by taking r rows and r columns, $U \in \mathbb{R}^{m \times r}$ and $V \in \mathbb{R}^{n \times r}$ have identity matrix as their submatrices and their norms are close to 1. The additional step for two-sided ID is simply applying the one-sided ID (as in Equation (3.19) to the transpose of M_J, producing

$$M_J^T \hat{\Pi}_l \approx M_{JK}^T \tilde{U}^T. \tag{3.22}$$

Because M_{JK} a submatrix of M_J, then it also is a submatrix of M. Then by putting the permutation matrix $\hat{\Pi}_l$ to the right-hand side and substituting the transpose of $(M_J^T)^T = M_J$ in Equation (3.22) into Equation (3.20), we are left with the expression

$$M \approx \underbrace{\hat{\Pi}_l \tilde{U}}_{U} \underbrace{M_{JK}}_{B} \underbrace{\tilde{V}^T \hat{\Pi}_r^T}_{V^T},$$

that is a two-sided interpolative decomposition as in Equation (3.21).

The computational cost of one-sided ID is equivalent to the cost of rank-revealing QR decomposition, combined with the cost of computing the inversion of $R_{1,1}$ and multiplying it with $R_{1,2}$. If we proceed to the two-sided ID, we add a similar cost for the one-sided ID of M_J. In total, a typical two-sided ID requires $\mathcal{O}(mnr)$ flops.

3.1.2.4 Randomized SVD

Although the singular value decomposition could find the lowest approximation rank that is required to reach a given accuracy, its expensive computational cost makes it not attractive for large-scale computations [12]. The *randomized* SVD solves this problem by first employing randomized techniques to reduce the size of the matrix so that the SVD is only performed on a small matrix whose size is proportional to a known rank estimate. The method is based on the randomized technique presented in [52].

Given a matrix $M \in \mathbb{R}^{m \times n}$ and a desired approximation rank r, we first generate a random matrix $\Omega \in \mathbb{R}^{n \times (r+p)}$ where p is a prescribed oversampling parameter. Then we construct a random sampling of the range of M by

$$Y = M\Omega.$$

After that, we construct the orthogonal basis of Y using the QR decomposition. If $Q \in \mathbb{R}^{m \times (r+p)}$ is the resulting orthogonal basis, then Q can be seen as the approximate orthogonal basis of the

original matrix M. According to [52], the error estimate for the projection of M into a subspace formed by the columns of Q is expressed as

$$E\left(\left\|M - QQ^T M\right\|_2\right) \leq \left(1 + \frac{4\sqrt{r+p}}{p-1}\sqrt{min(m,n)}\right)\sigma_{r+1}(M). \tag{3.23}$$

Moreover, they also showed that for most cases, a small oversampling parameter $p = 5$ is enough to produce a reliable approximation.

Since we have a good approximation for the range of M with an orthogonal basis Q whose dimension is r, we can use it to accelerate the approximation of M. Let the matrix $B \in \mathbb{R}^{n\times(r+p)} = (Q^T M)^T = M^T Q$, which corresponds to a change of basis operation into Q. Then, we take the SVD of B such that

$$B = U_B \Sigma_B V_B^T.$$

This results in the SVD representation of

$$QQ^T M = QB^T = \underbrace{QV_B}_{U}\underbrace{\Sigma_B}_{\Sigma}\underbrace{U_B^T}_{V^T}.$$

Finally, the r-rank approximation can be obtained by truncating $U\Sigma V^T$, resulting in

$$M \approx \tilde{M} = u_{:,1:r}\sigma_{1:r,1:r}v_{:,1:r}^T$$

whose error is equivalent to that of Equation (3.23).

Note that here the SVD is only performed on a tall-skinny matrix $B \in \mathbb{R}^{n\times(r+p)}$ (under the assumption that $(r+p) \ll n$). This is much more cheaper than taking the SVD of the full matrix $M \in \mathbb{R}^{m\times n}$ just to take the first r columns of the resulting column basis. The Randomized SVD is formulated in Algorithm 16, which requires a cost of $\mathcal{O}(mn(r+p))$ flops.

Algorithm 16: Randomized SVD

Input: $M \in \mathbb{R}^{m\times n}$, approximation rank r, oversampling parameter p
Output: $M \approx U\Sigma V^T$ where $U \in \mathbb{R}^{m\times r}$, $\Sigma \in \mathbb{R}^{r\times r}$, and $V \in \mathbb{R}^{n\times r}$

$k = r + p$
$\Omega =$ `random_gaussian`(n, k)
$Y = M\Omega$
$[Q, R] = QR(Y)$
$B = M^T Q$
$[\hat{U}, \hat{\Sigma}, \hat{V}] = SVD(B)$
$U =$ first r columns of $(Q\hat{V})$
$\Sigma = \hat{\Sigma}(1{:}r, 1{:}r)$
$V =$ first r columns of $(\hat{U})$

3.1.3 Low-Rank Matrix Arithmetic

In this section, we introduce two arithmetic operations involving approximated low-rank matrices: matrix-matrix addition and matrix-matrix multiplication. We also demonstrate how those operations benefit from the low-rank factorized form described in Section 3.1.1.

3.1.3.1 Low-Rank Addition

Consider two numerically low-rank matrices $X, Y \in \mathbb{R}^{m \times n}$ with rank r_X and r_Y, respectively, that satisfy a certain error threshold ϵ. Let $X = U_X V_X^T$ and $Y = U_Y V_Y^T$. The simplest method of computing their sum is by agglomerating their components, such that

$$X + Y = \begin{bmatrix} U_X & U_Y \end{bmatrix} \begin{bmatrix} V_X & V_Y \end{bmatrix}^T. \tag{3.24}$$

This method produces the exact sum of X and Y. However, in practice, one usually wants the approximation of $X + Y$ that still satisfies the desired accuracy ϵ. Therefore, it is common to truncate the resulting matrix in Equation (3.24) to a certain rank r. For certain numerically low-rank matrices, especially the submatrices of a matrix that arise from the discretization of integral and partial differential equations, the rank r is usually closer to $max(rank(X), rank(Y))$ than $rank(X) + rank(Y)$. This process of concatenation and truncation is referred to as the *rounded addition* [12], as shown in Figure 3.2.

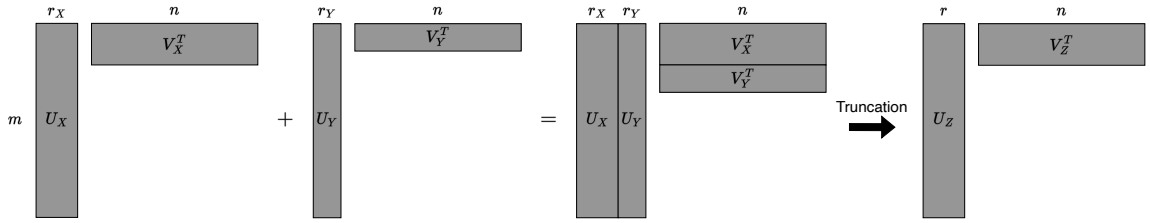


Figure 3.2: Rounded low-rank addition

By utilizing the factorized representation of the summands, the rounded addition method can compute the approximate addition in $\mathcal{O}((r_X + r_Y)^2(m+n))$ flops. This is much more efficient than recompressing the resulting matrix from scratch. Moreover, when the ranks r_X and r_Y are much smaller compared to the matrix dimensions m and n, the cost is smaller than that of dense-dense matrix addition that typically requires $\mathcal{O}(mn)$ flops. The steps for the rounded low-rank addition are summarized in Algorithm 17. Note that matrix subtraction can also be done by performing low-rank addition of X and $-1 \times Y$.

Algorithm 17: Rounded low-rank addition

Input: Low-rank matrices $X = U_X V_X^T$ and $Y = U_Y V_Y^T$, error threshold $\epsilon \in \mathbb{R}$
Output: $X + Y \approx Z = U_Z V_Z^T$, such that $U_Z \in \mathbb{R}^{m \times r}$ and $V_Z \in \mathbb{R}^{n \times r}$

1 $\hat{U} = \begin{bmatrix} U_X & U_Y \end{bmatrix}$, $\hat{V} = \begin{bmatrix} V_X & V_Y \end{bmatrix}$
2 $[Q_U, R_U] = QR(\hat{U})$
3 $[Q_V, R_V] = QR(\hat{V})$
4 $[U, \Sigma, V] = SVD(R_U R_V^T)$
5 Choose a rank r that satisfies the error threshold ϵ
6 $U_Z \leftarrow$ first r columns of $Q_U U$
7 $V_Z \leftarrow$ first r columns of $Q_V V \Sigma$

3.1.3.2 Low-Rank Multiplication

Let $X, Y \in \mathbb{R}^{n \times n}$ be low-rank matrices with rank r in their factorized forms such that $X = U_X V_X^T$ and $Y = U_Y V_Y^T$. For the multiplication involving low-rank matrices in their factorized form, let us divide them into two cases: the multiplication of dense matrix and low-rank matrix and the multiplication between two low-rank matrices.

For the first case, multiplication involving a dense matrix and a low-rank matrix amounts to a single multiplication of the dense matrix with either one of the left or right factor of the low-rank matrix. Let $C \in \mathbb{R}^{n \times n}$ be a dense matrix. Multiplying C to X from the left side results in the matrix

$$CX = Z = C(U_X V_X^T) = \underbrace{(CU_X)}_{U_Z} \underbrace{V_X^T}_{V_Z^T}.$$

On the other hand, multiplying C to X from the right side produces the matrix

$$XC = Z = (U_X V_X^T)C = \underbrace{U_X}_{U_Z} \underbrace{V_X^T C}_{V_Z^T}.$$

Hence the computational cost is that of the multiplication between C and either U_X or V_X^T, which is $\mathcal{O}(n^2 r)$ flops.

For the second case, multiplication between two low-rank matrices can be done by simply multiplying their factors. The product XY can be expressed as

$$XY = Z = (U_X V_X^T)(U_Y V_Y^T) = \underbrace{U_X}_{U_Z} \underbrace{(V_X^T U_Y) V_Y^T}_{V_Z^T}.$$

The same idea could be applied for the product YX. Hence, the computational cost for this operation is that of the multiplication of $(V_X^T U_Y)V_Y^T$ to construct V_Z^T, which is $\mathcal{O}(nr^2)$ flops.

From these two cases, it is clear that when the rank r is much smaller than the matrix dimension n, the multiplication involving low-rank matrices in their factorized forms requires much smaller cost compared to dense-dense matrix multiplication that requires $\mathcal{O}(n^3)$ flops.

3.2 Matrix Partition

While it seems that the most straightforward approach to accelerate the matrix computation is by approximating the whole matrix with one of the low-rank approximation techniques that we have described in Section 3.1.2, most matrices that arise in practical problems have slow decaying singular values [51]. This means that a very high compression rank is required to obtain a decent accuracy up to the point that the low-rank approximation is not beneficial anymore, i.e. the matrices are dense (full-rank). Instead of doing that, the hierarchical low-rank approximation exploits the underlying low-rank property of such matrices by first applying a suitably chosen partition to the matrix so

that (numerically) low-rank submatrices appear, notably on the off-diagonal part of the matrix. For the sake of simplicity, let us focus our description on the case of symmetric matrix $A \in \mathbb{R}^{n \times n}$. For a more general explanation and possible extension to rectangular matrices, we refer the reader to [12, 51].

Let I and J be index sets such that $I = J = \{1, 2, \ldots, n\}$, and $\#I$ denotes the cardinality of an index set I, which in this case $\#I = \#J = n$. Consequently, any submatrix of A should correspond to a certain block $\tau \times \sigma$ where τ and σ are row and column index sets, respectively. Some examples are:

- The submatrix $a_{1:4,1:4}$ corresponds to the block $\tau \times \sigma$ such that $\tau = \sigma = \{1, 2, 3, 4\}$.
- The submatrix $a_{1:4,5:8}$ corresponds to the block $\tau \times \sigma$ such that $\tau = \{1, 2, 3, 4\}$ and $\sigma = \{5, 6, 7, 8\}$.
- The submatrix $a_{5:8,1:4}$ corresponds to the block $\tau \times \sigma$ such that $\tau = \{5, 6, 7, 8\}$ and $\sigma = \{1, 2, 3, 4\}$.

According to [51], for the hierarchical low-rank approximation to be efficient, a partition P of $I \times J$ should be constructed in a way such that the following properties hold.

1. The partition P should contain as few blocks as possible since the storage cost is proportional to the number of blocks.
2. The blocks within P must be as large as possible. Furthermore, there is a minimal block size since approximating a block $\tau \times \sigma$ by a compression rank r is most beneficial when $r \ll min\{\#\tau, \#\sigma\}$.
3. The blocks within P must be small enough to ensure that the desired accuracy can be obtained with a possibly small rank r. If the blocks are too large, the approximation may require a large rank.
4. The block structure of the partition must be constructed such that all matrix operations can be performed easily.

Note that the second and third conditions are opposing each other since small storage cost and high approximation accuracy cannot be realized simultaneously. Therefore, it is important to find a good balance between these two, i.e. a satisfactory trade-off between storage cost and accuracy, using the *admissibility condition* that we will explain in Section 3.3. The fourth condition rejects a block partition that would hamper certain matrix operations. For example, a non-regular pattern where the row and column indices do not match with each other would bring forth substantial difficulty in performing matrix-matrix multiplication. A suitable block structure is usually found using hierarchical construction, which we will explain in the following sections.

3.2.1 Index Tree

The *index tree* provides a way to hierarchically partition an index set I. A common way is to partition using the *binary tree*, such that the root node contains the whole index set, and at each step, a node is evenly split into two children nodes, each containing half of the parent's indices that do not overlap with each other. This process of splitting is then repeated recursively for each child until a certain minimum size is reached and the node is not subdivided anymore. Such nodes are referred to as the *leaf nodes*. Note that in practice there are other schemes that use *quadtree* or even *octree*. Sometimes the indices within a node are not evenly split but divided according to some underlying geometric information of the indices. However, here we assume a balanced binary tree splitting because it provides a simple and intuitive way of understanding the hierarchical subdivision.

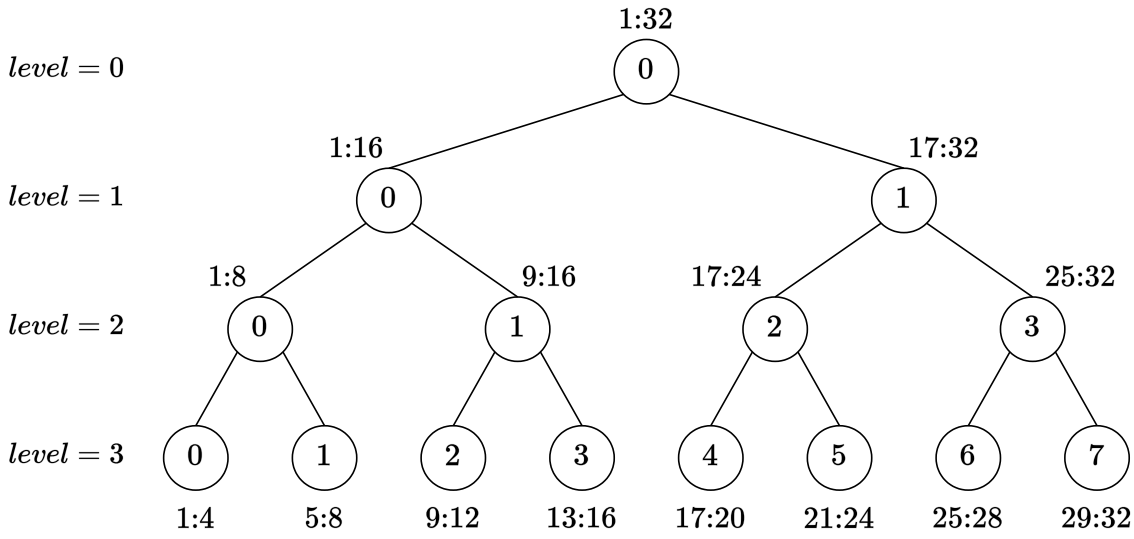


Figure 3.3: Example of a binary index tree over the index set 1:32

An example of a binary index tree is shown in Figure 3.3. The corresponding range of indices that a node is responsible for is written next to the node (outside of the node). It is clear that the lower the level is, which corresponds to a larger level number, the finer the partition becomes. In this example we assume a minimum block size of 4, resulting in a 3-level subdivision such that each leaf node is responsible for 4 indices. The minimum block size is also referred to as the *leaf size* since it corresponds to the size of the leaf-level nodes. In addition, the number inside a node corresponds to the ordering of nodes relative to each level of the tree. We use these numbers to index the nodes within an index tree according to their level, which we will go over in the following.

Let $\mathcal{I}$ be an index tree for the index set $I = \{1{:}n\} = \{1, 2, \ldots, n\}$ with a prescribed leaf size of b such that $n/b = 2^L$. Several properties of this index tree is listed as follows.

- $\mathcal{I}$ contains $L+1$ levels, namely $l = 0, 1, \ldots, L$ such that $l = 0$ is the root level and $l = L$ is the leaf level.
- At every level l $(0 \leq l \leq L)$, there are 2^l nodes numbered from 0 to $2^l - 1$, each holding a subset $\tau \in I$ where $\#\tau = n/2^l$ and none of the nodes overlap each other.
- The k-th node at level l is denoted as $\mathcal{I}_{l;k}$.
- Every non-leaf node $\mathcal{I}_{l;k}$ has exactly two children: $\mathcal{I}_{l+1;2k}$ and $\mathcal{I}_{l+1;2k+1}$.
- Every node $\mathcal{I}_{l;k}$ (except the root node) has a parent which is $\mathcal{I}_{l-1;\lfloor k/2 \rfloor}$.
- All nodes in the same level form the original index set, i.e. for each level l,

$$\bigcup_{k=0}^{2^l-1} \mathcal{I}_{l;k} = I.$$

3.2.2 Cluster Tree

Let $\mathcal{I}, \mathcal{J}$ be index trees. The tensor product $\mathcal{T} \in \mathcal{I} \times \mathcal{J}$ is called a *cluster tree* that contains the combinations of nodes of $\mathcal{I}$ and $\mathcal{J}$. The root node of $\mathcal{T}$ contains the root nodes of $\mathcal{I}$ and $\mathcal{J}$. The children of a node in $\mathcal{T}$ contain unique combinations of the corresponding children from its $\mathcal{I}$ and $\mathcal{J}$ components. For example, let $(\mathcal{I}_{0;0}, \mathcal{J}_{0;0})$ be the root node of $\mathcal{T}$. If $\mathcal{I}_{0;0}$ has two children $\{\mathcal{I}_{1;0}, \mathcal{I}_{1;1}\}$ and $\mathcal{J}_{0;0}$ also has two children $\{\mathcal{J}_{1;0}, \mathcal{J}_{1;1}\}$, then the root node of $\mathcal{T}$ will have exactly four children, containing the pairings $\{(\mathcal{I}_{1;0}, \mathcal{J}_{1;0}), (\mathcal{I}_{1;0}, \mathcal{J}_{1;1}), (\mathcal{I}_{1;1}, \mathcal{J}_{1;0}), (\mathcal{I}_{1;1}, \mathcal{J}_{1;1})\}$. This implies that if the index sets $\mathcal{I}, \mathcal{J}$ are binary trees, the resulting cluster tree will be a quadtree.

Let $\mathcal{I} = \mathcal{J}$ be binary index trees with $n = 16$ and $b = 4$. An example of cluster tree $\mathcal{T} \in \mathcal{I} \times \mathcal{J}$ is shown in Figure 3.4. Here, each node of the tree holds a pair of index sets corresponding to the row and column indices. Similarly to the index tree, the pair of numbers written inside each node represents relative 2D ordering within every level. These numbers will be used to index the hierarchically subdivided block of a matrix.

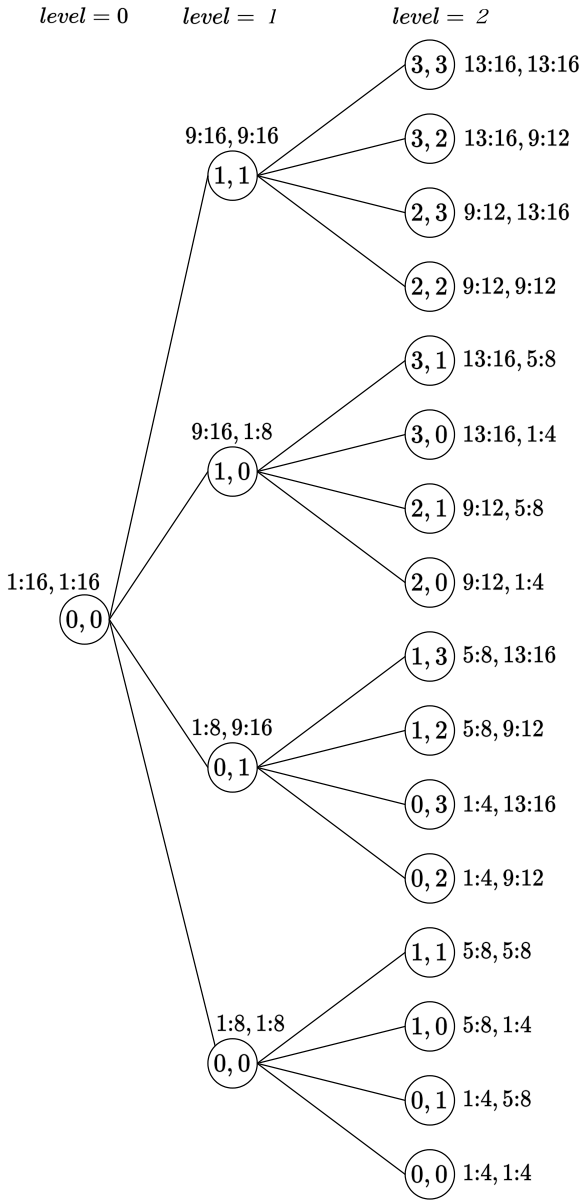


Figure 3.4: Example of a cluster tree with 2 levels of subdivision

3.3 Structure and Admissibility

The first step of the hierarchical low-rank approximation of a matrix $A \in \mathbb{R}^{n \times n}$ is to form the cluster tree over the row and column indices of A, which we have seen in Section 3.2.2. The leaf size that determines the depth of the subdivision tree has to be chosen carefully to obtain a good balance between storage cost and accuracy. This could be any number such that the index set is split at least once, but the choice of leaf size may also depend on things such as the underlying geometric information of the indices, the performance of low-level BLAS/LAPACK kernels for small matrices, etc.

Once the cluster tree is obtained, we apply it to hierarchically subdivide A, where the block corresponding to the (i, j) node at level l of the cluster tree is denoted as $A_{l;i,j}$. In other words, the block $A_{l;i,j}$ is a submatrix of A whose row indices correspond to the index set held by the node $\mathcal{I}_{l;i}$ and column indices correspond to the index set held by the node $\mathcal{J}_{l;j}$, that is

$$A_{l;i,j} = a_{\tau,\sigma}, \tag{3.25}$$

where $\tau \in I$ and $\sigma \in J$. For example, let $A \in \mathbb{R}^{16 \times 16}$. The hierarchical subdivision of A using the cluster tree $\mathcal{T}$ in Figure 3.4 is given in Figure 3.5. Here it is clear that each level of the cluster tree represents a full view of the matrix. One level of subdivision consists of dividing every block into a 2×2 equally sized sub-blocks. The larger the level number is, the finer the subdivision is.

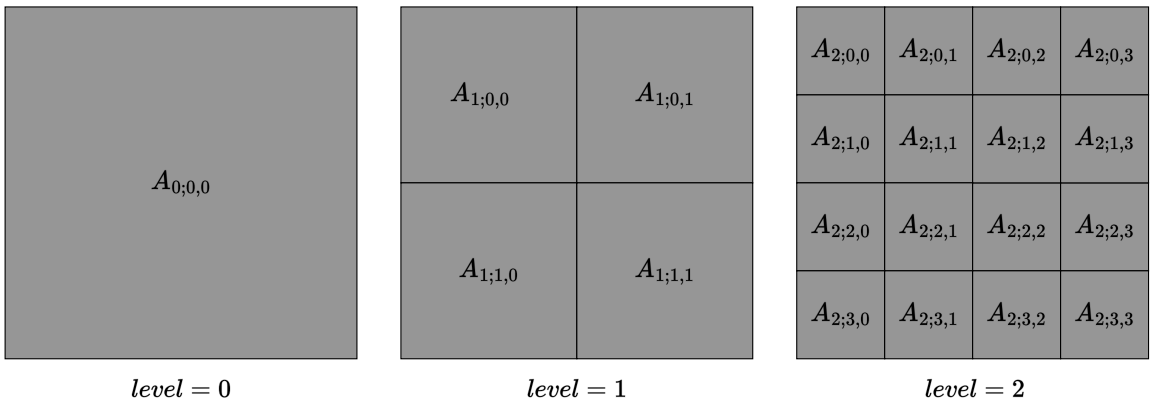


Figure 3.5: Example of hierarchical subdivision of a matrix using the cluster tree in Figure 3.4

The next step is to exploit the low-rank property within the subdivided blocks of A. The naive way is to perform a top-down traversal of the cluster tree from the root down to the leaf nodes and perform a low-rank approximation on each block corresponding to the node. If a block $\tau \times \sigma$ can be approximated by a small rank r (for example $r < (min\{\#\tau, \#\sigma\}/2)$) such that the desired accuracy is obtained, it is replaced by the low-rank approximation and all its children are deleted from the cluster tree. Otherwise, we traverse further down the tree and perform similar checks to the children blocks. When a leaf block is reached and it still requires a high compression rank to

reach the desired accuracy, it will be stored as a dense block. In other words, we keep subdividing the matrix until either a leaf block or a numerically low-rank block is reached. An example of the resulting hierarchical matrix from the subdivision corresponding to the cluster tree in Figure 3.4 is shown in Figure 3.6. In this case, the colored off-diagonal blocks are replaced by their low-rank approximations and they are not subdivided further, as can be seen in nodes $(0,1)$ and $(1,0)$ at level 1 whose children have been removed from the cluster tree.

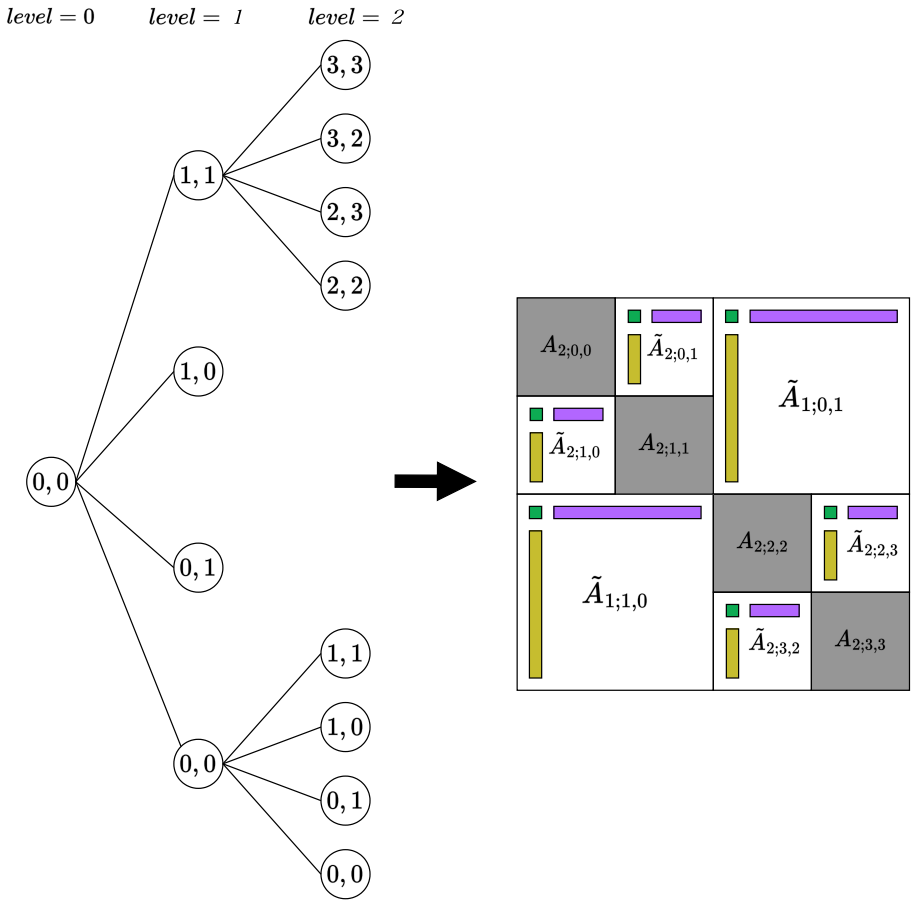


Figure 3.6: Example of hierarchical matrix obtained using the initial cluster tree in Figure 3.4

However, the process that we explained above is generally not used in practice, especially when the cluster tree contains many levels of subdivision. This is because of the expensive cost coming from the low-rank approximation that is performed on almost every block, leading to many discarded results in the end. Instead, if some sort of geometry information corresponding to the index sets is available, it is common to use that information to determine whether a block should be approximated or not without actually approximating it. This is referred to as the Boolean function called *admissibility condition*, which maps any pair of index sets $\tau \in I$ and $\sigma \in J$ such that

$$admissible : \tau \times \sigma \rightarrow \{true, false\}. \tag{3.26}$$

Therefore, the admissibility condition determines whether a matrix block should be approximated based on its corresponding row and column index sets that can be obtained from the underlying cluster tree. This way, a block $A_{l;i,j} = a_{\tau,\sigma}$ that satisfies the admissibility condition in Equation (3.26) is called an *admissible block*. Such blocks are stored in their low-rank approximated forms and are not subdivided further during the cluster tree traversal. On the other side, a matrix block that does not satisfy the admissibility condition is called an *inadmissible block*, where it is either subdivided further during the cluster tree traversal or stored in dense representation if it is a leaf block.

In general, there are two types of admissibility condition. One is the *weak admissibility*, where only blocks that lie on the diagonal are not admissible, that is

$$admissible(\tau,\sigma) = \begin{cases} true, & \text{if } \sigma \neq \tau \\ false, & \text{otherwise} \end{cases}. \tag{3.27}$$

Another one is called *strong admissibility*, sometimes also referred to as *standard admissibility*. Here, a block is determined to be admissible or not depending on the geometric information that is embedded within the row and column index sets. A typical example is when the matrix $A \in \mathbb{R}^{n\times n}$ represents the interaction between n particles within a system such that particles that have close proximity in space are indexed close to each other, and the entry $A_{i,j}$ denotes the interaction between the i-th and j-th particle. Let X be such a set of n particles and X_τ be a group of particles with indices according to the index set τ. Thus, the matrix block $A_{\tau,\sigma}$ denotes the interaction between two groups of particles X_τ and X_σ. In this case, according to [51], the strong admissibility condition is defined as

$$admissible(\tau,\sigma) = \begin{cases} true, & \text{if } min\{diam(X_\tau), diam(X_\sigma)\} \leq \eta\, dist(X_\tau, X_\sigma) \\ false, & \text{otherwise} \end{cases}, \tag{3.28}$$

where $\eta \in \mathbb{R}$ is a prescribed *admissibility constant*, $diam(X_\tau)$ and $diam(X_\sigma)$ are the values proportional to the spatial size of the groups X_τ and X_σ, respectively, and $dist(X_\tau, X_\sigma)$ is the spatial distance between X_τ and X_σ. A graphical illustration is shown in Figure 3.7.

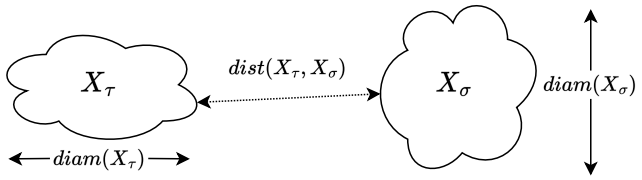


Figure 3.7: Groups of particles corresponding to index sets τ and σ

To put it simply, the strong admissibility condition determines whether a block should be compressed based on the distance between the interacting particles that it represents. In this sense, the interaction between two groups of particles that are well separated from each other in space

is assumed to be weak or data-sparse so that the corresponding matrix block can be well approximated using small rank r. Such admissible blocks often appear in the off-diagonal parts of the matrix because the blocks that are far from the diagonal represent interactions between groups of particles whose indices are far from each other. This can be easily visualized when the particles lie on a straight line such that the distance between two particles depends solely on the x-axis ordering, i.e. the index of the particle. On the other side, the interaction between groups that are close to each other in space is strong or dense so the corresponding matrix should be stored in dense representation. Such inadmissible blocks always appear on the main diagonal part since the diagonal block represents the interaction between particles that belong to the same group, which should be in close proximity. However, when the particles are positioned in 2-dimensional or 3-dimensional space, inadmissible blocks may also appear in the off-diagonal position. This is because the distance between particles also depends on the relative ordering in the y-axis and z-axis which may result in particles with significantly different index but are close to each other in space. In other words, the strong admissibility condition permits dense blocks not only on the diagonal but also on the off-diagonal parts of the matrix. This makes it usually preferred when the matrix originates from particles that lie in 2-dimensional and 3-dimensional space since using weak admissibility on such problems often leads to a high compression rank of the off-diagonal blocks.

3.4 Structured Low-Rank Matrix Formats

The hierarchical low-rank approximation can produce different kinds of structured low-rank formats depending on the imposed block structure, admissibility condition, and storage representation of the admissible blocks. These formats represent an approximation of the original dense matrix where only the admissible blocks are expressed in their low-rank approximation form. Therefore, the overall accuracy of these formats depends entirely on the low-rank approximation of the admissible blocks. Naturally, the accuracy increases as the compression rank r increases. On the other hand, a smaller compression rank leads to faster computation. Thus, there are two common ways to choose the compression rank of the admissible blocks:

1. Using a fixed rank r to approximate every admissible block. This allows for fast computation and easy implementation since the dimension of the low-rank approximations in the same level of the cluster tree is fixed. However, controlling the overall accuracy requires tuning the compression rank r since choosing a rank that is too small might lead to some blocks not being represented accurately.

2. Using a fixed error threshold ϵ to approximate every admissible block. This means that the compression rank may vary between blocks depending on how the singular values decay. The

overall compression accuracy can then be controlled, but the performance might suffer if the rank for some blocks becomes too large than the others.

The compression scheme of choice typically depends on several factors. For example in certain problems where the rank of the admissible blocks is known in advance, or when load balance of the computations involving admissible blocks is highly desired, fixed rank compression may be a good choice. Otherwise, the fixed accuracy scheme is commonly used since well-defined error control is desirable in many problems. When using fixed accuracy compression scheme, the overall error induced by the structured low-rank format is usually bounded by

$$\frac{\left\|A - \tilde{A}\right\|_F}{\|A\|_F} \leq \epsilon, \tag{3.29}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, A is the original structured low-rank matrix, $\tilde{A}$ is the hierarchical low-rank approximation of A, and ϵ is the relative error threshold. A similar bound can also be derived for absolute error threshold. For error bound with respect to other types of matrix norm, a mathematically rigorous analysis can be found in [51].

In this section, we discuss several well-known structured low-rank formats along with their advantages and disadvantages. It is important to note that since this thesis focus on the matrix decomposition of structured low-rank formats, we do not provide detailed a discussion about the construction scheme. For that, we refer the reader to the textbooks [12, 51] and some other works that focus on hierarchical matrix constructions [23, 19, 90, 25].

3.4.1 Block Low-Rank (BLR) Matrices

The block low-rank format was introduced in [5] as a specialization of hierarchical matrices that performs 1 level of subdivision that splits the root level directly into the finest level. In terms of the index tree $\mathcal{I}$, this amounts to splitting the root index set $I = J = \{1, 2, \ldots, n\}$ into sub-index sets of uniform size b. This is also equivalent to flattening the binary index tree into a fixed depth, bringing the leaf-level nodes into the level right below the root (level=1). An example is given in Figure 3.8.

The resulting cluster tree $\mathcal{T} \in \mathcal{I} \times \mathcal{J}$ produces a flat subdivision of the matrix A into $p \times q$ square blocks such that

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,q-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,q-1} \\ \vdots & \cdots & \cdots & \vdots \\ A_{p-1,0} & A_{p-1,1} & \cdots & A_{p-1,q-1} \end{bmatrix}, \tag{3.30}$$

where $p = q = n/b$ and $A_{i,j} \in \mathbb{R}^{b \times b}$. Note for flat subdivision like this, we omit the level l since the resulting cluster tree always have two levels. Then, approximating the admissible low-rank blocks

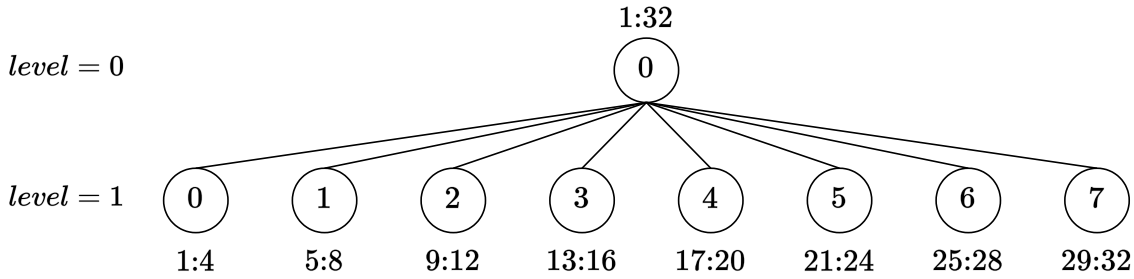


Figure 3.8: Flattened binary index tree in Figure 3.3

produces the block low-rank matrix

$$\tilde{A} = \begin{bmatrix} \tilde{A}_{0,0} & \tilde{A}_{0,1} & \cdots & \tilde{A}_{0,q-1} \\ \tilde{A}_{1,0} & \tilde{A}_{1,1} & \cdots & \tilde{A}_{1,q-1} \\ \vdots & \cdots & \cdots & \vdots \\ \tilde{A}_{p-1,0} & \tilde{A}_{p-1,1} & \cdots & \tilde{A}_{p-1,q-1} \end{bmatrix}, \tag{3.31}$$

where

$$\tilde{A}_{i,j} = \begin{cases} A_{i,j} & (A_{i,j} \text{ is not admissible}) \\ U_{i,j} S_{i,j} V_{i,j}^T & (A_{i,j} \text{ is admissible}) \end{cases} \Bigg| 0 \leq i < p,\ 0 \leq j < q \tag{3.32}$$

and $U_{i,j} S_{i,j} V_{i,j}^T$ is the low-rank approximation of $A_{i,j}$. An example of BLR matrix with weak admissibility is shown in Figure 3.9, where the off-diagonal blocks are represented in their low-rank approximated form. Nonetheless, strong admissibility is also applicable to this format.

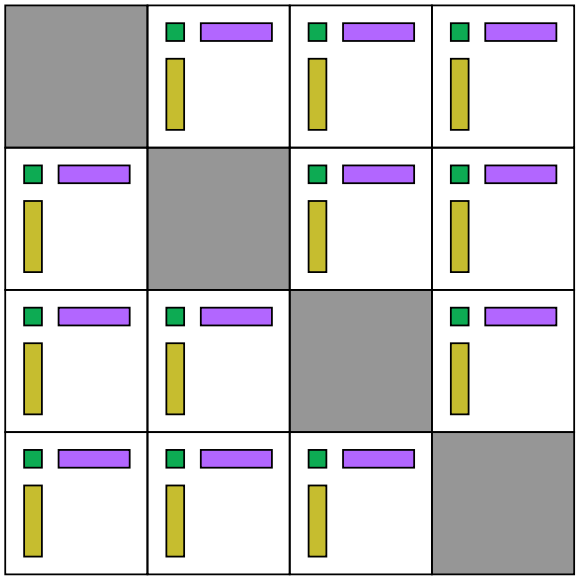


Figure 3.9: Example of a BLR matrix with weak admissibility

The simple flat-blocked structure of this format leads to several advantages. First, since all the blocks are on the same level, the matrix operations can be processed in a block-wise fashion without

the need for recursion. This greatly reduces implementation challenges. Second, blocked variants of traditional matrix algorithms are widely available in the literature, and their implementations (such as LAPACK [8]) are known to perform better than the unblocked counterparts in modern CPU architectures. Therefore, the matrix operations on BLR matrices can be performed using similar procedures since they share the same block structure, with an additional extension to handle the low-rank admissible blocks. Third, in the context of parallel computation, resolving dependencies between operations is easy because there are no hierarchical dependencies that arise during the matrix computation. Lastly, load balance is not difficult to achieve since all blocks are of similar size, or even better when they have the same size.

However, the major disadvantage of the simple block structure is that the computational and storage cost for matrix operations on BLR matrices are generally more expensive than the hierarchical formats that will be introduced later. Due to the flat blocking scheme, the choice of block size b plays a crucial role in determining the overall computational and storage cost. A fairly good choice of $b = \sqrt{n}$ was proposed in [6] so that constructing the BLR matrix costs $\mathcal{O}(n^2)$ flops, which takes $\mathcal{O}(n^{1.5})$ storage. Moreover, operations like matrix-matrix multiplication, LU, and QR decomposition also require $\mathcal{O}(n^2)$ flops. Even though this is faster than the $\mathcal{O}(n^3)$ cost of the dense counterpart, it is still significantly more expensive than the (near) linear cost achieved by the hierarchical formats.

3.4.2 Block Low-Rank with Shared Bases (BLR^2) Matrices

The concept of shared bases has been introduced in the context of $\mathcal{H}^2$ matrices [48, 12]. This technique was studied with BLR format in [11], in which they came up with the name BLR^2 to describe such BLR format with shared bases. The block structure and admissibility are the same with BLR format, except that the admissible low-rank blocks are expressed as

$$\tilde{A}_{i,j} = U_i^S \; S_{i,j}^{SS} \; \left(V_j^S\right)^T, \tag{3.33}$$

such that U_i^S is the basis that is shared among all admissible blocks in the entire row i and V_j^S is the basis that is shared among all admissible blocks in the entire column j. This could greatly reduce the storage cost since only the small matrix $S_{i,j}$ needs to be maintained at each admissible block. An example of a BLR^2 matrix with weak admissibility is shown in Figure 3.10, where the tall skinny matrices have been extracted out of the individual admissible blocks, leaving only the small $r \times r$ matrix (compare with Figure 3.9).

The investigation in [11] suggests that under the weak admissibility condition, the rank of the shared bases becomes prohibitively large with respect to the block size b so that the ordinary BLR format is more efficient. However, under the strong admissibility condition such that the admissible blocks are approximated with small rank r, the BLR^2 format manages to reduce the storage cost

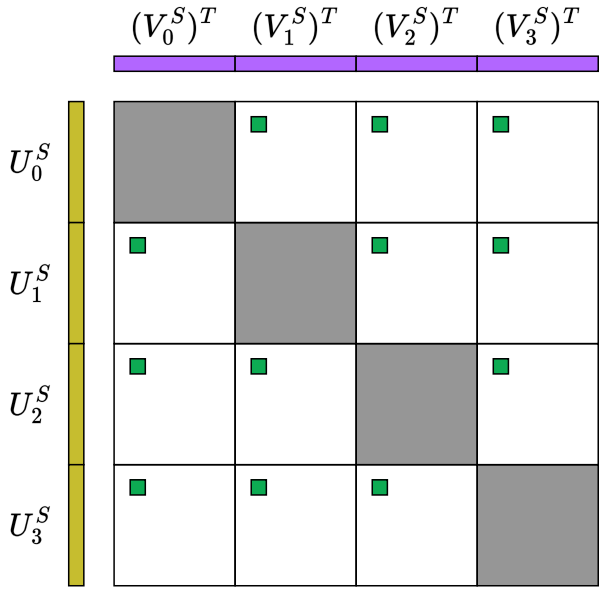


Figure 3.10: Example of a BLR^2 matrix with weak admissibility

of the BLR format from $\mathcal{O}(n^{3/2})$ down to $\mathcal{O}(n^{4/3})$. The computational cost of matrix operations is also reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^{9/5})$ flops.

The concept of shared bases can be related to the particle interactions example that we used when describing the admissibility condition in Section 3.3 (see the paragraphs near Figure 3.7). Using similar reasoning, we can summarize the outgoing interaction from the group i to all other groups that are well-separated from it via the shared basis U_i^S. The incoming interactions can also be summarized in a similar fashion via the shared basis V_j^S.

3.4.3 Hierarchically Off-Diagonal Low-Rank (HODLR) Matrices

Using the binary index and cluster tree (e.g. in Figure 3.6) with the weak admissibility condition in Equation (3.27), we obtain the simplest hierarchical representation called the HODLR format [4]. Unlike the BLR formats that we described in the previous sections, HODLR involves hierarchical subdivisions. An example is presented in Figure 3.11, where all off-diagonal blocks are admissible blocks stored in their low-rank approximated form.

As such, the HODLR matrix $\tilde{A}$ can be expressed as

$$\tilde{A} = \begin{bmatrix} \tilde{A}_{1;0,0} & \tilde{A}_{1;0,1} \\ \tilde{A}_{1;1,0} & \tilde{A}_{1;1,1} \end{bmatrix}, \tag{3.34}$$

where the off-diagonal blocks at each level are approximated as

$$\tilde{A}_{l;i,j} = U_{l;i,j} S_{l;i,j} V_{l;i,j}^T \tag{3.35}$$

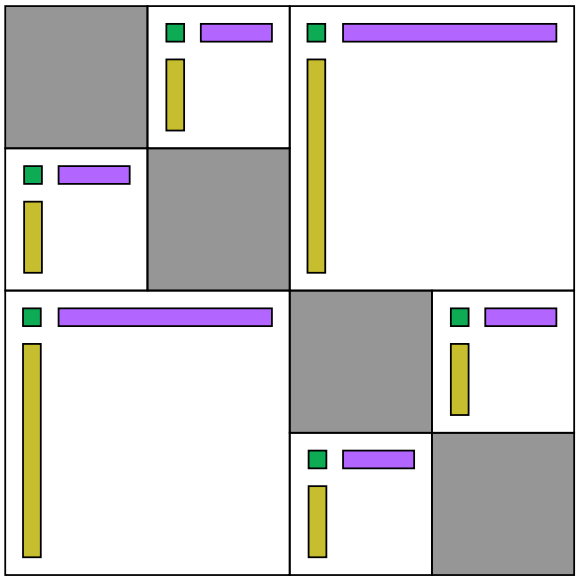


Figure 3.11: Example of a 2-level HODLR matrix

for $i \neq j$, and the inadmissible diagonal blocks are subdivided further as

$$\tilde{A}_{l;i,i} = \begin{bmatrix} \tilde{A}_{l+1;2i,2i} & \tilde{A}_{l+1;2i,2i+1} \\ \tilde{A}_{l+1;2i+1,21} & \tilde{A}_{l+1;2i+1,2i+1} \end{bmatrix}, \tag{3.36}$$

until the leaf level $l = L$ is reached where the subdivision is stopped and the diagonal blocks are stored in dense representation. With this kind of hierarchical structure, matrix algorithms will have to process the matrix recursively using a tree traversal technique. Consider the first level of the HODLR matrix in Equation (3.34). For example, if the LU decomposition of $\tilde{A}$ is to be computed, the algorithm would have to start from $\tilde{A}_{1;0,0}$ and recurse into the children blocks until the leaf level is reached before any of the other blocks in level 1 can be processed. After that, the off-diagonal block $\tilde{A}_{1;0,1}$ will be updated using a triangular solve operation, which involves another recursion to the children of $\tilde{A}_{1;0,0}$ in order to find the appropriate children blocks to update the corresponding parts of $\tilde{A}_{1;0,1}$. In the context of parallel computation, this gives rise to a hierarchical dependency between blocks of varying size across different levels, making the algorithm more complex and load balancing more difficult than the BLR format.

On the other side, the hierarchical subdivision allows the HODLR construction and typical matrix operations to be performed in $\mathcal{O}(n \log n)$ flops. The HODLR representation can also be stored in $\mathcal{O}(n \log n)$ storage. Moreover, the choice of leaf size b does not have to be dependent on the matrix dimension n to obtain the optimal computational and storage cost. This gives us more freedom in choosing the leaf size that yields the best balance between performance and accuracy.

3.4.4 Hierarchically Semi-Separable (HSS) Matrices

The HSS format was presented in [29], which is essentially a shared bases extension of the HODLR format with an addition of *nesting* the shared bases. The block structure and admissibility are the same as the HODLR format, except now the admissible block is written as

$$\tilde{A}_{l;i,j} = U^{big}_{l;i}\ S^{SS}_{l;i,j}\ \left(V^{big}_{l;j}\right)^T, \tag{3.37}$$

where

$$U^{big}_{l;i} = \begin{cases} U^S_{l;i} & \text{if } l = L \\ \begin{bmatrix} U^{big}_{l+1;2i} & \mathbf{0} \\ \mathbf{0} & U^{big}_{l+1;2i+1} \end{bmatrix} U^S_{l;i} & \text{otherwise} \end{cases} \tag{3.38}$$

$$V^{big}_{l;j} = \begin{cases} V^S_{l;j} & \text{if } l = L \\ \begin{bmatrix} V^{big}_{l+1;2j} & \mathbf{0} \\ \mathbf{0} & V^{big}_{l+1;2j+1} \end{bmatrix} V^S_{l;j} & \text{otherwise} \end{cases}. \tag{3.39}$$

Note that here U^{big} and V^{big} are the actual basis representing the low-rank approximation of the off-diagonal blocks, which are recursively constructed from the children's bases using the *transfer matrix* that projects the children's bases into their parent's basis. Hence, only the matrices $U^S_{l;i}$ and $V^S_{l;j}$ need to be stored, where they contain the actual shared bases at the leaf level and the transfer matrix at the non-leaf level. An example of an HSS matrix is shown in Figure 3.12, which has the same block structure as the HODLR example in Figure 3.11. For example, the approximated

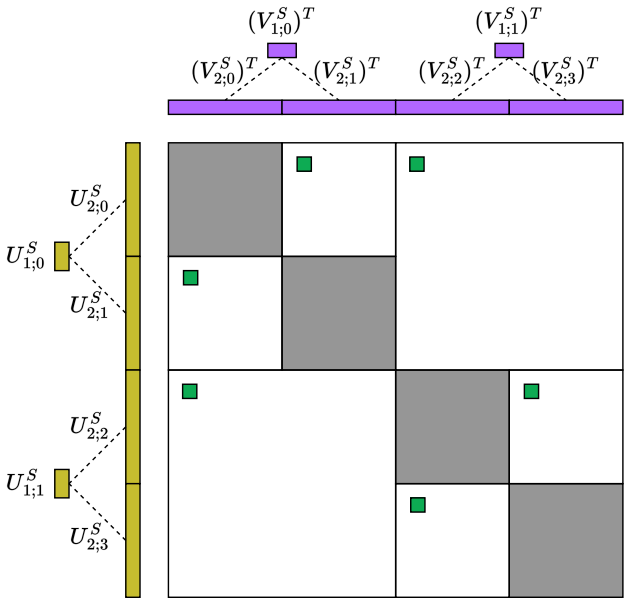


Figure 3.12: Example of a 2-level HSS matrix

off-diagonal block on the top right of Figure 3.12 is written as

$$\tilde{A}_{1;0,1} = \begin{bmatrix} U^S_{2;0} & \mathbf{0} \\ \mathbf{0} & U^S_{2;1} \end{bmatrix} U^S_{1;0} S^{SS}_{1;0,1} U^S_{1;1} \begin{bmatrix} U^S_{2;2} & \mathbf{0} \\ \mathbf{0} & U^S_{2;3} \end{bmatrix}. \tag{3.40}$$

The main advantage of the HSS format is that it reduces the storage cost down to $\mathcal{O}(n)$. This is attributed to the shared and nested bases property that allows the upper-level bases to be stored in a compact way using the children's bases and the transfer matrices so that only a small $r \times r$ matrix needs to be maintained at each admissible block. Moreover, the cost for matrix decomposition such as Cholesky and ULV decomposition, along with solving systems of linear equations with them are significantly reduced to $\mathcal{O}(n)$ flops [89]. However, constructing the HSS format generally costs $\mathcal{O}(n^2)$ flops, but this can be further reduced down to $\mathcal{O}(n \log n)$ with randomized techniques, for example with the method presented in [88].

3.4.5 $\mathcal{H}$-Matrices

The $\mathcal{H}$-matrix format, sometimes referred to as the Hierarchical matrix, is the extension of the HODLR format with strong admissibility conditions, for example using the one in Equation (3.28). The hierarchical block structure is similar to that of the HODLR matrix, but instead of only performing recursive subdivisions on the diagonal block, the off-diagonal block may also be subdivided further if it does not satisfy the admissibility condition. This makes the $\mathcal{H}$-matrix format flexible in handling a wide range of problems, even the ones that exhibit high-rank blocks in a small portion of the off-diagonal part. The computational and storage costs of construction and matrix operations are typically $\mathcal{O}(n \log n)$, under the assumption that the admissible blocks have a small numerical rank that does not grow with the matrix dimension n. An example of a 3-level $\mathcal{H}$-matrix is given in Figure 3.13, where it is clear that some off-diagonal blocks are stored in dense representation since they are not admissible.

However, it must be noted that while the $\mathcal{H}$-matrix is flexible and powerful in theory, it requires a deep understanding of the underlying problem that is represented by the matrix in order to be efficient. This is to ensure that the admissible blocks imposed by the strong admissibility condition are numerically low-rank so they can be well approximated using a small rank r. Such a process typically requires understanding the structure and geometric features of the problem to determine the optimal parameters such as the leaf size, admissibility constant, and a suitable permutation and grouping of the spatial objects (e.g. particles). These parameters may vary depending on the problem class as there is no one solution that fits all problems. If the admissible blocks are not numerically low-rank, their compression rank will become too large and negatively affect the performance. Therefore, domain expertise is essential in order to employ $\mathcal{H}$-matrix efficiently. Some good examples in the context of solving integral and partial differential equations can be found in [12, 51].

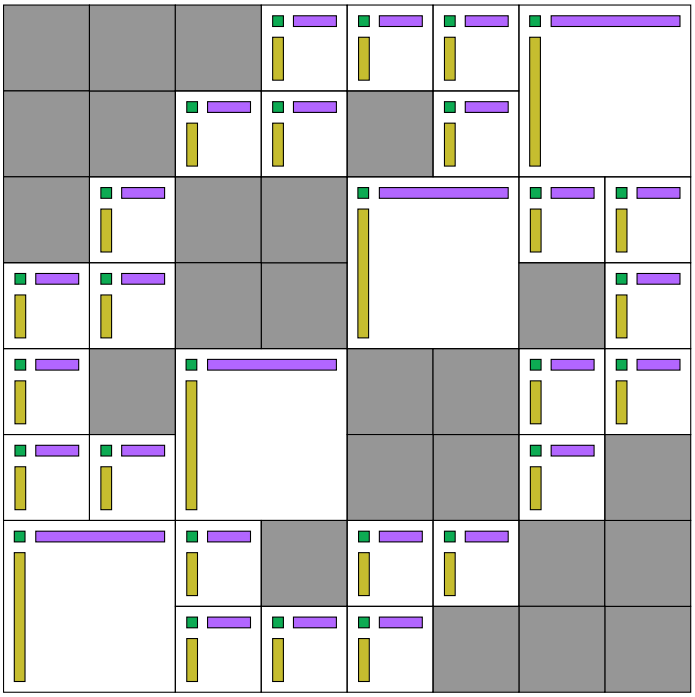


Figure 3.13: Example of a 3-level $\mathcal{H}$-matrix

3.4.6 $\mathcal{H}^2$-Matrices

The $\mathcal{H}^2$-matrix format can be regarded as the extension of the HSS format to strong admissibility conditions. It uses the same block structure as the HSS, except that the off-diagonal blocks can be recursively subdivided too if they do not satisfy the admissibility condition, similar to the $\mathcal{H}$-matrix format. The admissible low-rank blocks are expressed using shared and nested bases so that the storage cost is significantly reduced to $\mathcal{O}(n)$ compared to the $\mathcal{H}$-matrix that requires $\mathcal{O}(n \log n)$. Furthermore, the computational cost of matrix operations such as matrix-vector multiplication and matrix decomposition is reduced down to $\mathcal{O}(n)$ flops. Compared to the other formats that we have described so far, the $\mathcal{H}^2$-matrix is the most memory and compute-efficient representation. They are particularly known to work very well for finding the numerical solution of integral and elliptic partial differential equations [51]. An example of $\mathcal{H}^2$ matrix with 3 levels of subdivision is shown in Figure 3.14, which has the same block structure and admissibility as Figure 3.13 with the tall skinny matrices extracted out of the individual admissible block to be shared and nested among the blocks in the same row/column.

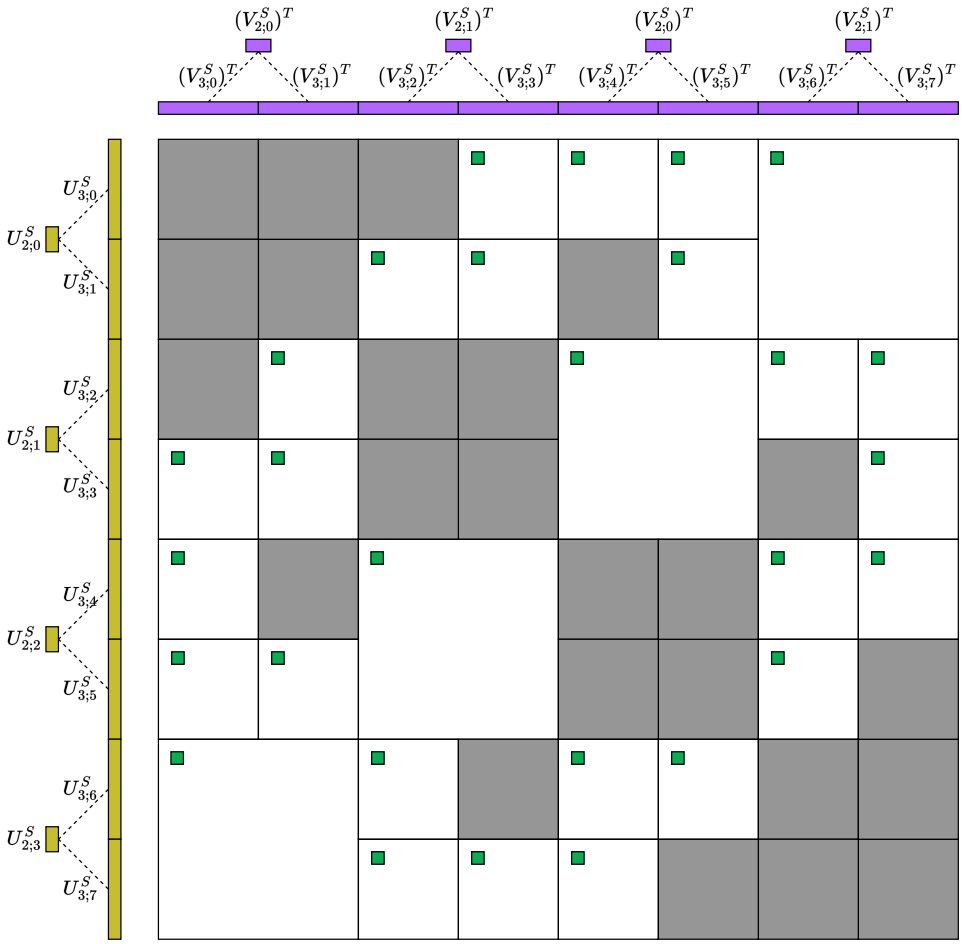


Figure 3.14: Example of a 3-level $\mathcal{H}^2$-matrix

Chapter 4

Block Low-Rank QR Decomposition for Eigenvalue Problems

Although the storage and computational costs associated with the BLR format are higher than the more advanced hierarchical formats, its simple and flat block structure has many favorable consequences. The simplicity and flexibility of the BLR format make it easy to use in the context of a general-purpose algebraic solver [7]. It is also known to be efficient on parallel computers [57, 3, 31, 76]. Moreover, it has been shown in [54] that matrix-vector multiplication based on BLR matrices is significantly faster than hierarchical matrices for a large number of processes. For these reasons, the BLR format is still competitive and may be a better choice, at least for some problem classes and sizes [7].

In recent years, BLR matrix decomposition has generated considerable research interest. Cholesky-based direct solvers using the BLR format have been utilized for weather modeling [2] and PASTIX supernodal solver [72] on multicore architectures. They have also been used in large-scale computation on distributed memory systems for seismic and electromagnetic [77] and geospatial statistics problems [3, 26]. However, the QR factorization, which has a wide range of usage in solving scientific and engineering problems including the eigenvalue problem, has not been studied very well with the BLR format. To the best of our knowledge, the only existing work is [55], where they studied the Gram-Schmidt orthogonalization with BLR-matrix arithmetic to perform QR factorization efficiently on distributed memory systems. Even so, the method still relies on the traditional fork-join approach that has a relatively large synchronization overhead. It may also suffer from numerical instability as it inherits the property of Gram-Schmidt iteration.

In this chapter, we present two new algorithms for the QR factorization of BLR matrices: one that performs block-column-wise QR based on the blocked Householder method [43], and another one that is based on the tiled QR [47]. Using the numerically stable Householder triangularization and BLR-matrix arithmetic, the block-column-wise algorithm achieves a theoretical complexity of

$\mathcal{O}(mn)$, while the tiled BLR-QR exhibits $\mathcal{O}(mn^{1.5})$ complexity. Nonetheless, the tiled BLR-QR has finer granularity that allows for parallel task-based execution on shared memory systems. This leads to an out-of-order execution with very loose synchronization compared to the fork-join model. Furthermore, we applied our proposed algorithm to accelerate the computation of eigenvalues using the QR algorithm.

The rest of this chapter is organized as follows. We start by summarizing the well-known blocked methods for QR decomposition of dense matrices in Section 4.1. Then we explain our proposed BLR-QR algorithms as well as the existing Gram-Schmidt-based method in Section 4.2, followed by their parallelization in Section 4.3. Section 4.4 summarizes the application of our proposed methods to accelerate the QR eigenvalue solver. Finally, we present the numerical results in Section 4.5. The majority of the content in this chapter has been published in [9].

4.1 Blocked QR Decomposition

In this section, we summarize the standard blocked and tiled methods for computing the QR decomposition of dense matrices. Although they have a similar cost of $O(mn^2)$ flops as the traditional unblocked version that we have seen in Section 2.2, the blocked and tiled methods are known to be more efficient on modern supercomputers since they are rich in Level-3 BLAS operations that provide high performance on memory hierarchy systems [22].

To facilitate this, we assume that the matrix $M \in \mathbb{R}^{m \times n}$ is subdivided into $p \times q$ square blocks, each of size $b \times b$ where $b \in \mathbb{Z}$ is the chosen block size, $p = m/b$, and $q = n/b$. This is essentially the extension of the flat subdivision shown in Equation (3.30) to a rectangular matrix with square blocks. Extension to the rectangular block is possible with extra permutation steps, e.g. as discussed in [59], which we do not discuss further here.

4.1.1 Blocked Modified Gram-Schmidt QR

The blocked version of the modified Gram-Schmidt QR was first introduced in [56]. Let $M = [M_0\ M_1]$ such that M_0 and M_1 are the block-columns of M. Then, the QR decomposition of M can also be expressed as

$$\begin{bmatrix} M_0 & M_1 \end{bmatrix} = \begin{bmatrix} Q_0 & Q_1 \end{bmatrix} \begin{bmatrix} R_{0,0} & R_{0,1} \\ \mathbf{0} & R_{1,1} \end{bmatrix}.$$

Thus, M can be orthogonalized by the following steps:

1. *Orthogonalize block-column* $M_0 = Q_0 R_{0,0}$.
2. *Compute* $R_{0,1} = Q_1^T M_1$.
3. *Update* $M_1 \leftarrow M_1 - Q_0 R_{0,1}$.

4. *Orthogonalize block-column* $M_1 = Q_1 R_{1,1}$.

Note that steps 1 and 4 can be done using the unblocked modified Gram-Schmidt method as in Algorithm 2. These steps can be extended for arbitrary number of block columns, as shown in Algorithm 18.

Algorithm 18: Blocked modified Gram-Schmidt (MGS) QR decomposition

Input: M with $p \times q$ blocks
Output: Q with $p \times q$ blocks and R with $q \times q$ blocks such that $M = QR$

```
for j = 0 to q − 1 do
    [Q_j, R_{j,j}] = QR(M_j)
    for k = j + 1 to q − 1 do
        R_{j,k} = Q_j^T M_k
        M_k ← M_k − Q_j R_{j,k}
    end
end
```

4.1.2 Blocked Householder QR

The blocked version of the Householder QR decomposition was introduced in [43]. The basic idea is to reorganize the computation by applying the Householder transformations in a cluster of columns at a time, i.e. triangularizing one block-column at a time. As an example, let M be partitioned into a 3×2 block matrix such that

$$M = \begin{bmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \\ M_{2,0} & M_{2,1} \end{bmatrix}, \tag{4.1}$$

where $M_{i,j} \in \mathbb{R}^{b \times b}$. Then, M can be triangularized as follows:

1. *Triangularize block-column* $\begin{bmatrix} M_{0,0} \\ M_{1,0} \\ M_{2,0} \end{bmatrix} = \hat{Q}_0 \begin{bmatrix} R_{0,0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}$
2. *Apply block-column reflector* $\begin{bmatrix} R_{0,1} \\ M_{1,1} \\ M_{2,1} \end{bmatrix} \leftarrow \hat{Q}_0^T \begin{bmatrix} M_{0,1} \\ M_{1,1} \\ M_{2,1} \end{bmatrix}$
3. *Triangularize block-column* $\begin{bmatrix} M_{1,1} \\ M_{2,1} \end{bmatrix} = \hat{Q}_1 \begin{bmatrix} R_{1,1} \\ \mathbf{0} \end{bmatrix}$

This produces the QR decomposition of M in the form of

$$\begin{bmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \\ M_{2,0} & M_{2,1} \end{bmatrix} = Q \begin{bmatrix} R_{0,0} & R_{0,1} \\ \mathbf{0} & R_{1,1} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{4.2}$$

The orthogonal factor Q can be constructed as

$$Q = \hat{Q}_0 \begin{bmatrix} I & \mathbf{0} \\ \mathbf{0} & \hat{Q}_1 \end{bmatrix}. \tag{4.3}$$

Algorithm 19 shows the generalization of blocked Householder QR based on steps (1)-(3) above for an arbitrary number of blocks. The QR decomposition in line 2 is performed using the unblocked Householder QR decomposition in Algorithm 4 so that the intermediate orthogonal matrices $\hat{Q}_k$ are stored implicitly using Y and T matrices. Figure 4.1 shows the operations in Algorithm 19 performed on a dense matrix with 3×3 blocks. Thick borders show the tiles that are being read at each operation. Red and orange colors denote the tiles that are being written at each of the corresponding operation written on top of the matrices.

Algorithm 19: Blocked Householder QR decomposition

Input: M with $p \times q$ blocks

Output: Y, R with $p \times q$ blocks and T with $1 \times q$ blocks such that R is upper triangular and Y, T contain intermediate orthogonal factors

1 **for** $k = 0$ **to** $q - 1$ **do**

2 $\quad$ QR $\left(\begin{bmatrix} M_{k,k} \\ M_{k+1,k} \\ \vdots \\ M_{p-1,k} \end{bmatrix} \right) = \hat{Q}_k \begin{bmatrix} R_{k,k} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}$, such that $\hat{Q}_k = I - \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix} T_k \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix}^T$

3 $\quad$ **for** $j = k + 1$ **to** $q - 1$ **do**

4 $\quad\quad$ Update $\begin{bmatrix} R_{k,j} \\ M_{k+1,j} \\ \vdots \\ M_{p-1,j} \end{bmatrix} \leftarrow \hat{Q}_k^T \begin{bmatrix} M_{k,j} \\ M_{k+1,j} \\ \vdots \\ M_{p-1,j} \end{bmatrix}$

5 $\quad$ **end**

6 **end**

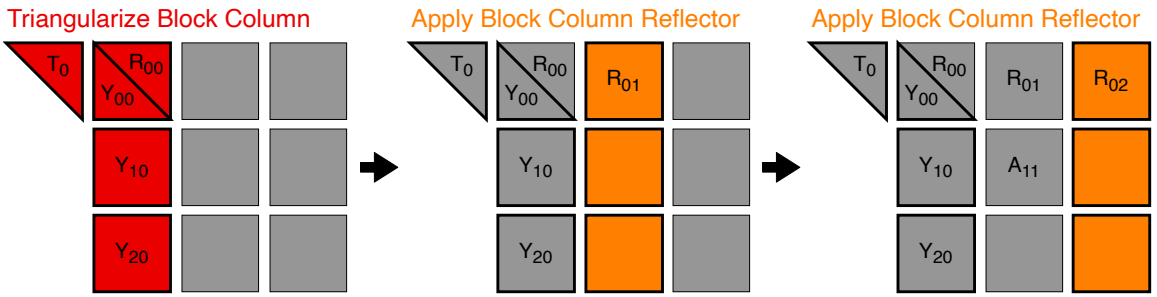


Figure 4.1: Triangularization of the first block-column of a dense matrix with 3×3 blocks using blocked Householder method

Furthermore, since the Householder triangularization ultimately amounts to multiplication by Q^T from the left, multiplication with Q can be done by performing similar steps to that of the decomposition in the correct order and transposition. Algorithm 20 shows the method to multiply a matrix C by Q from the left, which has about the same cost as Algorithm 19.

Algorithm 20: Left multiplication by Q from blocked Householder algorithm

Input: Y with $p \times q$ blocks, T with $1 \times q$ blocks, C with $p \times q$ blocks
Output: $C \leftarrow QC$

for $k = q - 1$ ***downto*** 0 **do**
  for $j = q - 1$ ***downto*** k **do**
    Update $\begin{bmatrix} C_{k,j} \\ C_{k+1,j} \\ \vdots \\ C_{p-1,j} \end{bmatrix} \leftarrow \left(I - \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix} T_k \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix}^T \right) \begin{bmatrix} C_{k,j} \\ C_{k+1,j} \\ \vdots \\ C_{p-1,j} \end{bmatrix}$
  end
end

4.1.3 Tiled Householder QR

Gunter and Van De Geijn further decomposed the operations of blocked Householder QR further to obtain the tiled Householder method in [47]. Note that we use the term *tiled* here to distinguish against the *blocked* Householder variant, in the sense that the tiled variant operates on tiles, i.e. square blocks, whereas the blocked variant operates on the block-columns, which most of the time are rectangles instead of squares.

The tiled Householder method takes its root in the updating factorization technique that is described in [43, 79]. From Equation (4.1), the triangularization of M can also be done by the following steps:

1. *Triangularize diagonal block* $M_{0,0} = \hat{Q}_{0,0} R_{0,0}$
2. *Apply block reflector* $R_{0,1} = \hat{Q}_{0,0}^T M_{0,1}$
3. *Update QR factorization* $\begin{bmatrix} R_{0,0} \\ M_{1,0} \end{bmatrix} = \hat{Q}_{1,0} \begin{bmatrix} {R_{0,0}}' \\ \mathbf{0} \end{bmatrix}$, which zeroes $M_{1,0}$ and updates $R_{0,0}$
4. *Apply trapezoidal reflector* $\begin{bmatrix} R_{0,1} \\ M_{1,1} \end{bmatrix} \leftarrow \hat{Q}_{1,0}^T \begin{bmatrix} R_{0,1} \\ M_{1,1} \end{bmatrix}$
5. *Update QR factorization* $\begin{bmatrix} {R_{0,0}}' \\ M_{2,0} \end{bmatrix} = \hat{Q}_{2,0} \begin{bmatrix} {R_{0,0}}'' \\ \mathbf{0} \end{bmatrix}$, which zeroes $M_{2,0}$ and updates ${R_{0,0}}'$
6. *Apply trapezoidal reflector* $\begin{bmatrix} R_{0,1} \\ M_{2,1} \end{bmatrix} \leftarrow \hat{Q}_{2,0}^T \begin{bmatrix} R_{0,1} \\ M_{2,1} \end{bmatrix}$.
7. *Triangularize diagonal block* $M_{1,1} = \hat{Q}_{1,1} R_{1,1}$
8. *Update QR factorization* $\begin{bmatrix} R_{1,1} \\ M_{2,1} \end{bmatrix} = \hat{Q}_{2,1} \begin{bmatrix} {R_{1,1}}' \\ \mathbf{0} \end{bmatrix}$, which zeroes $M_{2,1}$ and updates $R_{1,1}$

Algorithm 21 shows the generalization of steps (1)-(8) for an arbitrary number of blocks. Analogous to the blocked Householder method, the construction of Q can also be done by performing similar steps in the correct order and transposition. Algorithm 22 shows the method to left-multiply by Q, which also has about the same cost as Algorithm 21.

Algorithm 21: Tiled Householder QR factorization

Input: M with $p \times q$ blocks

Output: Y, T, R with $p \times q$ blocks such that R is upper triangular and Y, T contain intermediate orthogonal factors

for $k = 0$ **to** $q - 1$ **do**
  QR $(M_{k,k}) = \hat{Q}_{k,k} R_{k,k}$, such that $\hat{Q}_{k,k} = I - Y_{k,k} T_{k,k} Y_{k,k}^T$
  for $j = k + 1$ **to** $q - 1$ **do**
    $R_{k,j} = \hat{Q}_{k,k}^T M_{k,j}$
  end
  for $i = k + 1$ **to** $p - 1$ **do**
    QR $\left(\begin{bmatrix} R_{k,k} \\ M_{i,k} \end{bmatrix} \right) = \hat{Q}_{i,k} \begin{bmatrix} R'_{k,k} \\ \mathbf{0} \end{bmatrix}$, such that $\hat{Q}_{i,k} = I - \begin{bmatrix} I \\ Y_{i,k} \end{bmatrix} T_{i,k} \begin{bmatrix} I \\ Y_{i,k} \end{bmatrix}^T$
    for $j = k + 1$ **to** $q - 1$ **do**
      $\begin{bmatrix} R_{k,j} \\ M_{i,j} \end{bmatrix} \leftarrow \hat{Q}_{i,k}^T \begin{bmatrix} R_{k,j} \\ M_{i,j} \end{bmatrix}$
    end
  end
end

Algorithm 22: Left multiplication by Q from tiled Householder algorithm

Input: C, Y, T with $p \times q$ blocks

Output: $C \leftarrow QC$

for $k = q - 1$ ***downto*** 0 **do**
  for $i = p - 1$ ***downto*** $k + 1$ **do**
    for $j = q - 1$ ***downto*** k **do**
      Update $\begin{bmatrix} C_{k,j} \\ C_{i,j} \end{bmatrix} \leftarrow \left(\begin{bmatrix} I \\ Y_{i,k} \end{bmatrix} T_{i,k} \begin{bmatrix} I \\ Y_{i,k} \end{bmatrix}^T \right) \begin{bmatrix} C_{k,j} \\ C_{i,j} \end{bmatrix}$
    end
  end
  for $j = q - 1$ ***downto*** k **do**
    Update $C_{k,j} \leftarrow (I - Y_{k,k} T_{k,k} Y_{k,k}^T) C_{k,j}$
  end
end

Notice that the tiled Householder QR is similar to the blocked Householder QR, except that the upper triangularization of a block column is decomposed into multiple, smaller operations involving at most two blocks at a time. A graphical illustration is shown in Figure 4.2, where thick borders show the tiles that are being read at each operation and the red, orange, green, and blue colors denote the tiles that are being written at each of the corresponding operation written on top of the matrices. This improves the granularity and data locality of the operations. However, this leads to larger storage requirements as we need to store more T matrices from the intermediate QR decomposition. On the contrary, the same approach to decompose block-column operation is difficult to apply to the blocked MGS method since orthogonalization needs all information of the column, so the blocking of operations can only be done in the column direction.

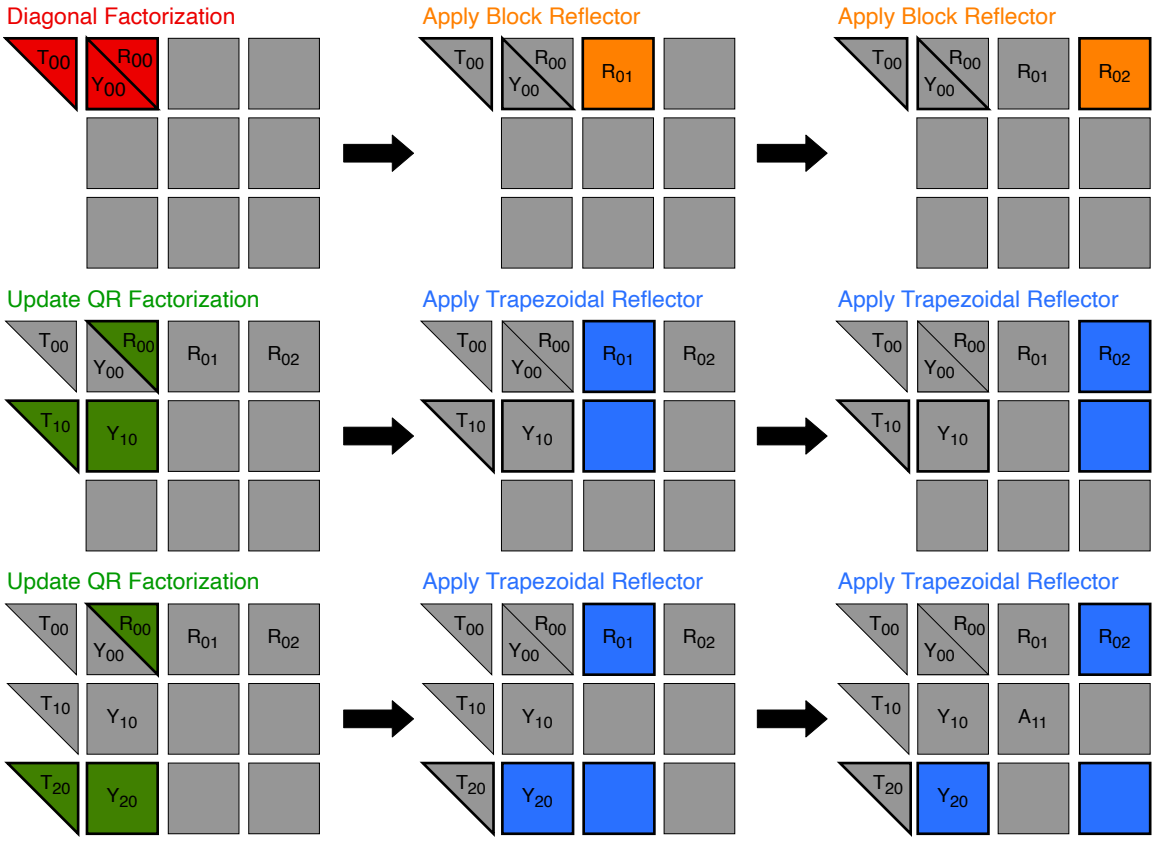


Figure 4.2: Triangularization of the first block-column of a dense matrix with 3×3 blocks using tiled Householder method

4.2 QR Decomposition of BLR Matrices

In this section we discuss the QR decomposition of BLR matrices. Consider a BLR matrix $\tilde{M} \in \mathbb{R}^{m \times n}$ with weak admissibility containing $p \times q$ blocks, each of size $b \times b$, which is expressed similarly

to that of Equation (3.31). The idea is to produce an approximate QR decomposition of the form

$$\tilde{M} \approx \tilde{Q}\tilde{R}, \tag{4.4}$$

such that:

- $\tilde{Q}$ and $\tilde{R}$ are BLR-matrices with the same block structure and admissibility as $\tilde{M}$
- The admissible blocks in $\tilde{Q}$ and $\tilde{R}$ are approximated using the same error threshold as the corresponding admissible block in $\tilde{M}$.

We first introduce an existing Gram-Schmidt based method presented in [55]. After that, we introduce our proposed algorithms based on the blocked and tiled Householder methods.

4.2.1 Blocked Modified Gram-Schmidt BLR-QR

Ida et al. in [55] have combined the blocked MGS method with BLR matrix arithmetic to formulate a BLR-QR decomposition algorithm. The algorithm proceeds in the same way as Algorithm 18 with some operations tailored to accommodate low-rank blocks that exist within a BLR matrix. Particularly, line 2 of Algorithm 18, which corresponds to the orthogonalization of a block column, is performed based on the method presented in [13]. Furthermore, matrix multiplications in line 4 and 5 are performed using BLR matrix arithmetic.

The QR decomposition of a block column is described as follows. First, let us write the block column $\tilde{M}_j$ $(0 \leq j < q)$ as

$$\tilde{M}_j = \begin{bmatrix} \tilde{M}_{0,j} \\ \tilde{M}_{1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} = \begin{bmatrix} \hat{U}_{0,j}\hat{V}_{0,j} \\ \hat{U}_{1,j}\hat{V}_{1,j} \\ \vdots \\ \hat{U}_{p-1,j}\hat{V}_{p-1,j} \end{bmatrix}, \tag{4.5}$$

where

$$\hat{U}_{i,j} = \begin{cases} I_b & (\tilde{M}_{i,j} \text{ is not admissible}) \\ U_{i,j} & (\tilde{M}_{i,j} \text{ is admissible}) \end{cases}, \; \hat{V}_{i,j} = \begin{cases} \tilde{M}_{i,j} & (\tilde{M}_{i,j} \text{ is not admissible}) \\ S_{i,j}V_{i,j}^T & (\tilde{M}_{i,j} \text{ is admissible}) \end{cases} \tag{4.6}$$

for $i = 0, 1, \ldots, p-1$. Note that we have just written each $\tilde{M}_{i,j}$ in left-orthogonal form. Now since each $\hat{V}_{i,j}$ is a dense block, we can write the matrix

$$B_j = \begin{bmatrix} \hat{V}_{0,j} \\ \hat{V}_{1,j} \\ \vdots \\ \hat{V}_{p-1,j} \end{bmatrix} \tag{4.7}$$

and perform the dense MGS QR decomposition $B_j = \tilde{Q}_j^B \tilde{R}_j^B$. Then we partition the matrix $\tilde{Q}_j^B$ back according to the subdivision of B_j in Equation (4.7). Since each $\hat{U}_{i,j}$ has orthonormal columns,

we can form the orthogonal factor by

$$\tilde{Q}_j = \begin{bmatrix} \hat{U}_{0,j} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \hat{U}_{1,j} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \hat{U}_{p-1,j} \end{bmatrix} \tilde{Q}_j^B = \begin{bmatrix} \hat{U}_{0,j}\tilde{Q}_{0,j}^B \\ \hat{U}_{1,j}\tilde{Q}_{1,j}^B \\ \vdots \\ \hat{U}_{p-1,j}\tilde{Q}_{p-1,j}^B \end{bmatrix}, \tag{4.8}$$

and upper triangular $\tilde{R}_{j,j} = \tilde{R}_j^B$ such that $\tilde{M}_j = \tilde{Q}_j \tilde{R}_{j,j}$, that is the QR decomposition of the block column $\tilde{M}_j$.

Assuming a block size $b = \mathcal{O}(\sqrt{n})$, this algorithm has a cost of $\mathcal{O}(mn)$ flops, which is smaller than the $\mathcal{O}(mn^2)$ cost of traditional MGS algorithm. We refer the reader to [55] for a more detailed explanation.

4.2.2 Blocked Householder BLR-QR

Our first proposed algorithm follows the blocked Householder dense QR in Section 4.1.2 to perform orthogonal triangularization of BLR-matrices. However, a BLR matrix contains admissible (low-rank) blocks in their factorized form that need to be handled in a different way than the dense blocks to yield an efficient algorithm. Thus, the operations need to be extended to handle these low-rank blocks. Our algorithm uses the steps from Algorithm 19 to produce the approximate QR decomposition, where some operations have been extended as follows.

4.2.2.1 Triangularization of Block Column

The first operation that we need to redefine is the QR decomposition of a block-column (line 2 of Algorithm 19). We adopt a similar approach to the one presented in [59]. Let us write the k-th block column $\tilde{M}_k$ $(0 \leq k < q)$ that needs to be triangularized as

$$\begin{bmatrix} \tilde{M}_{k,k} \\ \tilde{M}_{k+1,k} \\ \vdots \\ \tilde{M}_{p-1,k} \end{bmatrix} = \begin{bmatrix} \hat{U}_{k,k}\hat{V}_{k,k} \\ \hat{U}_{k+1,k}\hat{V}_{k+1,k} \\ \vdots \\ \hat{U}_{p-1,k}\hat{V}_{p-1,k} \end{bmatrix} \tag{4.9}$$

where $\hat{U}_{i,k}\hat{V}_{i,k}$ for $i = k, k+1, \ldots, p-1$ are the left-orthogonal forms of $\tilde{M}_{i,k}$ as defined in Equation (4.6). Next, we perform Householder triangularization to factorize the matrix

$$\begin{bmatrix} \hat{V}_{k,k} \\ \hat{V}_{k+1,k} \\ \vdots \\ \hat{V}_{p-1,k} \end{bmatrix} = \tilde{Q}_k^V \begin{bmatrix} \tilde{R}_{k,k} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}, \tag{4.10}$$

where

$$\tilde{Q}_k^V = I - \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix} T_k \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix}^T \tag{4.11}$$

such that $T_k \in \mathbb{R}^{b\times b}$ and $Y_{i,k}$ has the same dimension as $\hat{V}_{i,k}$ for $i = k, k+1, \ldots, p-1$. Since each $\hat{U}_{i,k}$ has orthonormal columns, we set $\tilde{Y}_{i,k} = \hat{U}_{i,k}Y_{i,k}$ to obtain the orthogonal factor

$$\hat{Q}_k = I - \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix} T_k \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix}^T . \tag{4.12}$$

It is important to note that $\tilde{Y}_{i,k}$ is a low-rank block if the corresponding $\tilde{M}_{i,k}$ is low-rank, otherwise it is dense.

To see why this works, let us define $W = diag(\hat{U}_{k,k}, \hat{U}_{k+1,k}, \ldots, \hat{U}_{p-1,k})$. Because diagonal blocks are dense, $W = diag(I, \hat{U}_{k+1,k}, \ldots, \hat{U}_{p-1,k})$. Multiplying $\hat{Q}_k^T$ to the block-column in Equation (4.9) yields

$$\begin{aligned} \hat{Q}_k^T \begin{bmatrix} \tilde{M}_{k,k} \\ \tilde{M}_{k+1,k} \\ \vdots \\ \tilde{M}_{p-1,k} \end{bmatrix} &= \left(I - W \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix} T_k^T \begin{bmatrix} Y_{k,k} \\ Y_{k+1,k} \\ \vdots \\ Y_{p-1,k} \end{bmatrix}^T W^T \right) W \begin{bmatrix} \hat{V}_{k,k} \\ \hat{V}_{k+1,k} \\ \vdots \\ \hat{V}_{p-1,k} \end{bmatrix} \\ &= W \left(\tilde{Q}_k^V \right)^T \begin{bmatrix} \hat{V}_{k,k} \\ \hat{V}_{k+1,k} \\ \vdots \\ \hat{V}_{p-1,k} \end{bmatrix} = W \begin{bmatrix} \tilde{R}_{k,k} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \tilde{R}_{k,k} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} . \end{aligned} \tag{4.13}$$

Thus, we have upper triangularized the k-th block-column and at the same time obtained the QR decomposition of it.

The cost of this operation is dominated by the QR decomposition in Equation (4.10). Each $\hat{V}_{i,k}$ has a small number of rows r ($\ll b$) if $\tilde{M}_{i,k}$ is low-rank. Under weak admissibility condition, this leads to dense QR decomposition of a $(b + (p-k-1)r) \times b$ matrix that costs $\mathcal{O}(b^3 + pb^2r)$ flops. The cost is similar under strong admissibility condition as long as the number of dense blocks in a block-column is bounded by a constant.

4.2.2.2 Apply Block Column Reflector

This operation corresponds to line 4 of Algorithm 19 where we multiply the resulting $\hat{Q}_k$ with a block-column. Let us write it as

$$\begin{aligned} \hat{Q}_k^T \begin{bmatrix} \tilde{M}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} &= \begin{bmatrix} \tilde{M}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} - \left(\begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix} T_k^T \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix}^T \begin{bmatrix} \tilde{M}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} \right) \\ &= \begin{bmatrix} \tilde{M}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} - \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix} \left[T_k^T \right] \left[\tilde{Y}_{k,k}^T \tilde{M}_{k,j} + \tilde{Y}_{k+1,k}^T \tilde{M}_{k+1,j} + \cdots + \tilde{Y}_{p-1,k}^T \tilde{M}_{p-1,j} \right] . \end{aligned} \tag{4.14}$$

Table 4.1: Operations inside one k-iteration ($0 \leq k < q$) of Algorithm 19 and their costs on a BLR matrix

Operation	**Complexity**	**Number of calls**
Triangularization of block-column	$b^3 + pb^2r$	1
Apply block column reflector	$b^2r + pbr^2$	$q - k - 1$

The process shown in Equation (4.14) consists of multiplication between dense and low-rank blocks, and accumulation by low-rank addition. Under the weak admissibility condition, this operation costs $\mathcal{O}(b^2r + pbr^2)$ flops. Under strong admissibility condition some block-columns require multiplication between dense blocks, leading to a cost of $\mathcal{O}(b^3 + b^2r + pbr^2)$ flops.

4.2.2.3 Algorithm and Cost Estimate

Figure 4.3 shows the operations in Algorithm 19 performed on a BLR matrix with 3×3 blocks under weak admissibility condition. Notice the difference between parts of the blocks that are modified compared to the dense counterpart in Figure 4.1.

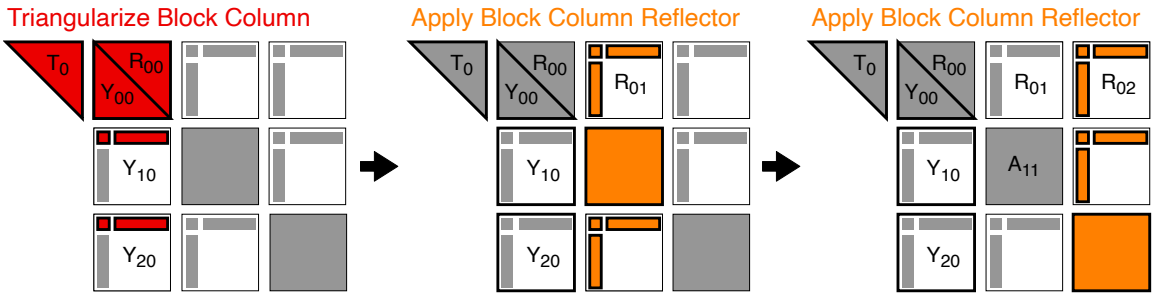


Figure 4.3: Triangularization of the first block-column of a BLR matrix with 3×3 blocks using blocked Householder method

In the following, we estimate the arithmetic complexity of our first algorithm under the weak admissibility condition. Table 4.1 shows the cost breakdown of one k-iteration in Algorithm 19 when applied on a BLR matrix. We get the total operation count by taking the sum for $k = 0, 1, \ldots, q-1$:

$$\begin{aligned} T_{blocked}(m,n,b) &= \sum_{k=0}^{q-1} b^3 + pb^2r + (q-k-1)(b^2r + pbr^2) \\ &= \frac{1}{2}\left(2nb^2 + 2mnr + n^2r + \frac{mn^2r^2}{b^2} - nbr - \frac{mnr^2}{b}\right). \end{aligned}$$

Setting $b = \mathcal{O}(\sqrt{n})$ yields a total cost of $\mathcal{O}(mn)$ flops. In terms of storage, this algorithm uses the existing space of $\tilde{M}$ plus extra space to store $T_0, T_1, \ldots, T_{q-1}$ matrices, each of size $b \times b$. This extra storage is not larger than the space for a BLR matrix, meaning that this algorithm requires $\mathcal{O}(m\sqrt{n})$ storage, which is similar to storage cost of a BLR matrix.

4.2.3 Tiled Householder BLR-QR

Our second algorithm is based on the tiled Householder dense QR explained in Section 4.1.3. It proceeds using the same steps as Algorithm 21 with some operations extended to handle low-rank block operands, which we explain in the following.

4.2.3.1 Diagonal Factorization

This operation corresponds to line 2 of Algorithm 21 where we perform Householder triangularization of a diagonal block

$$\tilde{M}_{k,k} = \hat{Q}_{k,k}\tilde{R}_{k,k}.$$

Since diagonal blocks are always dense, there is no need to handle low-rank blocks. This operation costs $\mathcal{O}(b^3)$ flops.

4.2.3.2 Apply Block Reflector

This operation corresponds to line 4 of Algorithm 21 where we multiply an off-diagonal block with orthogonal reflector, that is:

$$\tilde{R}_{k,j} = \hat{Q}_{k,k}^T\tilde{M}_{k,j}.$$

The off-diagonal block of a BLR matrix can be dense or low-rank. If the particular block is low-rank, this operation amounts to a multiplication of dense and low-rank block that costs $\mathcal{O}(b^2r)$ flops. Otherwise, the cost will be $\mathcal{O}(b^3)$ flops.

4.2.3.3 Update QR Factorization

This operation corresponds to line 7 of Algorithm 21 where we zero the off-diagonal block below $\tilde{M}_{k,k}$ using the QR decomposition

$$\mathrm{QR}\left(\begin{bmatrix} \tilde{R}_{k,k} \\ \tilde{M}_{i,k} \end{bmatrix}\right) = \hat{Q}_{i,k}\begin{bmatrix} \tilde{R}'_{k,k} \\ \mathbf{0} \end{bmatrix}.$$

The upper triangular $\tilde{R}_{k,k}$ comes from the QR decomposition of a diagonal block, so it is always a dense block. However $\tilde{M}_{i,k}$ can be a dense or low-rank block. In the case of a dense block, we simply concatenate the blocks and perform Householder dense QR on them. But if it is low-rank, we first perform dense QR factorization of

$$\begin{bmatrix} \tilde{R}_{k,k} \\ \tilde{M}^V_{i,k} \end{bmatrix} = \hat{Q}^V_{i,k}\begin{bmatrix} \tilde{R}'_{k,k} \\ \mathbf{0} \end{bmatrix}, \tag{4.15}$$

where

$$\tilde{Q}^V_{i,k} = I - \begin{bmatrix} I \\ Y_{i,k} \end{bmatrix} T_{i,k} \begin{bmatrix} I \\ Y_{i,k} \end{bmatrix}^T. \tag{4.16}$$

Since $U_{i,k}$ has orthonormal columns, we set $\tilde{Y}_{i,k} = U_{i,k}Y_{i,k}$ to obtain the orthogonal factor

$$\hat{Q}_{i,k} = I - \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix} T_{i,k} \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix}^T. \tag{4.17}$$

Table 4.2: Operations inside one k-iteration ($0 \leq k < q$) of Algorithm 21 and their costs on a BLR matrix

Operation	**Complexity**	**Number of calls**
Diagonal factorization	b^3	1
Apply block reflector	$b^2 r$	$q-k-1$
Update QR factorization	b^3	$p-k-1$
Apply trapezoidal reflector	$b^2 r$	$(p-k-1)(q-k-1)$

Note that this is a specialization of the operation explained in Section 4.2.2.1 where the block column is composed of a dense upper triangular block on top of an off-diagonal block. This step requires $\mathcal{O}(b^3)$ flops regardless of the type of $\tilde{M}_{i,k}$ due to the cost for generating $T_{i,k}$.

4.2.3.4 Apply Trapezoidal Reflector

This operation corresponds to line 9 of Algorithm 21 where we multiply the orthogonal reflector from Equation (4.17) to the corresponding blocks, that is

$$\begin{aligned}\begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} \leftarrow \hat{Q}_{i,k}^T \begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} &= \begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} - \left(\begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix} T_{i,k}^T \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix}^T \begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} \right) \\ &= \begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} - \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix} \left[T_{i,k}^T\right] \left[\tilde{R}_{k,j} + \tilde{Y}_{i,k}^T \tilde{M}_{i,j}\right].\end{aligned}$$

The cost of this operation depends on the blocks $\tilde{Y}_{i,k}$, $\tilde{R}_{k,j}$, and $\tilde{M}_{i,j}$. If $\tilde{R}_{k,j}$ is low-rank and at least one of $\{\tilde{Y}_{i,k}, \tilde{M}_{i,j}\}$ is low-rank, the cost is $\mathcal{O}(b^2 r)$ flops; Otherwise it is $\mathcal{O}(b^3)$ flops.

4.2.3.5 Algorithm and Cost Estimate

Figure 4.4 shows the operations that are performed inside one outer iteration of Algorithm 21 on a BLR matrix with 3×3 blocks under weak admissibility condition. Notice the difference between parts of the blocks that are operated compared to the dense counterpart in Figure 4.2.

In the following, we estimate the arithmetic complexity of our second algorithm under the weak admissibility condition. Table 4.2 shows the cost breakdown of one k-iteration of Algorithm 21 when applied on a BLR matrix. Summing up for $k = 0, 1, \ldots, q-1$ gets us the total operation count:

$$\begin{aligned}T_{tiled}(m,n,b) &= \sum_{k=0}^{q-1} (p-k)b^3 + (p-k)(q-k-1)b^2 r \\ &= \frac{1}{6}\left(6mnb + 3nb^2 + \frac{3mn^2 r}{b} - 3n^2 b - \frac{5n^3 r}{b} - 3mnr\right)\end{aligned}$$

For $b = \mathcal{O}(\sqrt{n})$, this algorithm costs $\mathcal{O}(mn^{1.5})$, requiring more flops than our first algorithm, the blocked Householder variant. It also produces more T matrices, which in total amounts to storing a lower trapezoidal $m \times n$ matrix. This leads to a storage requirement that grows similarly to that of dense Householder decomposition, i.e. $\mathcal{O}(mn)$. However, this algorithm has finer granularity that makes it more efficient for parallel computation, which will be shown in a later section.

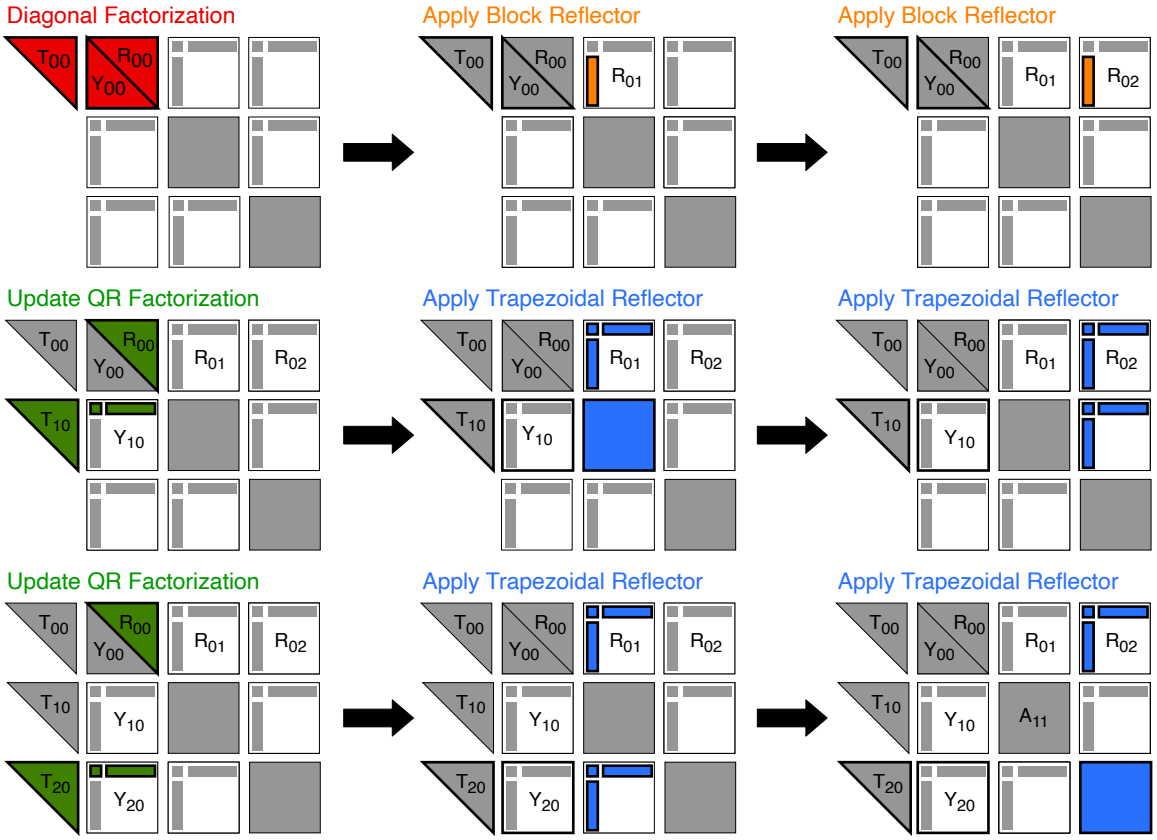


Figure 4.4: Triangularization of the first block-column of a BLR matrix with 3×3 blocks using tiled Householder method

4.3 Parallel QR Decomposition of BLR Matrices

The simple block structure of BLR matrices makes it very attractive for parallel computation. Here we discuss the parallelization of the QR decomposition techniques described in Section 4.2. We first recall the fork-join parallelization of the blocked modified Gram-Schmidt method that has been discussed in [55]. Then we use a similar fork-join approach to parallelize our proposed algorithms. Lastly, we present a fine-grained task-based parallel version of our tiled Householder BLR-QR decomposition.

4.3.1 Parallel Blocked Modified Gram-Schmidt BLR-QR

Algorithm 18 can be executed in parallel using a simple fork-join model. The computations of $R_{j,k}$ for different k in line 4 are independent of each other so they can be performed in parallel. After that, the updates to A_k can be performed in parallel too. These updates are first decomposed into independent block-by-block multiplications and computed simultaneously. Lastly, The block-column QR (line 2) can utilize a multithreaded BLAS/LAPACK kernel. Algorithm 23 shows the fork-join parallel version of Algorithm 18.

Algorithm 23: Fork-join blocked MGS BLR-QR factorization

Input: $\tilde{M}$ with $p \times q$ blocks
Output: $\tilde{Q}$ with $p \times q$ blocks and $\tilde{R}$ with $q \times q$ blocks such that $\tilde{M} \approx \tilde{Q}\tilde{R}$

```
1 for j = 0 to q − 1 do
2 |   [Q̃_j, R̃_{j,j}] = QR(M̃_j) // multithreaded
3 |   for k = j + 1 to q − 1 do in parallel
4 |   |   R̃_{j,k} = Q̃_j^T M̃_k
5 |   end
6 |   forall M̃_{i,k} where k > j and 0 ≤ i < p do in parallel
7 |   |   M̃_{i,k} ← M̃_{i,k} − Q̃_{i,k} R̃_{j,k}
8 |   end
9 end
```

4.3.2 Parallel Blocked Householder BLR-QR

The fork-join approach can also be used to parallelize Algorithm 19. Let us look at the dependency among the operations. Operations in line 4 for different j update different block-columns and only have a common dependency to the computation of $\hat{Q}_k$ in line 2. Thus they can be computed simultaneously as soon as the computation of $\hat{Q}_k$ is done. The computation of $\hat{Q}_k$ itself can be done by utilizing a multithreaded BLAS/LAPACK kernel. An example of the task dependencies is shown in Figure 4.5.

Algorithm 24 shows the parallel version where the j-loop (line 3) branches off to become a parallel region. Similarly, the left multiplication by $\tilde{Q}$ in Algorithm 20 can also be executed in parallel using the same approach.

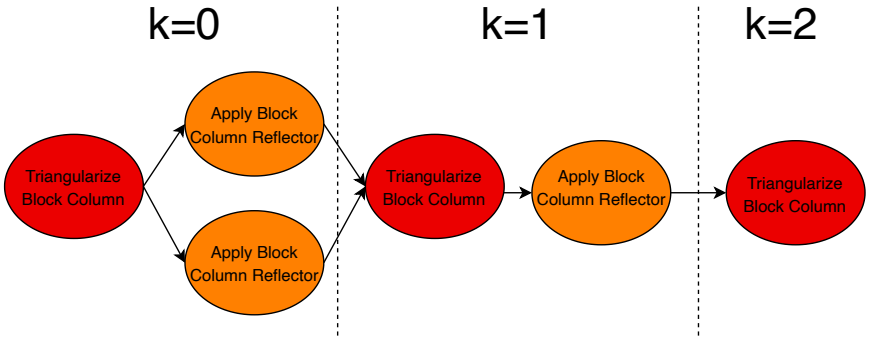


Figure 4.5: Dependency graph of Algorithm 19 on a BLR matrix with $p = q = 3$

Algorithm 24: Fork-join blocked Householder BLR-QR factorization

Input: $\tilde{M}$ with $p \times q$ blocks

Output: $\tilde{Y}, \tilde{R}$ with $p \times q$ blocks and T with $1 \times q$ blocks such that R is upper triangular and $\tilde{Y}, T$ contain intermediate orthogonal factors

for $k = 0$ **to** $q - 1$ **do**

  QR $\left(\begin{bmatrix} \tilde{M}_{k,k} \\ \tilde{M}_{k+1,k} \\ \vdots \\ \tilde{M}_{p-1,k} \end{bmatrix} \right) = \hat{Q}_k \begin{bmatrix} \tilde{R}_{k,k} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}$, such that $\hat{Q}_k = I - \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix} T_k \begin{bmatrix} \tilde{Y}_{k,k} \\ \tilde{Y}_{k+1,k} \\ \vdots \\ \tilde{Y}_{p-1,k} \end{bmatrix}^T$

  `// multithreaded`

  for $j = k + 1$ **to** $q - 1$ **do in parallel**

    Update $\begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix} \leftarrow \hat{Q}_k^T \begin{bmatrix} \tilde{M}_{k,j} \\ \tilde{M}_{k+1,j} \\ \vdots \\ \tilde{M}_{p-1,j} \end{bmatrix}$

  end

end

4.3.3 Parallel Tiled Householder BLR-QR

Looking at the operations in Algorithm 21, the fork-join approach is also applicable to obtain a parallel algorithm. The computations of $R_{k,j}$ in line 4 for different j only have a common dependency to the computation of $\hat{Q}_{k,k}$ in line 2, and thus can be performed in parallel. A similar dependency can be seen among the update operations in line 9. Algorithm 25 shows the fork-join parallel version of Algorithm 21 where the two j-loops (line 3 and 8) branch off to become parallel regions. Lastly, the computation of $\hat{Q}_{k,k}$ (line 2) and $\hat{Q}_{i,k}$ (line 7) can be performed using a multithreaded BLAS/LAPACK kernel.

Algorithm 25: Fork-join tiled Householder BLR-QR factorization

Input: $\tilde{M}$ with $p \times q$ blocks
Output: $\tilde{Y}, T, \tilde{R}$ with $p \times q$ blocks such that $\tilde{R}$ is upper triangular and $\tilde{Y}, T$ contain intermediate orthogonal factors

1 **for** $k = 0$ **to** $q - 1$ **do**
2 QR $(\tilde{M}_{k,k}) = \hat{Q}_{k,k}\tilde{R}_{k,k}$, such that $\hat{Q}_{k,k} = I - \tilde{Y}_{k,k}T_{k,k}\tilde{Y}_{k,k}^T$ `// multithreaded`
3 **for** $j = k + 1$ **to** $q - 1$ **do in parallel**
4 $\tilde{R}_{k,j} = \hat{Q}_{k,k}^T \tilde{M}_{k,j}$
5 **end**
6 **for** $i = k + 1$ **to** $p - 1$ **do**
7 QR $\left(\begin{bmatrix} \tilde{R}_{k,k} \\ \tilde{M}_{i,k} \end{bmatrix}\right) = \hat{Q}_{i,k} \begin{bmatrix} \tilde{R}'_{k,k} \\ \mathbf{0} \end{bmatrix}$, such that $\hat{Q}_{i,k} = I - \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix} T_{i,k} \begin{bmatrix} I \\ \tilde{Y}_{i,k} \end{bmatrix}^T$ `// multithreaded`
8 **for** $j = k + 1$ **to** $q - 1$ **do in parallel**
9 $\begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix} \leftarrow \hat{Q}_{i,k}^T \begin{bmatrix} \tilde{R}_{k,j} \\ \tilde{M}_{i,j} \end{bmatrix}$
10 **end**
11 **end**
12 **end**

As we have mentioned before, the tiled Householder QR has finer granularity compared to the other blocked algorithms, which could be leveraged in a parallel environment. The idea of exploiting finer granularity of tiled Householder QR to obtain a highly parallel algorithm has been introduced in [22] in the context of optimizing dense factorization. Since we are using the same tiled algorithm but adapted to the BLR format, we follow similar steps to reach an efficient parallelization scheme of our BLR-QR algorithm.

Let us again look at the operations in Algorithm 21, but this time without limiting ourselves to one k iteration. It turns out that there are direct dependencies between operations across consecutive iterations of k. This chain of dependencies is best described by a Directed Acyclic Graph (DAG), where a node represents an operation and an edge represents a dependency between two operations. Figure 4.6 shows the dependency graph when Algorithm 21 is executed on a BLR matrix with $p = q = 3$. It can be seen from Figure 4.6 that the DAG also has a recursive structure. For any

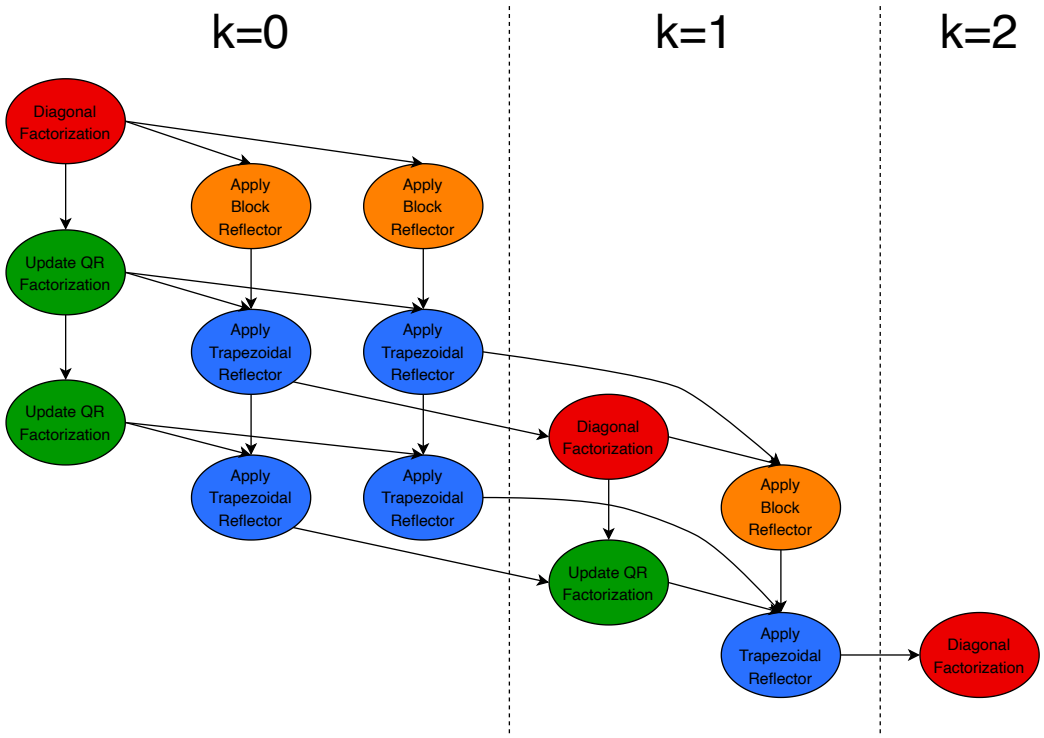


Figure 4.6: Dependency graph of Algorithm 21 on a BLR matrix with $p = q = 3$

$p_1 \geq p_2$, the DAG for BLR matrix with $p_2 \times p_2$ blocks is a subgraph of the DAG for BLR matrix with $p_1 \times p_1$ blocks. This property allows for reusing the existing DAG to accelerate the construction of a larger graph.

Once we obtain the DAG, we can use it as a guide in executing the tasks. A task can be started as soon as all of its dependencies are fulfilled. Once a task T is finished, the scheduler fulfills the dependency of tasks that are dependent on T. As a result, the threads only need to check the task pool and execute tasks that are "ready" to be executed. All threads repeat this cycle until the task pool is empty and the algorithm is finished. This results in an out-of-order execution with very loose synchronization required between the threads compared to the fork-join model.

A closer look into the graph in Figure 4.6 reveals that certain kinds of task have more outgoing edges than the others. This offers a chance for improvement because executing a task with a large number of outgoing edges will fulfill more dependencies, thus bringing more tasks into the "ready" state. Therefore, a priority value can be assigned to each task, such that a task with more outgoing edges has a higher priority to be executed first. In our algorithm, it is clear that the factorization of diagonal blocks has the highest priority. The second one is updating QR factorization, followed by the application of trapezoidal and block reflectors.

Note that it is also possible to use task-based execution for the blocked MGS and Householder-based methods. This has been discussed in [60], where the algorithm is expressed as a DAG such that

nodes represent tasks (either block-column factorization or update) and edges represent dependencies among them. The authors in [60] refer it as *dynamic lookahead* technique. However, results show that their technique is still exposed to scalability problems due to the relatively coarse granularity of the tasks. Therefore, we only discuss the task-based execution on the fine-grained tiled Householder-based algorithm.

4.4 QR Algorithm for BLR Matrices

The bottleneck of the QR eigensolver in Algorithm 14 lies in the tridiagonal reduction in line 1, which is done to reduce the cost of subsequent QR decomposition within the iteration from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. However, since our blocked BLR-QR already requires $\mathcal{O}(n^2)$ flops, we should be able to avoid the tridiagonalization step and proceed directly to the QR iteration using BLR-QR decomposition. Since the BLR matrix multiplication of $\tilde{R}\tilde{Q}$ also costs $\mathcal{O}(n^2)$, we should be able to get $\mathcal{O}(n^2)$ cost per iteration. With $\mathcal{O}(n)$ iterations required to compute all eigenvalues, the overall cost is $\mathcal{O}(n^3)$ flops. Even though this is similar to that of the dense counterpart, the storage cost of the BLR algorithm, which is $\mathcal{O}(n^{1.5})$, is still smaller than the $\mathcal{O}(n^2)$ storage cost of the dense algorithm. In addition, as the iterate gets closer to becoming a diagonal matrix, the BLR-QR will benefit from the zeroed off-diagonal blocks since their numerical ranks are close to 0, thus making the computation more efficient.

Aside from the QR decomposition, one other thing that we have to consider when adapting the QR algorithm to the BLR matrix is the deflation step, which enables us to concentrate the computation on the eigenvalues that are not already converged. For a full matrix $\tilde{A} \in \mathbb{R}^{n \times n}$, we can deflate the problem if there exists k $(1 \leq k \leq n)$ such that

$$\|\tilde{a}_{k:n,1:k-1}\| < \epsilon. \tag{4.18}$$

Since this costs $\mathcal{O}(n^3)$ to compute even for a BLR matrix $\tilde{A}$, we only check Equation (4.18) for $k = n$ to deflate the last row as soon as possible. The others are simplified by checking if all the blocks on the lower-left hand side have the numerical rank of 0, which is equivalent to

$$\|\tilde{A}_{j:p-1,0:j-1}\| < \epsilon$$

for $j = 1, 2, \ldots, p-1$ where p is the number of blocks within a row/column of $\tilde{A}$. This results in the BLR-QR eigensolver as shown in Algorithm 26. A similar approach has been done before by [65] in the context of extending the LR-Cholesky eigensolver to $\mathcal{H}$ matrices.

Algorithm 26: BLR-QR algorithm eigensolver

Input: BLR matrix $\tilde{A} \in \mathbb{R}^{n \times n}$ with $p \times p$ blocks
Output: $\Lambda \approx \Lambda(\tilde{A})$
Function $[\Lambda] = \textit{BLR-QR-algorithm}(\tilde{A})$
  while *true* **do**
    Compute shift μ
    $[\tilde{Q}, \tilde{R}] = \text{BLR-QR-decomposition}(\tilde{A} - \mu I)$
    $\tilde{A} \leftarrow \tilde{R}\tilde{Q} + \mu I$
    if $\|\tilde{a}_{n,1:n-1}\| < \epsilon$ **then**
      return $\tilde{a}_{n,n} \cup \text{BLR-QR-algorithm}(\tilde{a}_{1:n-1,1:n-1})$
    end
    if $\exists k : \|\tilde{A}_{k:p-1,0:k-1}\| < \epsilon$ **then**
      return $\text{BLR-QR-algorithm}(\tilde{A}_{0:k-1,0:k-1}) \cup$
        $\text{BLR-QR-algorithm}(\tilde{A}_{k:p-1,k:p-1})$
    end
  end
end

4.5 Numerical Result

In this section, we demonstrate the performance and accuracy of our proposed BLR algorithms using several example matrices on a shared-memory system. All algorithms were implemented in C++ where floating point calculations were performed in double precision. Fork-join and task-based parallelization were performed using OpenMP. BLAS and LAPACK routines from Intel MKL were used for the inner kernels involving dense matrices, where single-threaded kernels were used inside OpenMP parallel regions and multi-threaded kernels were used outside of parallel regions. We assumed relative error threshold ϵ for low-rank approximation of all admissible blocks, where we used our own modified version of LAPACK `DGEQP3` routine to obtain a truncated rank revealing QR factorization based on relative error threshold. Our implementation is publicly available in GitHub[1].

4.5.1 Performance and Accuracy of BLR-QR Decomposition

Here we compared our proposed BLR-QR decompositions with some existing methods. The following algorithms have been compared:

- **Dense Householder**: Householder QR factorization subroutine of Intel MKL (`DGEQRF`).
- **Blocked MGS**: Blocked modified Gram-Schmidt-based QR decomposition of weakly admissible BLR-matrices explained in Section 4.2.1.
- **Blocked Householder**: Blocked Householder-based QR decomposition of BLR-matrices explained in Section 4.2.2.

[1] `https://github.com/rioyokotalab/FRANK/tree/master`

- **Tiled Householder**: Tiled Householder-based QR decomposition of BLR-matrices explained in Section 4.2.3.

We evaluated the accuracy of BLR-QR decomposition using two metrics: one is *residual* which measures the quality of the approximate factorization; the other one is *orthogonality* which measures the quality of the orthogonal factor $\tilde{Q}$. Both metrics are respectively given by

$$\text{Res} = \frac{\|\tilde{Q}\tilde{R} - A\|_F}{\|A\|_F}, \quad \text{Orth} = \frac{\|\tilde{Q}^T\tilde{Q} - I\|_F}{\sqrt{n}},$$

where m, n denote the matrix dimensions and $\sqrt{n}$ is the Frobenius norm of order n identity matrix. Experiments were conducted on a system described in Table 4.3.

Table 4.3: Details of system used for BLR-QR decomposition experiments

	Dual AMD EPYC™ 7502
Clock speed	2.5 GHz
# cores	2 x 32 = 64
Peak performance	2560 GFlop/s
Memory	500 GB
Compiler suite	GCC 8.4
BLAS & LAPACK library	Intel MKL 2020.1.217
Multithreading	OpenMP 4.5
DGEMM performance	1861 GFlop/s

4.5.1.1 Performance on Random BLR Matrices

First, we tested our methods using randomly generated BLR matrices such that each diagonal block was a random dense matrix and each off-diagonal block was a rank-k matrix obtained from the outer product of two $b \times k$ random matrices, where b is the chosen BLR block size. We assumed weakly admissible BLR compression with error tolerance $\epsilon = 10^{-10}$.

We first show the operation and memory complexities of our algorithms using BLR matrices of varying sizes. We generated $m \times n$ ($m = 2n$) random BLR matrices with block size $b = 2\sqrt{n}$ and rank $k = 1$ off-diagonal blocks. Table 4.4 shows that our BLR methods produced approximate factorization accurately to the level of the prescribed error tolerance. Figure 4.7 shows the flops count, BLR construction times, and factorization times using a single core of the machine, and Figure 4.8 shows the corresponding memory consumption.

Figure 4.7 clearly shows that as the matrix size became larger, the flops count of both BLR-QR algorithms grew in accordance with our estimate in Section 4.2. The right part of the figure shows that our BLR methods outperformed Dense QR on large matrices. The time to construct the BLR format grew as $\mathcal{O}(mn)$ according to our description in Section 3.4.1 and they were consistently smaller than the factorization times. On the largest matrix ($n = 65,536$), the BLR methods were more than an order of magnitude faster. However, for the smallest matrix ($n = 1,024$), the BLR-QR

methods, which operate on a set of small matrix blocks, suffered from the suboptimal performance of BLAS libraries on small data sizes. The benefit of using BLR factorization started to appear when the matrix is large enough.

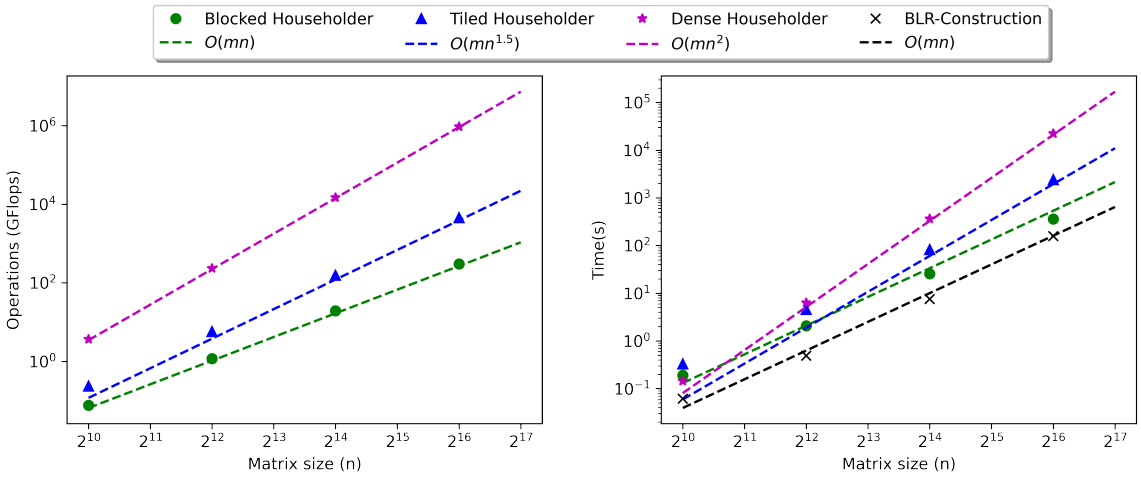


Figure 4.7: QR factorization using a single core: flops count (left); time (right)

Table 4.4: Accuracy on random BLR matrices ($\epsilon = 10^{-10}$)

m	n	Blocked Householder		Tiled Householder	
		Res	Orth	Res	Orth
2,048	1,024	$4.9 \cdot 10^{-15}$	$3.7 \cdot 10^{-15}$	$6.5 \cdot 10^{-14}$	$4.1 \cdot 10^{-13}$
8,192	4,096	$1.9 \cdot 10^{-14}$	$8.0 \cdot 10^{-15}$	$1.6 \cdot 10^{-13}$	$1.7 \cdot 10^{-12}$
32,768	16,384	$2.8 \cdot 10^{-14}$	$1.7 \cdot 10^{-14}$	$9.8 \cdot 10^{-14}$	$1.1 \cdot 10^{-12}$
131,072	65,536	$2.2 \cdot 10^{-13}$	$3.7 \cdot 10^{-14}$	$3.9 \cdot 10^{-13}$	$5.3 \cdot 10^{-14}$

Figure 4.8 shows that compression using BLR matrix, followed by performing QR factorization on the compressed form led to orders of magnitude smaller memory consumption compared to performing direct factorization on the dense matrix. The memory consumption of both blocked Householder and MGS-based methods grew as $\mathcal{O}(m\sqrt{n})$, which corresponds to the storage requirement of a rectangular BLR matrix. Our blocked Householder-based method consumed slightly less memory compared to the existing blocked MGS-based method because it reuses the lower triangular part of $\tilde{R}$ (for Y matrices) plus a block diagonal matrix (for T matrices) to implicitly store the orthogonal factor $\tilde{Q}$, whereas the MGS-based method explicitly forms two BLR-matrices $\tilde{Q}$ and $\tilde{R}$ during the factorization. However, the tiled Householder-based method consumed more memory than the other BLR methods since it needs more additional space to store the T matrices coming from the update QR factorization steps. This led to a memory consumption that grows similarly to the dense QR. As a remedy, an inner blocking technique [22, 73] could be employed to reduce additional storage for the T matrices.

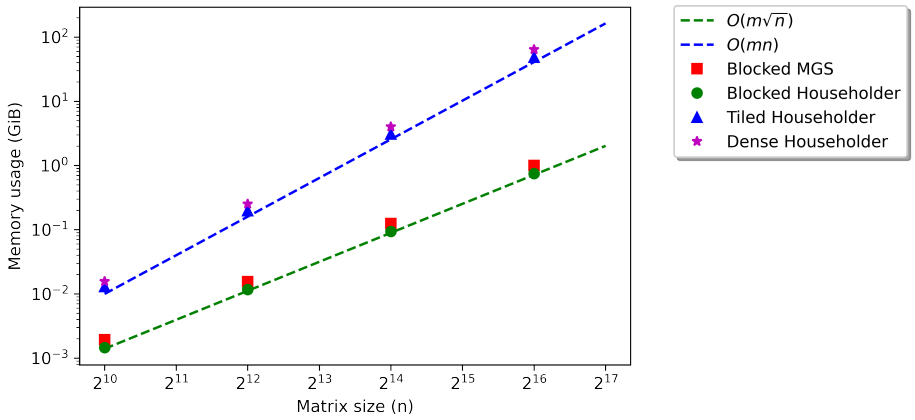


Figure 4.8: Memory consumption during executions using a single-core

We now demonstrate the parallel scalability of our methods. We used two random $m \times n$ ($m = 2n$) BLR-matrices with $n = 16,384$ and $n = 32,768$, block size $b = 256$, and rank $k = 16$ off-diagonal blocks. We compared our parallel algorithms with the existing parallel blocked MGS-based algorithm. Figures 4.9, 4.10, and 4.11 show the factorization time, speedup, and flops rate using different numbers of threads, respectively.

Figure 4.9 shows that all BLR methods scaled nicely using up to 64 cores of the machine. For the matrix of size $n = 16,384$, the task-based tiled Householder method outperformed the blocked Householder when using 64 cores. However, on the larger matrix ($n = 32,768$), using 64 cores of our machine was not sufficient for this to happen. But we can expect that when the number of threads increases, the tiled method would eventually outperform the blocked method. This shows that the finer granularity of the tiled Householder method that allows for efficient dynamic task-based execution was able to overcome the induced extra operations once we have a large number of computing cores.

Figure 4.10 shows that the fork-join blocked MGS, tiled Householder, and blocked Householder-based methods showed a speedup of up to 26, 26, and 37 times, respectively. The fork-join blocked Householder-based methods showed higher speedups compared to the MGS-based. Even though both of these methods perform similar block-column-wise QR, the blocked MGS has a bottleneck of computing the $\tilde{R}_{j,k}$ (line 3-5 of Algorithm 23) [55]. Furthermore, the blocked Householder outperformed the tiled Householder when using fork-join execution model. On the other hand, the task-based tiled Householder achieved up to 50 times speedup, thanks to the DAG-based execution that fully utilized the dependency between operations in the tiled Householder QR, allowing it to scale almost perfectly as the number of threads increases.

Although the scalability is promising, the actual performance of the BLR-QR algorithms is still far from the peak performance of the machine, as shown in Figure 4.11. There are two reasons behind

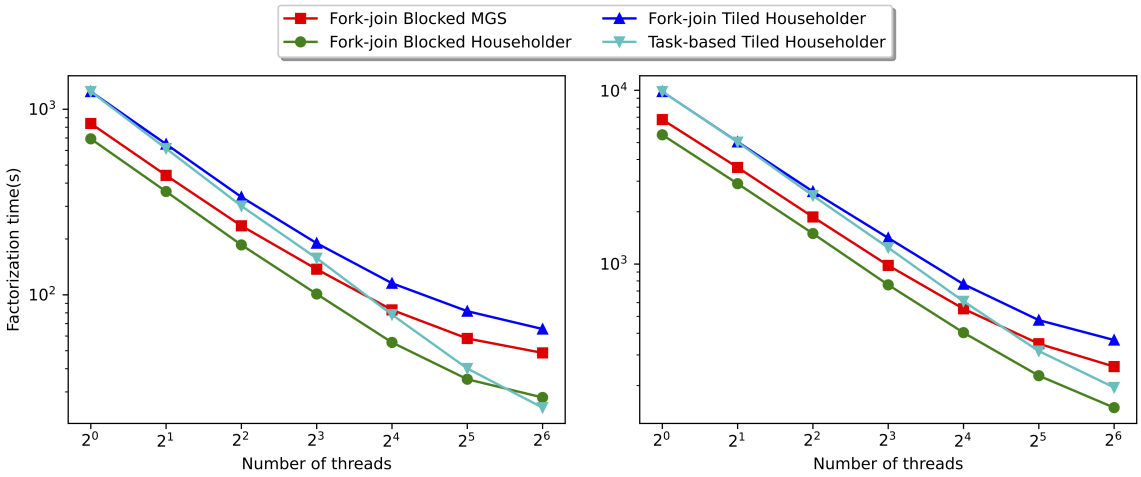


Figure 4.9: Factorization time using various number of threads: n=16,384 (left); n=32,768 (right)

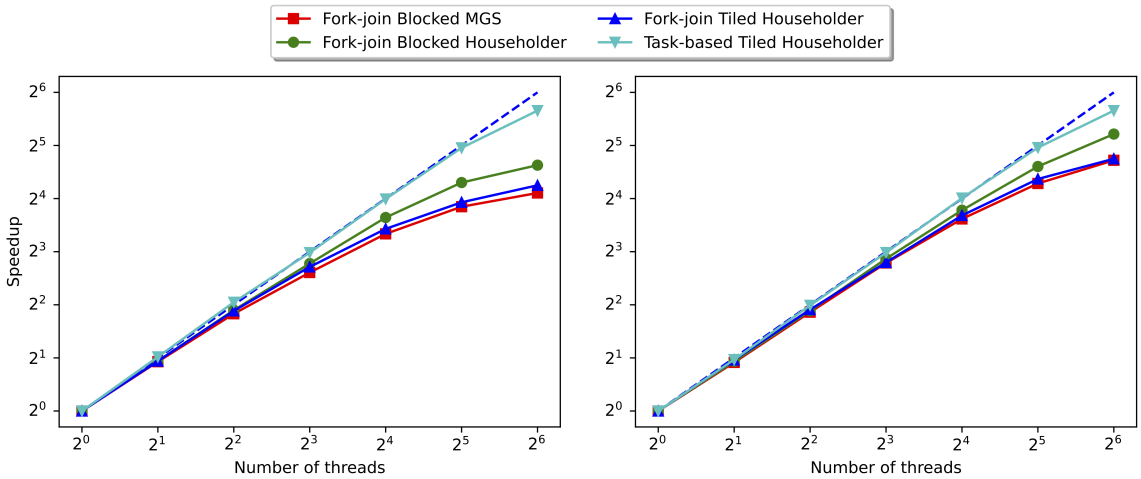


Figure 4.10: Speedup using various number of threads: n=16,384 (left); n=32,768 (right)

this. First, BLR-QR methods have to deal with low-rank block operations, which involve manipulating a collection of small matrices instead of one large matrix, making it more memory-bound [74, 91]. Second is the suboptimal performance of BLAS routines on small data sizes. Therefore we cannot expect these algorithms to reach the same flops rate as the traditional dense algorithms.

Samples of a parallel execution trace are shown in Figure 4.12 where the grey part corresponds to computation and the white part corresponds to synchronization and overhead. The fork-join model has many synchronizations involving all threads, which means threads that completed their task first need to wait for the others before proceeding with the execution. However, the task-based execution showed very loose synchronization because once a thread finishes a task, it can take another "ready" task from the task pool and begin another execution without waiting for other threads. This led to an out-of-order execution that eliminates unnecessary synchronizations and significantly reduces

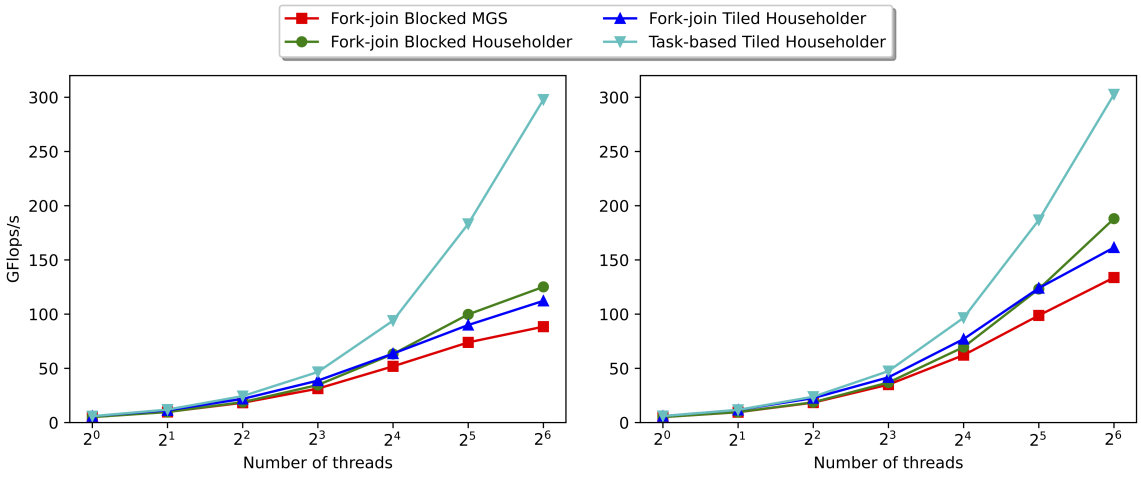


Figure 4.11: Flops rate using various number of threads: n=16,384 (left); n=32,768 (right)

the idle time of threads. Furthermore, one can also expect the tiled method to perform better in distributed memory systems due to its fine granularity which would lead to smaller communication overhead compared to the blocked method. Note that due to the limitation of OpenMP, in our implementation, the dependency graph was constructed on the fly by the master thread, which corresponds to the large overhead in the beginning. This is not very efficient when using a small number of threads since the master thread spends more than 70% of its time generating tasks. This overhead however is not significant when using a large number of threads.

4.5.1.2 Accuracy on Ill-Conditioned Matrices

In this example, we demonstrate the numerical stability of our methods in factorizing ill-conditioned matrices. We used matrices arising from the Boundary-Element-Method (BEM) discretization of a Single-Layer Potential (SLP) operator using the 2D Laplace equation on the unit circle mesh, generated using `H2Lib` [24]. The resulting square matrices were ill-conditioned and had off-diagonal blocks with small ranks, hence we assumed weakly admissible BLR compression. We set the block size $b = 2\sqrt{n}$ and error tolerance $\epsilon = 10^{-9}$.

We compared the accuracy of our methods with the existing blocked MGS-based method. Table 4.5 shows that as the condition number increases, our Householder methods were robust to this increase and produced numerical orthogonality on the level of the prescribed tolerance. On the other side, the orthogonality produced by the MGS method clearly deteriorated as the condition number increases.

4.5.1.3 Performance on Spatial Statistics Problems

In this example, we used square matrices arising from the Spatial Statistics problem with exponential kernel on uniform 3D grids, generated using `STARS-H` [1]. The resulting matrices have many off-

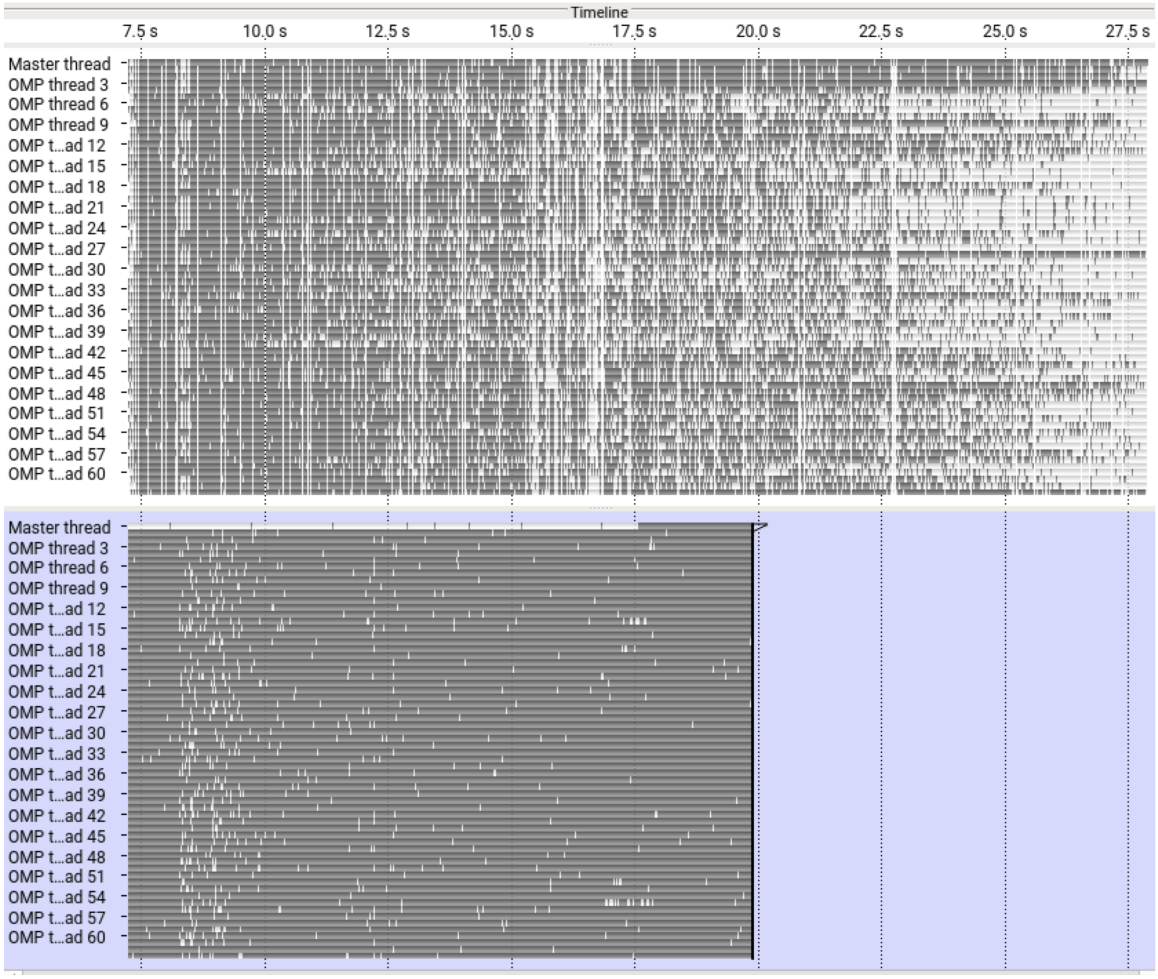


Figure 4.12: Execution traces of parallel tiled Householder BLR-QR on an order 8,192 matrix: fork-join (top, 20.8s); task-based (bottom, 12.9s). BLR-QR executions start at around 7.5 seconds of the timeline

diagonal blocks with relatively high ranks, thus a strongly admissible compression is preferred. We used a modified version of Equation (3.28) so that a larger value of η introduces more inadmissible off-diagonal block.

We first demonstrate the parallel scalability of our methods for the matrix of order $n = 16,384$ compressed with block size $b = 256$, tolerance $\epsilon = 10^{-6}$, and admissibility constant $\eta = 0.3$. Figure 4.13 shows that using 64 cores the fork-join tiled and blocked householder methods achieved a speedup of 13 and 17 times, respectively. On the other hand, the task-based tiled Householder method achieved 47 times speedup, allowing it to become faster than the blocked method.

Next, we evaluate the performance and accuracy of our methods using matrices of varying sizes and admissibility constant. We compared our fork-join blocked and task-based tiled Householder with the parallel dense QR of Intel MKL using 64 cores in terms of factorization time and memory consumption. Table 4.6 shows that our BLR-QR methods could produce residual and orthogonality

Table 4.5: Accuracy on ill-conditioned matrices ($\epsilon = 10^{-9}$)

n	$\kappa_F(A)$	Block Size	Max Rank	Blocked Householder Res	Blocked Householder Orth	Tiled Householder Res	Tiled Householder Orth	Blocked MGS Res	Blocked MGS Orth
1,024	$2.8 \cdot 10^5$	64	11	$6.8 \cdot 10^{-10}$	$6.9 \cdot 10^{-11}$	$6.1 \cdot 10^{-10}$	$5.2 \cdot 10^{-11}$	$5.1 \cdot 10^{-10}$	$1.9 \cdot 10^{-8}$
4,096	$4.6 \cdot 10^6$	128	12	$1.0 \cdot 10^{-9}$	$1.2 \cdot 10^{-10}$	$9.6 \cdot 10^{-10}$	$4.5 \cdot 10^{-11}$	$8.6 \cdot 10^{-10}$	$1.0 \cdot 10^{-7}$
16,384	$7.4 \cdot 10^7$	256	12	$2.1 \cdot 10^{-9}$	$6.2 \cdot 10^{-11}$	$1.8 \cdot 10^{-9}$	$6.7 \cdot 10^{-11}$	$1.8 \cdot 10^{-9}$	$1.5 \cdot 10^{-6}$
32,768	$2.9 \cdot 10^8$	512	13	$2.3 \cdot 10^{-9}$	$6.0 \cdot 10^{-11}$	$2.0 \cdot 10^{-9}$	$5.3 \cdot 10^{-11}$	$2.0 \cdot 10^{-9}$	$6.9 \cdot 10^{-6}$

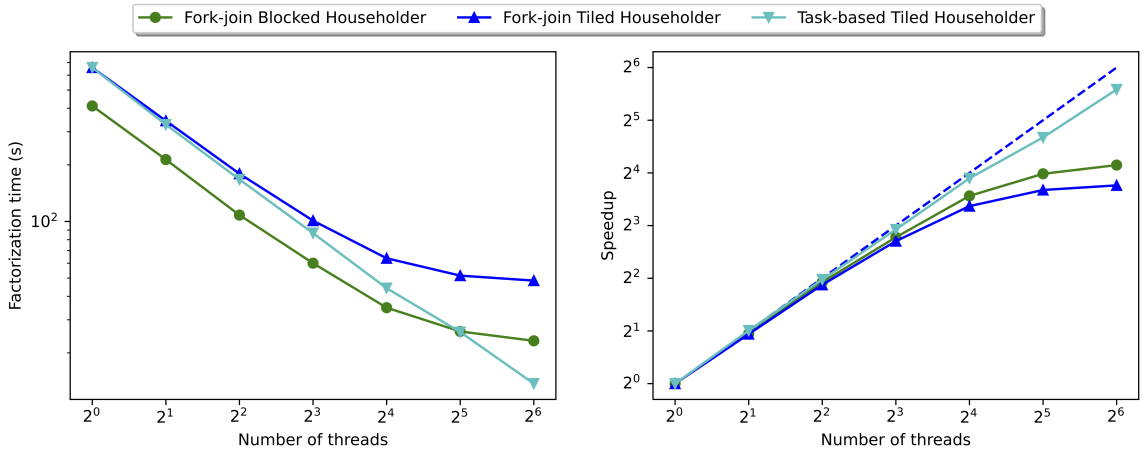


Figure 4.13: Parallel scalability on spatial statistics problems: factorization time (left); speedup (right)

to the level of the prescribed error tolerance. It also shows that we could fine-tune the admissibility constant (introduce more/less off-diagonal inadmissible blocks, i.e. decrease/increase the maximum rank of admissible off-diagonal blocks) to reach the optimal factorization time and memory consumption. On the matrix of order 16k, the BLR methods were slower than the dense QR. However, for the larger matrix of order 65k, the BLR methods were faster and achieved up to 80% less memory usage compared to the dense QR.

Table 4.6: Accuracy on 3D Spatial Statistics problems ($\epsilon = 10^{-6}$)

n	Dense QR Mem (MB)	Dense QR Time (s)	BLR Block Size	BLR η	BLR Max Rank	Blocked Householder Mem (MB)	Blocked Householder Time (s)	Blocked Householder Res	Blocked Householder Orth	Tiled Householder Mem (MB)	Tiled Householder Time (s)	Tiled Householder Res	Tiled Householder Orth
16,384	2,048	8.26	256	0.2	117	847	32.5	$1.1 \cdot 10^{-9}$	$1.3 \cdot 10^{-11}$	1,857	16.9	$1.5 \cdot 10^{-9}$	$1.1 \cdot 10^{-9}$
			256	0.3	86	954	23	$8.0 \cdot 10^{-10}$	$1.1 \cdot 10^{-11}$	1,964	13.7	$1.4 \cdot 10^{-9}$	$1.0 \cdot 10^{-9}$
			256	0.4	58	1,284	16.4	$2.1 \cdot 10^{-10}$	$3.8 \cdot 10^{-12}$	2,293	10.8	$3.8 \cdot 10^{-10}$	$2.2 \cdot 10^{-10}$
65,536	32,768	310	512	0.2	175	6,698	257.7	$3.7 \cdot 10^{-9}$	$1.5 \cdot 10^{-10}$	22,957	215.9	$6.3 \cdot 10^{-9}$	$4.0 \cdot 10^{-9}$
			512	0.3	94	10,236	185.6	$1.5 \cdot 10^{-9}$	$8.8 \cdot 10^{-11}$	26,491	218.8	$2.8 \cdot 10^{-9}$	$1.8 \cdot 10^{-9}$
			512	0.4	51	14,684	230.9	$5.1 \cdot 10^{-10}$	$4.3 \cdot 10^{-11}$	30,937	286.9	$9.9 \cdot 10^{-10}$	$6.3 \cdot 10^{-10}$

4.5.1.4 Performance on Inverse Poisson Problems

Here, we tested our BLR-QR decomposition algorithms on sparse least squares matrices arising from the Inverse Poisson problem defined on uniform 2D grids, generated using the MATLAB code of

spaQR [40]. We used strongly admissible BLR compression with block size $b = 2\sqrt{n}$ and tolerance $\epsilon = 10^{-10}$. Since the geometry information was not available, we attempted to compress every off-diagonal block and reverted back to dense the blocks whose rank was larger than $b/2$.

Figure 4.14 shows the parallel scalability of our BLR methods for the matrix of size $74,112 \times 36,864$. Using 64 cores, the fork-join tiled, fork-join blocked, and task-based tiled Householder methods achieved a speedup of 2, 3, and 14 times, respectively. These relatively lower speedups came from the fact that although the dimension was quite large, the resulting BLR matrix was dominated by zero blocks that made the actual computation load smaller.

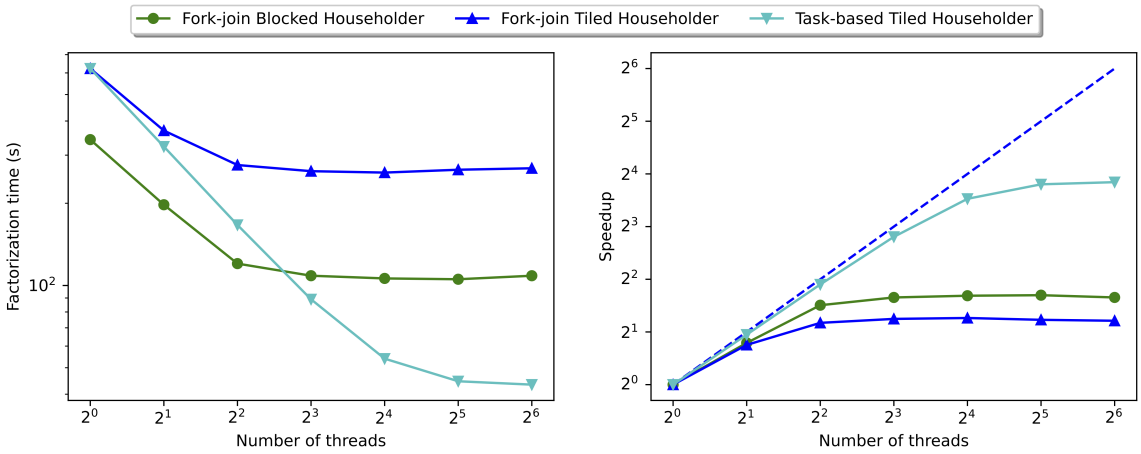


Figure 4.14: Parallel scalability on inverse Poisson problems: factorization time (left); speedup (right)

We then compared our fork-join blocked and task-based tiled Householder with the parallel dense QR of Intel MKL using all 64 cores of the machine. Table 4.7 shows that the BLR methods were faster than Dense QR while producing residual and orthogonality to the level of the prescribed error tolerance. Moreover, the tiled method was faster than the blocked method in all three sparse matrices that we used.

Table 4.7: Accuracy on 2D Inverse Poisson problems ($\epsilon = 10^{-10}$)

		Dense QR		Blocked Householder				Tiled Householder			
m	n	Mem (MB)	Time (s)	Mem (MB)	Time (s)	Res	Orth	Mem (MB)	Time (s)	Res	Orth
74,112	36,864	20,844	200	3,727	106	$1.4 \cdot 10^{-11}$	$1.3 \cdot 10^{-11}$	20,017	46	$8.4 \cdot 10^{-12}$	$5.7 \cdot 10^{-12}$
100,800	50,176	38,587	479	6,117	190	$1.5 \cdot 10^{-11}$	$1.4 \cdot 10^{-11}$	36,125	82	$8.4 \cdot 10^{-12}$	$5.4 \cdot 10^{-12}$
131,584	65,536	65,792	1569	9,399	354	$1.6 \cdot 10^{-11}$	$1.6 \cdot 10^{-11}$	60,351	161	$8.6 \cdot 10^{-12}$	$5.4 \cdot 10^{-12}$

4.5.2 Performance and Accuracy of BLR-QR Eigenvalue Solver

In this section, we evaluate the performance and accuracy of our BLR-QR decomposition when applied to the QR eigenvalue solver by comparing the following algorithms:

- **Dense-QR Algorithm**: Algorithm 14 without tridiagonalization using `DGEQRF` subroutine for the QR decomposition.
- **BLR-QR Algorithm**: BLR-based QR algorithm without tridiagonalization presented in Algorithm 26 using the fork-join parallel blocked Householder BLR-QR decomposition described in Algorithm 24.
- **LAPACK dsyev**: Dense symmetric eigenvalue solver of Intel MKL, which is an optimized implementation of implicit QR algorithm with tridiagonalization.

We evaluated the accuracy of our eigenvalue solver using the absolute error of the produced eigenvalues, such that

$$e_{abs} = \max_{1 \leq j \leq n} \left| \lambda_j - \tilde{\lambda}_j \right|$$

where $\tilde{\lambda}_j$ is the computed eigenvalue and λ_j is the machine precision accuracy eigenvalue computed by LAPACK `dsyev`. For the test matrices, we used the 1D Laplace kernel to generate $n \times n$ structured low-rank matrix A such that

$$a_{i,j} = \frac{1}{|x_i - x_j| + 10^{-3}},$$

where $|x_i - x_j|$ denotes the Euclidean distance between two points x_i and x_j. The points x_i were uniformly distributed on the interval $[1, n]$. Experiments were conducted on a system described in Table 4.8.

Table 4.8: Details of system used for eigenvalue experiments

	Dual AMD Ryzen Threadripper 3960X
Clock speed	3.8 GHz
# cores	2 x 24 = 48
Memory	120 GB
Compiler suite	GCC 9.4
BLAS & LAPACK library	Intel 2022.1.0
Multithreading	OpenMP 4.5

We first tested the accuracy of the BLR-QR eigensolver using various relative error thresholds ϵ for the low-rank approximations. Table 4.9 shows that the eigenvalue errors were proportional to the chosen threshold times the number of iterations. The number of iterations also grew linearly with the matrix dimension n. Furthermore, the computation times seemed to be growing as $\mathcal{O}(n^3)$, which is in accordance with our estimation.

We then compared the performance of our BLR-QR eigenvalue solver against the existing methods in Figure 4.15. The left part of the figure shows that our method was slower than LAPACK `dsyev`. Moreover, our method was also slower than its dense counterpart which is expected to have $\mathcal{O}(n^4)$ complexity. This is a disappointing result, which suggests that the cost per iteration in our

Table 4.9: Numerical results of the BLR-QR algorithm eigenvalue solver

n	Block Size	$\epsilon = 10^{-6}$ time (s)	#iter	e_{abs}	$\epsilon = 10^{-8}$ time (s)	#iter	e_{abs}	$\epsilon = 10^{-10}$ time (s)	#iter	e_{abs}
128	16	1.5	175	$8.9 \cdot 10^{-7}$	1.7	202	$6.3 \cdot 10^{-9}$	1.9	226	$9.1 \cdot 10^{-11}$
512	32	57	562	$2.3 \cdot 10^{-6}$	65	617	$1.0 \cdot 10^{-8}$	74	695	$9.1 \cdot 10^{-11}$
2048	64	3971	2093	$2.4 \cdot 10^{-6}$	4258	2127	$1.8 \cdot 10^{-8}$	4766	2230	$1.6 \cdot 10^{-10}$

method was $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n^2)$. This was verified by the flops count measurement shown in the right part of Figure 4.15, where it is clear that our method requires $\mathcal{O}(n^4)$ flops, similar to that of its dense counterpart. This is mainly attributed to the rank growth of the off-diagonal admissible blocks caused by the arithmetic operations within the QR iteration. Particularly, the low-rank approximations within the BLR matrix multiplication $\tilde{R}\tilde{Q}$ in line 5 of Algorithm 26 made the local block-wise ranks increase, as shown in the example in Figure 4.16 where after 25 iterations some off-diagonal blocks start to become full-rank matrices. This rank increase would affect the subsequent BLR-QR decomposition, making them cost $\mathcal{O}(n^3)$ flops instead of $\mathcal{O}(n^2)$. Following the example in Figure 4.16, we observed that after a few hundred iterations the convergence to the diagonal matrix started to reduce the ranks of the off-diagonal blocks. But this effect was too late since the iterations in between were too expensive. Similar behavior was also observed by [65] in the context of applying QR decomposition of $\mathcal{H}$-matrix to the QR algorithm eigenvalue solver.

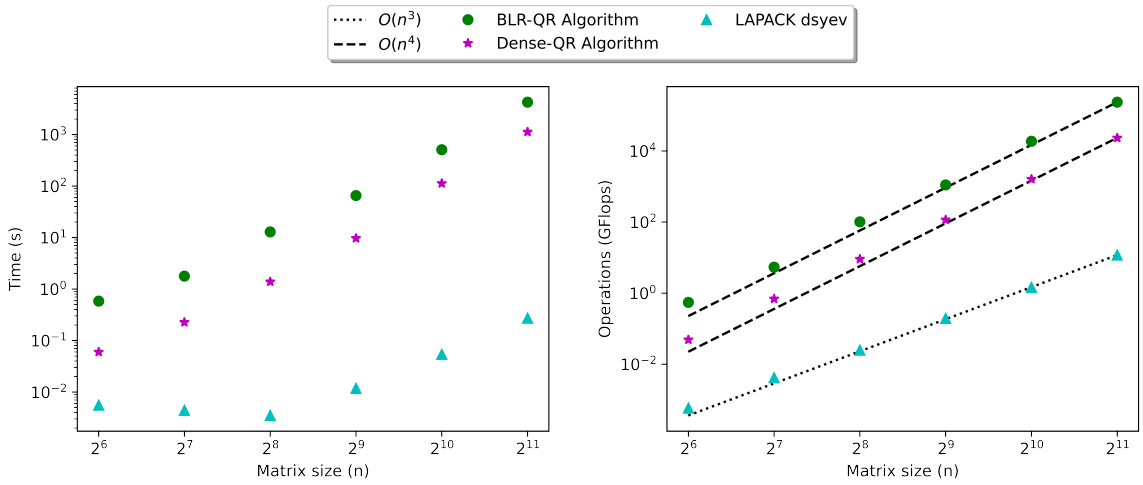
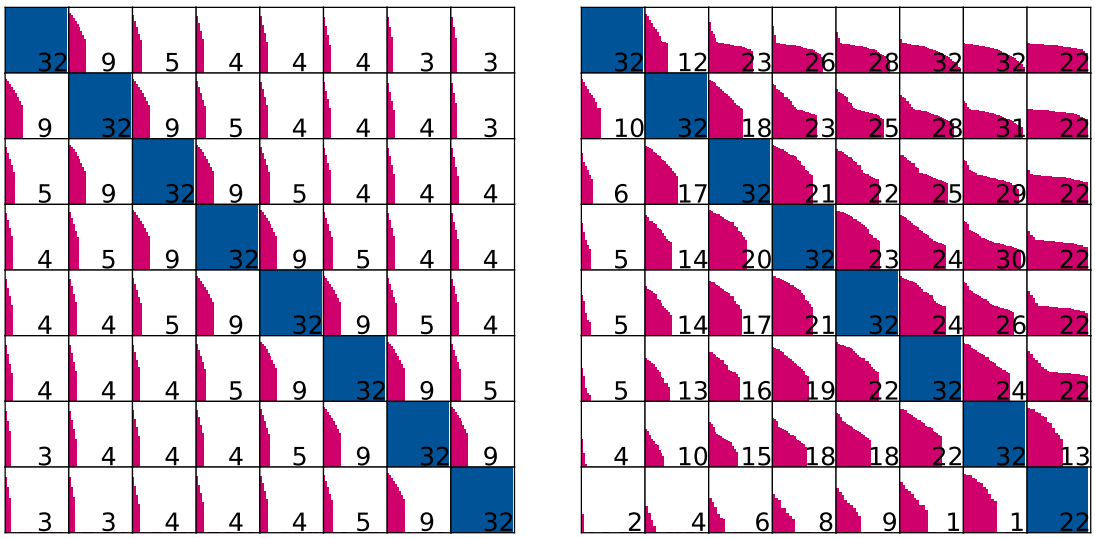


Figure 4.15: Computing all eigenvalues using 48 cores: time (left); flops count (right)

It has to be remarked that the seemingly $\mathcal{O}(n^3)$ computation time of our method in Table 4.9 can be caused by the small matrix sizes that were yet to reflect the actual scaling as the problem size grows. If larger matrices were used, we expect the computation time to grow as $\mathcal{O}(n^4)$, which is even more expensive than the QR algorithm with tridiagonalization of LAPACK `dsyev`. Therefore, here we conclude that our BLR-QR decomposition could not be used to obtain an efficient QR algorithm



(a) (b)

Figure 4.16: Structure of an example of BLR matrix: initial (left); after 25 steps of QR iteration (right)

eigenvalue solver of BLR matrices, at least when applied within the QR iteration. However, it could still be applied to, for example, the tridiagonal reduction of BLR matrices, as they both are based on the Householder method.

Chapter 5

Hierarchical LDL Decomposition for Eigenvalue Problems

In this chapter we study a method that has been receiving less attention in recent literature, that is the bisection eigenvalue solver described in Section 2.4.7. This method requires the LDL decomposition of the shifted matrix $A - \mu I$ at each bisection iteration to find the desired k-th smallest eigenvalue of A. For general dense matrix A, this evaluation requires $\mathcal{O}(n^3)$ flops, making it prohibitive to compute for large matrices. Structured low-rank matrices on the other hand allow for fast LDL decomposition in down to linear time so that it can significantly reduce the cost of computing the eigenvalue.

Here we present two linear time generalized LDL decomposition algorithms based on the HSS and $\mathcal{H}^2$-matrices formats. The first one is based on the generalized LDL decomposition of HSS matrices in [88] with an extension to distributed memory systems. Another one is based on the $\mathcal{H}^2$-ULV decomposition described in [63]. We then apply both methods to compute the k-th smallest eigenvalue of structured low-rank matrices efficiently using the slicing-the-spectrum method.

The rest of this chapter is organized as follows. In Section 5.1 we describe the generalized Cholesky decomposition, which serves as the basis of our proposed algorithms. We then describe our proposed generalized LDL decomposition of HSS and $\mathcal{H}^2$ matrices along with their application to compute the k-th smallest eigenvalue in Sections 5.2 and 5.3, respectively. The content of Section 5.3 is scheduled to be published in [10].

5.1 Generalized Cholesky Decomposition of HSS Matrices

The generalized Cholesky decomposition was first introduced in [89] for symmetric positive definite HSS-matrices. Instead of directly performing Cholesky decomposition on the HSS matrix, the method first utilizes the shared bases to introduce zeros to the off-diagonal low-rank blocks. Then, a partial factorization is performed on the sparsified HSS matrix such that the computations of

Schur's complements only occur within the diagonal blocks. After that, the same process is performed recursively to the remaining unfactorized parts, from the leaf level going up to the root level, which is possible thanks to the nested property of the shared bases. This method is often called the ULV decomposition and it has an overall cost of $\mathcal{O}(n)$ flops. Moreover, under the weak admissibility condition, the ULV factorization does not involve further approximation of the HSS matrix, resulting in an exact factorization of the given HSS matrix. This is particularly useful in certain cases, notably for the bisection eigenvalue solver that will be discussed in later sections.

The main idea of ULV decomposition is the introduction of zeros to the off-diagonal low-rank blocks, which we will describe as follows. Let $U^S_{l;i} \in \mathbb{R}^{k\times r}$ $(r \leq k)$ be the shared column basis (or transfer matrix) for the i-th row at level l, containing r orthonormal columns as defined in Section 3.4.4. Thus, there exists a matrix $U^R_{l;i} \in \mathbb{R}^{k\times(k-r)}$ with orthonormal columns that form the basis of a subspace orthogonal to the range of $U^S_{l;i}$ such that

$$\begin{aligned}\left(U^R_{l;i}\right)^T U^S_{l;i} &= \mathbf{0}\\ \left(U^S_{l;i}\right)^T U^R_{l;i} &= \mathbf{0}.\end{aligned}$$

Since k and r are small numbers that do not grow with the matrix dimension n, the matrix $U^R_{l;i}$ can be obtained quickly by performing the full QR decomposition

$$\left[\begin{bmatrix}U^S_{l;i} & U^R_{l;i}\end{bmatrix}, R\right] = QR\left(U^S_{l;i}\right). \tag{5.1}$$

Thus, the matrices $U^R_{l;i}$ for every row i at all levels $l = 1, 2, \ldots, L$ can be obtained in $\mathcal{O}(n)$ flops. The S and R superscripts denote the *skeleton* and *redundant* part of the basis, respectively. We use the term "redundant" because it corresponds to the part of the basis that is usually discarded during low-rank approximation. For example during the construction of the leaf level shared bases

$$\begin{bmatrix}U^S_{L;i} & U^R_{L;i}\end{bmatrix} V^T = A_{L;i,+}, \tag{5.2}$$

where the skeleton part is taken as the approximate shared column basis for the entire row i. A similar process can be observed on the non-leaf level too. For example during the construction of transfer matrices on the direct parent of the leaf level:

$$\begin{bmatrix}U^S_{L-1;i} & U^R_{L-1;i}\end{bmatrix} V^T = \begin{bmatrix}U^S_{L;2i} & 0\\ 0 & U^S_{L;2i+1}\end{bmatrix}^T A_{L-1;i,+}. \tag{5.3}$$

Therefore, these redundant parts can also be obtained immediately during the shared/nested bases construction stage, which does not cost additional flops. The redundant part of the bases will then be used to introduce zeros to the off-diagonal low-rank blocks at each level by performing the left and right multiplications

$$\left(U^{(l)}\right)^T \tilde{A}^{(l)} U^{(l)}, \tag{5.4}$$

where

$$U^{(l)} = \begin{bmatrix} \begin{bmatrix} U^R_{l;0} & U^S_{l;0} \end{bmatrix} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \begin{bmatrix} U^R_{l;1} & U^S_{l;1} \end{bmatrix} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \begin{bmatrix} U^R_{l;2^l-1} & U^S_{l;2^l-1} \end{bmatrix} \end{bmatrix}$$

and $\tilde{A}^{(l)}$ is the remaining blocks to be factorized at level l. A more detailed discussion can be found in [89, 88].

5.2 Computing the k-th Eigenvalue of Symmetric HSS Matrices

Here we describe our proposed method for the generalized LDL decomposition of HSS matrices and its application to computing the k-th eigenvalue. We start by introducing the generalized LDL decomposition of the weakly admissible BLR^2 matrices in Section 5.2.1, which can be seen as a simple, non-hierarchical version of HSS matrices. Then, we introduce hierarchy to obtain generalized HSS-LDL decomposition in Section 5.2.2. Finally, we employ it to compute the k-th eigenvalue of HSS matrices using slicing-the-spectrum in Section 5.2.3 and discuss its parallelization in Section 5.2.4. Numerical results that demonstrate the performance and accuracy of our method on both shared and distributed memory systems are presented in Section 5.2.5.

5.2.1 Generalized LDL Decomposition of BLR^2 Matrices

Let $\tilde{A}$ be a BLR^2 matrix under the weak admissibility condition. The generalized LDL decomposition proceeds as follows. The overall flow for a 4×4 blocks example is shown in Figure 5.1. First,

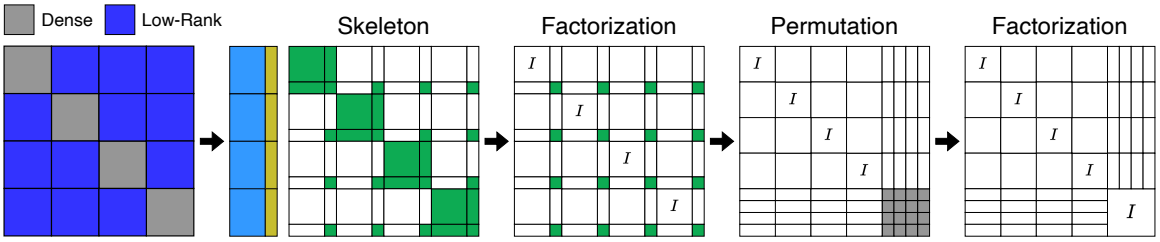


Figure 5.1: Flow of generalized BLR^2-LDL factorization

the skeleton matrices for both dense diagonal and off-diagonal low-rank blocks are formed by incorporating the redundant part of the shared bases. For each dense diagonal block $A_{1;i,i}$, the skeleton matrix is written as

$$\begin{bmatrix} S^{RR}_{1;i,i} & S^{RS}_{1;i,i} \\ S^{SR}_{1;i,i} & S^{SS}_{1;i,i} \end{bmatrix} = \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}^T A_{1;i,i} \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}. \tag{5.5}$$

For each off-diagonal low-rank block $\tilde{A}_{1;i,j}$ $(i \neq j)$, the skeleton matrix is obtained from

$$\begin{aligned}\begin{bmatrix} S^{RR}_{1;i,j} & S^{RS}_{1;i,j} \\ S^{SR}_{1;i,j} & S^{SS}_{1;i,j} \end{bmatrix} &= \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}^T \tilde{A}_{1;i,j} \begin{bmatrix} U^R_{1;j} & U^S_{1;j} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & U^S_{1;i} A_{1;i,j} U^S_{1;j} \end{bmatrix}. \end{aligned} \tag{5.6}$$

This formation of skeleton matrices is equivalent to making the bases $\begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}$ and $\begin{bmatrix} U^R_{1;j} & U^S_{1;j} \end{bmatrix}^T$ to be shared among both dense and low-rank blocks in the same row and column, respectively. As a result, it introduces zeros to a large portion of each off-diagonal low-rank block, as shown in Equation (5.6). This process corresponds to the "Skeleton" phase of Figure 5.1.

The next step is to partially factorize the skeleton matrices. We first compute the LDL factorization of the redundant part of the diagonal blocks to obtain

$$L^{RR}_{1;i,i}, D^{RR}_{1;i,i} = LDL(S^{RR}_{1;i,i}). \tag{5.7}$$

After that we eliminate the $S^{SR}_{1;i,i}$ blocks by

$$L^{SR}_{1;i,i} = S^{SR}_{1;i,i} \left(L^{RR}_{1;i,i}\right)^{-T} \left(D^{RR}_{1;i,i}\right)^{-1}, \tag{5.8}$$

then compute the trailing update

$$S^{SS}_{1;i,i} = S^{SS}_{1;i,i} - L^{SR}_{1;i,i} D^{RR}_{1;i,i} \left(L^{SR}_{1;i,i}\right)^T. \tag{5.9}$$

Note that since the trailing update in Equation (5.9) is only performed on the skeleton part of each diagonal block, the processes described in Equations (5.7)-(5.9) for different i are independent of each other and thus can be done in parallel, as shown in the "Factorization" phase in Figure 5.1. Once the redundant part of the diagonal blocks ($S^{RR}_{1;i,i}$) are factorized, the remaining $S^{SS}_{1;i,j}$ blocks are permuted to the bottom-right corner, as shown in the "Permutation" phase in Figure 5.1. Lastly, the entire remaining block is factorized as a single dense matrix:

$$L^{SS}_{1;0:3,0:3}, D^{SS}_{1;0:3,0:3} = LDL\left(\begin{bmatrix} S^{SS}_{1;0,0} & S^{SS}_{1;0,1} & S^{SS}_{1;0,2} & S^{SS}_{1;0,3} \\ S^{SS}_{1;1,0} & S^{SS}_{1;1,1} & S^{SS}_{1;1,2} & S^{SS}_{1;1,3} \\ S^{SS}_{1;2,0} & S^{SS}_{1;2,1} & S^{SS}_{1;2,2} & S^{SS}_{1;2,3} \\ S^{SS}_{1;3,0} & S^{SS}_{1;3,1} & S^{SS}_{1;3,2} & S^{SS}_{1;3,3} \end{bmatrix}\right). \tag{5.10}$$

The final step in Equation (5.10) corresponds to the rightmost "Factorization" phase of Figure 5.1. This concludes the generalized LDL factorization of weakly admissible BLR^2 matrix.

5.2.2 Generalized LDL Decomposition of HSS Matrices

Here we describe generalized LDL factorization of HSS matrices by introducing hierarchy to the process that we have just demonstrated on BLR^2 matrices. For HSS matrices, the factorization proceeds in a bottom-up fashion starting from the leaf level and going up to the root. We use an

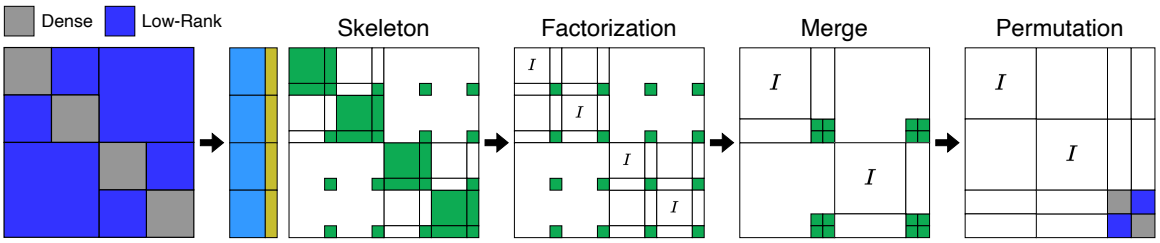


Figure 5.2: Flow of generalized HSS-LDL factorization at the leaf level

example of a 2-level HSS matrix to illustrate this process, as shown in Figure 5.2 and Figure 5.3. The same recursive steps can be applied to general HSS matrices of arbitrary size and number of levels.

The flow at each level is identical to that of the BLR^2 version. Starting from the leaf level (level=2), we perform a similar process described in Equations (5.5)-(5.9), except that now the leaf level is $l = 2$ instead of $l = 1$ in the BLR^2 case. This corresponds to the "Skeleton" and "Factorization" phases in Figure 5.2. After that, instead of factorizing the remaining blocks as a single dense matrix shown in Equation (5.10), we treat them as another HSS matrix whose level is reduced by 1 and recursively apply a similar procedure. At this upper level, the remaining blocks can be expressed as

$$\begin{bmatrix} \hat{S}_{1;0,0} & \hat{S}_{1;0,1} \\ \hat{S}_{1;1,0} & \hat{S}_{1;1,1} \end{bmatrix} = \begin{bmatrix} S^{SS}_{2;0,0} & S^{SS}_{2;0,1} & S^{SS}_{2;0,2} & S^{SS}_{2;0,3} \\ S^{SS}_{2;1,0} & S^{SS}_{2;1,1} & S^{SS}_{2;1,2} & S^{SS}_{2;1,3} \\ S^{SS}_{2;2,0} & S^{SS}_{2;2,1} & S^{SS}_{2;2,2} & S^{SS}_{2;2,3} \\ S^{SS}_{2;3,0} & S^{SS}_{2;3,1} & S^{SS}_{2;3,2} & S^{SS}_{2;3,3} \end{bmatrix}, \tag{5.11}$$

which corresponds to the "Merge" and "Permutation" phases in Figure 5.2. As a consequence, the shared bases at this level ($U_{1;i}$) are the transfer matrices along with their redundant parts as shown in Equation (5.3). Therefore, now we can see this level=1 as our new "leaf-level" and proceed to continue the factorization as shown in Figure 5.3.

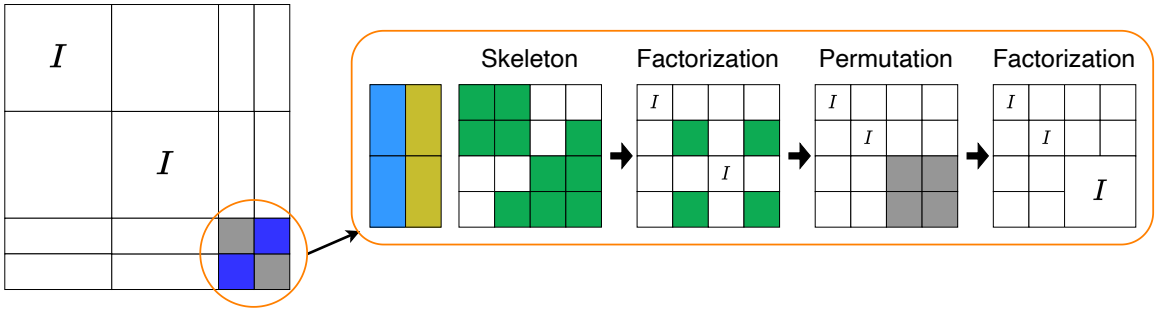


Figure 5.3: Flow of generalized HSS-LDL factorization at the upper level

The flow at the upper level starts by incorporating the redundant part of the shared bases to form the skeleton matrices of both dense and low-rank blocks. For each dense diagonal block $\hat{S}_{1;i,i}$

we obtain its skeleton matrix

$$\begin{bmatrix} S^{RR}_{1;i,i} & S^{RS}_{1;i,i} \\ S^{SR}_{1;i,i} & S^{SS}_{1;i,i} \end{bmatrix} = \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}^T \hat{S}_{1;i,i} \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}. \tag{5.12}$$

For each off-diagonal low-rank block $\hat{S}_{1;i,j}$ $(i \neq j)$ we compute

$$\begin{aligned} \begin{bmatrix} S^{RR}_{1;i,j} & S^{RS}_{1;i,j} \\ S^{SR}_{1;i,j} & S^{SS}_{1;i,j} \end{bmatrix} &= \begin{bmatrix} U^R_{1;i} & U^S_{1;i} \end{bmatrix}^T \hat{S}_{1;i,j} \begin{bmatrix} U^R_{1;j} & U^S_{1;j} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & U^S_{1;i} \hat{S}_{1;i,j} U^S_{1;j} \end{bmatrix}. \end{aligned} \tag{5.13}$$

Note that this process corresponds to the "Skeleton" phase in Figure 5.3 and it is similar to that of Equations (5.5)-(5.6). Next, we factorize the redundant parts again by following the steps as in Equations (5.7)-(5.9) followed by permuting and merging into the upper level. Then we recursively perform the same procedure until the root level is reached and we are left with a single dense block, which will be factorized as a single dense matrix.

At the end of the whole process, we have a factorization in the form of

$$\tilde{A} = \mathcal{L}\mathcal{D}\mathcal{L}^T, \tag{5.14}$$

where $\tilde{A}$ is an HSS matrix, $\mathcal{L}$ is a product of lower triangular and orthogonal matrices and $\mathcal{D}$ is a diagonal matrix. Note the equal sign in Equation (5.14) since the generalized LDL factorization that we just described does not involve any form of approximation, i.e. the given HSS matrix is factorized in an almost exact manner up to machine precision. The overall cost of this method is $\mathcal{O}(r^2 n)$ flops, where r is the maximum compression rank of the off-diagonal blocks. The complexity analysis of this method directly follows from the generalized Cholesky factorization, so we refer the reader to [89] for a detailed discussion.

5.2.3 Slicing the Spectrum with Generalized LDL Decomposition of HSS Matrices

Consider the generalized LDL decomposition in Equation (5.14). Since the matrix $\mathcal{L}$ is a product of lower triangular and orthogonal matrices, then it must be invertible. Therefore, we can use the bisection eigenvalue solver in Algorithm 15 to compute the k-th eigenvalue of HSS matrices by replacing the LDL decomposition with the corresponding generalized LDL decomposition. We call this method the *HSS-Bisection*. This method requires $\mathcal{O}(r^2 n)$ flops per bisection iteration where r is the HSS off-diagonal compression rank. This results in an overall cost of

$$\mathcal{O}(r^2 n \log_2((b-a)/\epsilon_{ev})) \tag{5.15}$$

flops to find the k-th eigenvalue with an accuracy of ϵ_{ev} from an initial interval of $[a, b]$. This is a significant reduction compared to the dense counterpart that requires $\mathcal{O}(n^3)$ flops for tridiagonal reduction to obtain $\mathcal{O}(n)$ cost per iteration.

The first and main advantage of the HSS-Bisection is that it allows fast inertia evaluation by reusing a significant portion of work from one inertia computation for all subsequent inertia computations with different shifts. Notice that the slicing-the-spectrum requires inertia evaluation of varying shifts to find the k-th eigenvalue, meaning that the generalized LDL decomposition of the form

$$\tilde{A} - \mu I = \mathcal{L}_\mu \mathcal{D}_\mu \mathcal{L}_\mu^T \tag{5.16}$$

for different shifts μ need to be performed. However, a closer look at the LDL decomposition in Section 5.2.2 reveals that the shared bases and skeleton matrices of the off-diagonal low-rank blocks stay constant for different shift values since they are independent of the shifted diagonal entries. Therefore, we can pre-compute the skeleton and redundant parts of the shared bases, along with the skeleton matrices for all off-diagonal low-rank blocks (that is readily available from the HSS matrix input) only once and reuse them for computing LDL decomposition of different shifted matrices. This is one of the main advantages of inertia evaluation using generalized HSS-LDL decomposition in contrast to the pure Cholesky-based existing methods in [14, 16] that may need to perform shared bases and low-rank updates for every shift value.

The second advantage is in terms of accuracy. Since the generalized LDL decomposition shown in Equation (5.14) is performed in (almost) exact manner, aside from the bisection threshold ϵ_{ev}, the accuracy of the computed eigenvalue depends only on the accuracy of the HSS approximation $\tilde{A}$ so that the inertia of the original matrix A is preserved, that is

$$\text{Inertia}(A - \mu I) = \text{Inertia}(\tilde{A} - \mu I).$$

This has been discussed in [88], where it has also been shown that aggressively using small HSS compression rank even when the off-diagonal blocks have slow decaying singular values can preserve the inertia for certain shifts, especially for computing the eigenvalues that are close to the largest eigenvalue.

However, the main limitation of the HSS-Bisection is in its application to matrices originating from 3D problems. In such cases, the HSS-compression rank r often grows proportionally with the matrix dimension n, making the cost for one generalized LDL decomposition become $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n)$. Although the report in [88] suggests a potential application of the HSS-Bisection to certain 3D problems by aggressive truncation, we expect the method to suffer from high truncation rank in general 3D problems, especially when the part of the spectrum that is far from the largest eigenvalue is desired.

5.2.4 Parallelization

In the context of computing a single eigenvalue, one possible location to exploit parallelism is the generalized LDL decomposition. As we have seen in Section 5.2.2, the formation of dense diagonal

skeleton matrices and their partial factorization within each level are independent of each other, so they can be performed in an embarrassingly parallel fashion. On a single node, this can be done by simply parallelizing the loop with multiple threads. In order to parallelize across multiple nodes, we follow the conventional 1D row data distribution scheme. The advantage of using the 1D row data distribution is to maximize the data locality when merging between the different levels of factorization. Communication is only needed beyond level $l = \log_2 P$ where P is the number of processes, as all the data required before that point is present locally.

In Figure 5.4, we illustrated the data and process distribution scheme by coloring data located on different process ranks. The LDL decomposition follows a bottom-up order in the partition tree, and the places where inter-level merges are needed are represented as edges in the tree. It is obvious from the illustration that the parallel elimination of the lower levels does not need communications amongst nodes. When closer to the root, the communication happens as an `MPI_AllReduce` using split communicators. For a binary tree partition, the communicator size for merging is exactly 2, and for other tree configurations, the communication volume at each level is also a small constant number bounded by the number of children of each node in the process tree.

The complexity of the communication cost can be determined as a constant when the number of processes is fixed. With an increasing number of processes and finer partitioning, the amount of communication for each level is still bounded by a constant. However, an increasing number of processes also enlarges the number of levels where communication is needed, making the eventual communication complexity grow as $O(\log_2 P)$.

On the other side, dense linear algebra libraries that run on distributed memory environments typically use 1D or 2D block-cyclic data distribution schemes. When executing ScaLAPACK routines such as LU decomposition or eigen decomposition, the number of communicating neighbors is $\mathcal{O}(P)$ and $\mathcal{O}(\sqrt{P})$ correspondingly for 1D and 2D block-cyclic data distributions, and the message sizes are typically dependent on problem size n. Although we are using a simpler non-cyclic variant data distribution scheme, we expect the simple 1D row data distribution to show an advantage in communication scalability when compared with the more complex schemes that were made to accommodate dense matrix algorithms.

Further, when multiple eigenvalues are desired, we can extend the approach above by dividing the processes into groups and letting each group performs HSS construction redundantly and then compute the target eigenvalues in parallel. This is still efficient thanks to the small memory consumption of the HSS matrix, so even if one process computes one eigenvalue, a relatively large problem size can still be handled.

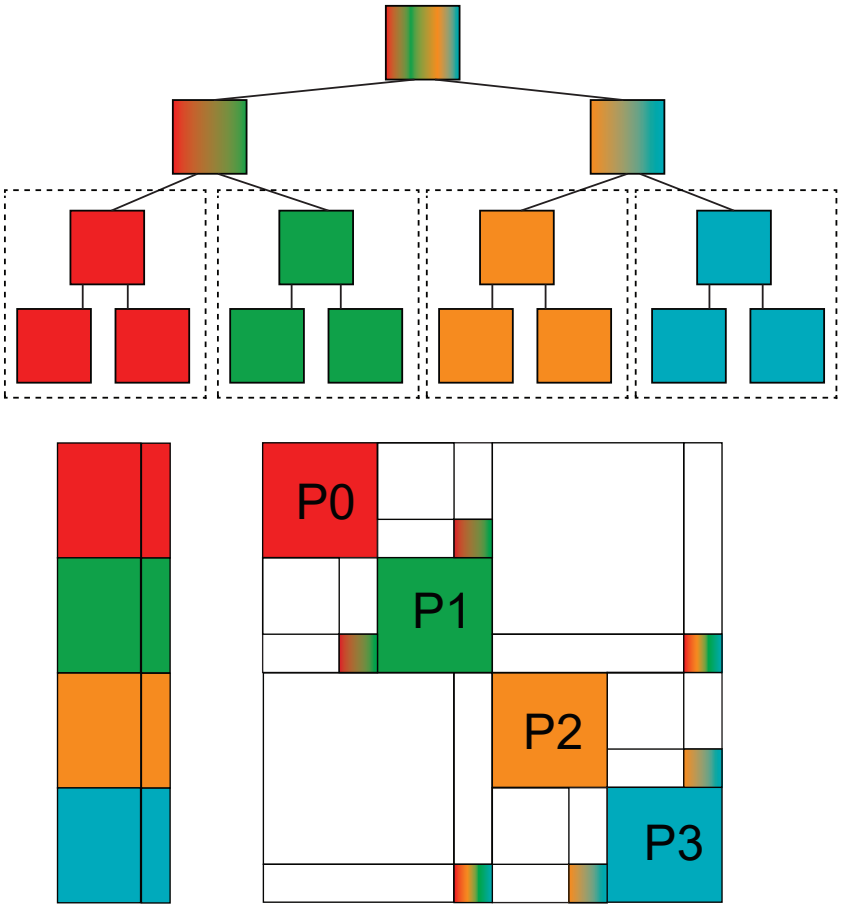


Figure 5.4: The data distribution scheme of the HSS matrix for generalized LDL decomposition

5.2.5 Numerical Results

We present numerical results on two different sets of synthetically generated rank-structured matrices to demonstrate the performance and accuracy of our method using both single and multiple computing nodes. Our implementations were written in C++, with floating point calculations performed in double precision. BLAS and LAPACK routines were used for inner kernels involving dense matrices. Multithreading within a node was performed using OpenMP, whereas distributed memory parallelism was performed using MPI. Our code is publicly available in GitHub[1].

We compared our HSS-Bisection method that computes the k-th eigenvalue with controllable accuracy against LAPACK dense symmetric eigenvalue solver `dsyev` that computes all eigenvalues in double precision accuracy. The eigenvalue error was obtained by comparing the computed k-th eigenvalue with the corresponding eigenvalue produced by `dsyev`. Note that although there exists

[1] `https://github.com/rioyokotalab/nbd-cpu-only/tree/hss-eigen`

another LAPACK routine `dsyevx` that can compute only the k-th eigenvalue of a given dense matrix with controllable accuracy, our preliminary experiments found that it took about the same amount of time for it to compute one eigenvalue as the time it takes `dsyev` to compute all eigenvalues of the same matrix. Therefore, we chose to only compare with `dsyev`.

Also note that in all experiments we used a uniform rank r to compress all off-diagonal blocks of the HSS matrices, which we expect to give a good load balance of the task among different computing resources. In order to ensure that the inertia of the original matrix A was preserved in the HSS approximation so that we can produce accurate eigenvalue down to the bisection threshold ϵ_{ev}, we have selected the rank so that the compression error was much smaller than ϵ_{ev}. The HSS construction was performed based on the idea of sample points described in [25] with a simple uniform index-based selection for the sampling algorithm.

5.2.5.1 Single Node Performance and Accuracy

Here we show the accuracy and performance of our method using a single node of the Wisteria-A subsystem of the BDEC-01 supercomputer, equipped with two Intel Xeon Platinum 8360Y CPUs (2x36 = 72 cores) and 512 GiB of memory. We used sequential Intel MKL 2023.0 for BLAS/LAPACK kernels and Intel OpenMP for multithreading. We compared our implementation against LAPACK `dsyev` from parallel MKL.

For the test matrices, we used the 2D Laplace kernel to generate dense matrices with low-rank off-diagonal blocks such that

$$a_{i,j} = \begin{cases} 1000, & \text{if } dist(i,j) = 0 \\ ln(dist(i,j)), & \text{otherwise} \end{cases},$$

where $dist(i,j)$ denotes the Euclidean distance between two points x_i and x_j that were uniformly distributed on the circumference of a unit circle.

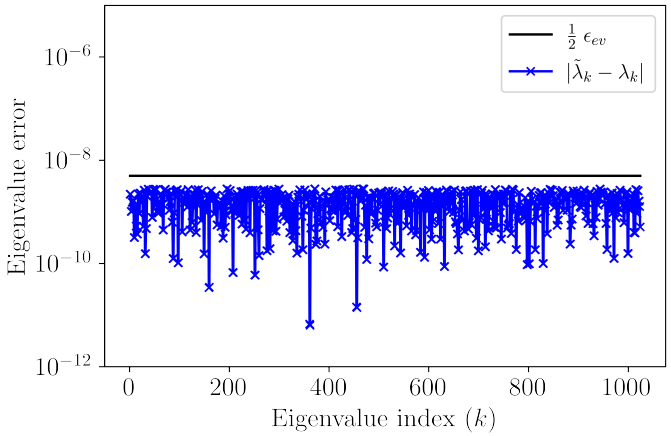


Figure 5.5: Eigenvalue errors for a 1024 × 1024 Laplace matrix

We first show that our method produces accurate eigenvalues in accordance with the theoretical bound. Figure 5.5 presents the eigenvalue errors when computing all eigenvalues of a 1024×1024 matrix using the HSS-Bisection method with compression rank r=48 and bisection threshold $\epsilon_{ev} = 10^{-8}$. It can be seen that all of the eigenvalue errors lie below the theoretical error bound in Equation (2.55). As we varied the matrix size and bisection threshold, our method still produced accurate

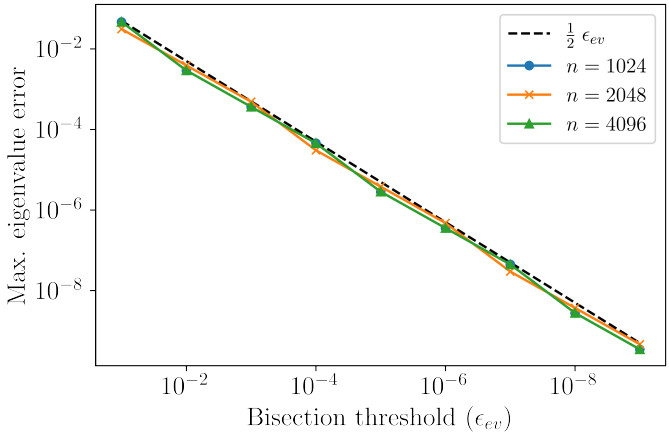


Figure 5.6: Maximum eigenvalue errors when computing all eigenvalues of Laplace matrices

eigenvalues, as presented in Figure 5.6.

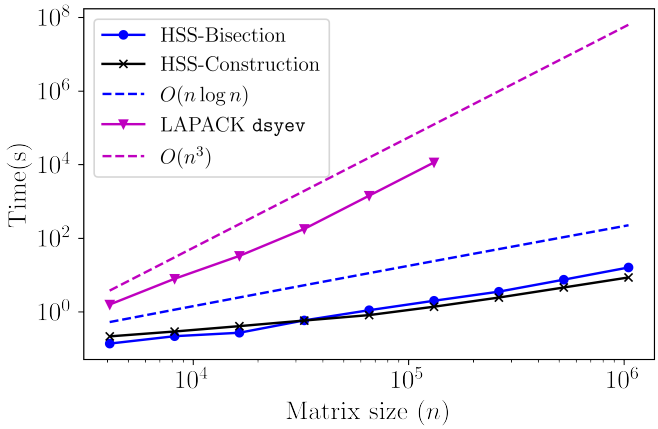


Figure 5.7: Time to compute the k-th eigenvalue on a single node

Next, we show in Figures 5.7 and 5.8 the execution time and memory consumption of our method when computing the k-th eigenvalue with varying matrix sizes using a single node, utilizing all available 72 cores of the machine. We assumed b=128, r=80, k=$n/2$, $\epsilon_{ev} = 10^{-7}$, and starting interval $[-n/2, 1024 + \log_2(n)]$ that we have found to contain all eigenvalues of the $n \times n$ Laplace matrix from a preliminary experiment on some small matrices. The data points for LAPACK `dsyev` are limited to $n \leq 131{,}072$ because the larger dense matrices did not fit into the memory of a single

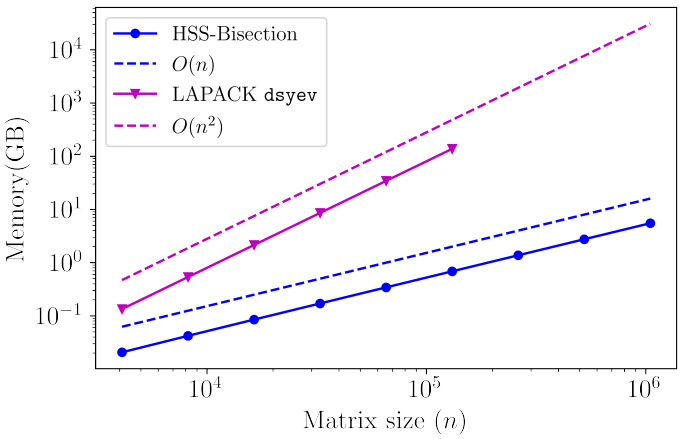


Figure 5.8: Memory consumption when computing the k-th eigenvalue on a single node

node. Figure 5.7 shows that our method exhibits log-linear scaling with respect to the matrix size. This is expected because the size of our choice of starting interval grows linearly with the matrix size, adding a $\log n$ term. This term can be removed if prior knowledge of the matrix spectrum is available beforehand and if the spectrum does not grow with the matrix size. In addition, the HSS construction times were comparable to the eigenvalue computation times, whereas for large matrices the construction times were smaller. Furthermore, the figure also demonstrates that our method consistently required smaller amount of time than LAPACK `dsyev` to compute the target k-th eigenvalue. For a matrix size of n=131,072, our method was about 3 orders of magnitude faster.

Figure 5.8 shows that our method scaled linearly with respect to the matrix size in terms of memory consumption. For the matrix of order n=131,072, our method only required about 0.68 GB of memory, while LAPACK `dsyev` required about 137 GB. This small memory consumption is another advantage that allows our method to solve larger problems on a single node.

5.2.5.2 Parallel Scalability

Here we show the performance of our method using up to 512 nodes of the Wisteria-O subsystem of the BDEC-01 supercomputer connected with Tofu InterConnect-D, where each node was equipped with a Fujitsu A64FX CPU (48 cores) and 32 GiB of memory. We used Fujitsu implementations of sequential BLAS/LAPACK kernels, OpenMP for multithreading, and MPI for distributed memory parallelism. We compared our method against the Fujitsu implementation of ScaLAPACK `pdsyev` linked with multi-threaded BLAS/LAPACK.

For the test matrices, we used the 2D Yukawa kernel to generate dense matrices with low-rank off-diagonal blocks such that

$$A_{i,j} = \begin{cases} 1000, & \text{if } dist(i,j) = 0 \\ e^{\frac{-1 dist(i,j)}{dist(i,j)}}, & \text{otherwise} \end{cases},$$

where $dist(i, j)$ denotes the Euclidean distance between two points x_i and x_j that were uniformly distributed on the circumference of a unit circle. Preliminary experiments on some small matrices showed us that a starting interval of $[-n/2, 4n]$ contains all eigenvalues of the Yukawa matrix of order n. We assumed the bisection threshold of $\epsilon_{ev} = 10^{-7}$ for all experiments in this section, which is usually enough for major applications, such as electronic structure calculations [53].

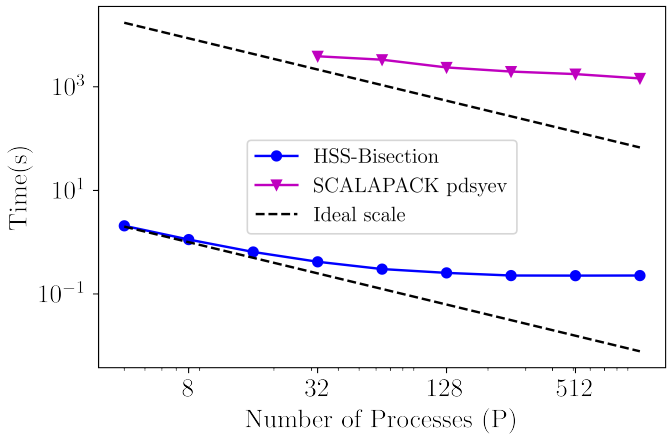


Figure 5.9: Strong scaling of time to compute one eigenvalue with increasing number of processes

The strong scaling result is presented in Figure 5.9 for a fixed matrix size n=131,072 and an increasing number of nodes up to 256. We used b=64 and r=50 for the HSS compression. For both ScaLAPACK and our method, we spawned 4 MPI processes per node and assigned 12 threads per process. The data points for ScaLAPACK `pdsyev` start from 8 nodes due to the dense matrix size that requires a minimum number of 8 nodes to fit into the memory. Our method showed good strong scaling at the beginning, but the speedup started to decay as we use a larger number of nodes. This is because keeping the problem size fixed while increasing the number of processes reduces the amount of work per process and at the same time increases the number of levels where communication is needed. Therefore, it is expected that at some point the problem size will become too small to fully utilize all of the available computing resources and the communication time starts to dominate. In the particular case that we were testing, there were 131,072/64=2,048 nodes in the leaf level of the HSS partition tree. Therefore, when $P \geq 256$, each process will be responsible for less than 12 nodes in the leaf level, making it unable to fully utilize all the available 12 threads within a process. Results using up to 256 nodes show that our method is about 6,000x faster than ScaLAPACK.

We also present the weak scaling result in Figure 5.10 to further demonstrate the parallel scalability of our method. Here we spawned 1 MPI process per node and assigned each process with 48 threads. For ScaLAPACK, the matrix size started at 16,384 using 1 node. We then increased the number of nodes by a factor of 8 as we doubled the matrix size so that we have a matrix size of

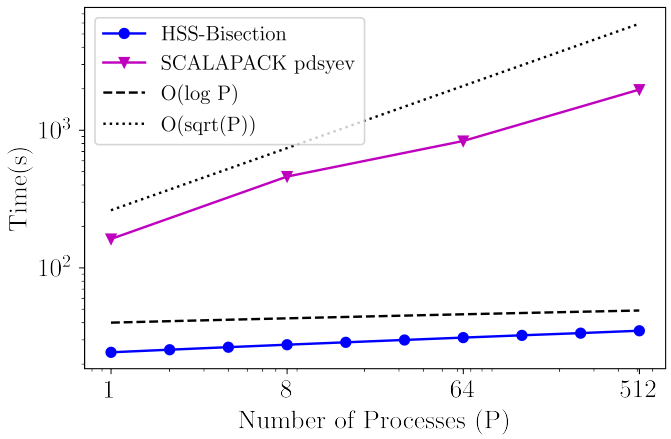


Figure 5.10: Weak scaling of time to compute one eigenvalue with increasing number of processes

131,072 when using 512 nodes. This was done to maintain constant work per node considering the $\mathcal{O}(n^3)$ arithmetic complexity of `pdsyev`. For our HSS-Bisection method, we started with a matrix size of 32,768 with 1 node and increased the number of nodes linearly with the matrix size due to the $\mathcal{O}(n)$ arithmetic complexity. Therefore, at 512 nodes we have a maximum matrix size of 16,777,216, where we were able to compute its k-th eigenvalue in about 34 seconds. Our method exhibited better weak scalability than ScaLAPACK `pdsyev`, showing almost perfect scaling with respect to our theoretical estimate of the communication time $O(\log_2 P)$. ScaLAPACK on the other hand, has higher communication times that are expected to grow as $\mathcal{O}(\sqrt{P})$ due to the 2D block-cyclic data distribution. This confirmed our estimate that our simple 1D row data distribution combined with the inherent parallelism of generalized HSS-LDL factorization leads to better communication scalability compared to the more complex data distribution schemes that were originally made for general dense matrix algorithms.

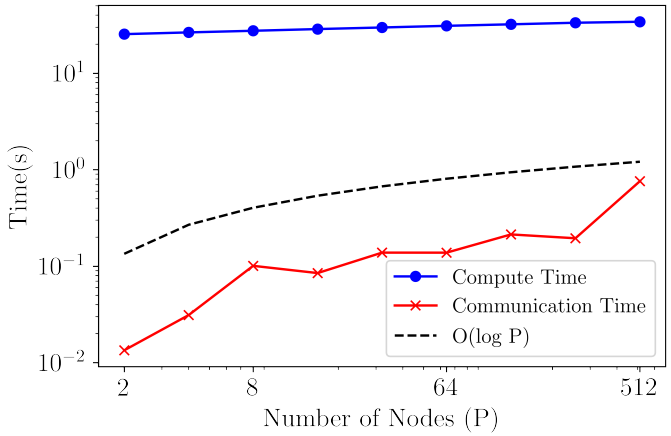


Figure 5.11: Performance breakdown of HSS-Bisection in Figure 5.10

We also present the performance breakdown of the weak-scaling experiment of our HSS-Bisection in Figure 5.11. The computation times were flat as we increased the number of nodes, meaning that the same amount of work was being done by each node as we increased the problem size proportional to the number of nodes.

5.3 Computing the k-th Eigenvalue of Symmetric $\mathcal{H}^2$-Matrices

Here we present an algorithm to compute the generalized LDL decomposition of $\mathcal{H}^2$ matrices along with its application to solve the k-th eigenvalue problem. We first describe the proposed algorithm in Section 5.3.1. We then discuss its application to the slicing-the-spectrum method and the parallelization in Sections 5.3.2 and 5.3.3, respectively. Finally, we demonstrate the performance and accuracy of our method in Section 5.3.4.

5.3.1 Generalized LDL Decomposition of $\mathcal{H}^2$-Matrices

Here we further extend the generalized LDL factorization for HSS matrices in Section 5.2.2 to $\mathcal{H}^2$ matrices by introducing dense off-diagonal blocks. We take as an example a 2-level $\mathcal{H}^2$ matrix $\tilde{A}$ with dense off-diagonal blocks as shown in the leftmost part of Figure 5.12. While this example has a rather simple block tri-diagonal structure, the same steps can be applied to general $\mathcal{H}^2$ matrices with arbitrary subdivision levels and any pattern of dense off-diagonal blocks.

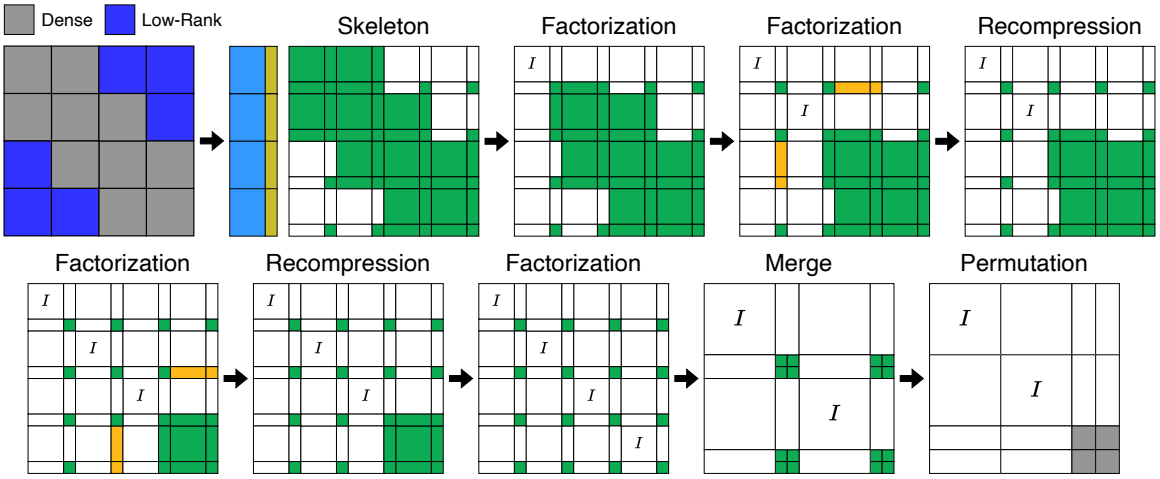


Figure 5.12: Flow of Generalized $\mathcal{H}^2$-LDL Factorization at the leaf level

The flow at each level begins by forming the skeleton matrices using both the redundant and skeleton part of the shared bases, which is identical to the BLR^2 version described in Equations (5.5)-(5.6). This corresponds to the "Skeleton" phase in Figure 5.12. Note that dense off-diagonal blocks are never used for the construction of shared bases. Also, the skeleton matrices of dense off-diagonal blocks take a similar form as the dense diagonal blocks, meaning that Equation (5.5) also applies to dense off-diagonal blocks.

The next step is to factorize the redundant part of the skeleton matrices. As can be seen from the figure, dense off-diagonal blocks lead to the computation of Schur's complements in the trailing submatrices. When it fills into a low-rank block, it creates a fill-in block that may destroy the sparsity of the matrix, shown as orange blocks in Figure 5.12. In order to mitigate this, the fill-ins in an entire row/column i are incorporated into the shared basis just before the i-th diagonal block is factorized. This corresponds to the "Recompression" step where the orange blocks are merged into the shared basis and disappear into the low-rank block. For example in the second matrix from the right on the top part of Figure 5.12, there are two fill-in blocks within $S_{2;0,2}$ and $S_{2;2,0}$. So, before factorizing $S^{RR}_{2;2,2}$, a recompression step is performed that involves an update to the column basis using the low-rank approximation, for example using the rank-revealing QR decomposition of

$$U^S_{2;2}, R \leftarrow RRQR\left(\begin{bmatrix} U^S_{2;2} S^{SS}_{2;2,0} & \begin{bmatrix} U^R_{2;2} & U^S_{2;2} \end{bmatrix} \begin{bmatrix} F^{RS}_{2;2,0} \\ F^{SS}_{2;2,0} \end{bmatrix} \end{bmatrix}\right), \tag{5.17}$$

where $F_{2;2,0}$ is the corresponding fill-in block within $S_{2;2,0}$. This incorporates the fill-ins to the shared column basis so that they can be merged into $S^{SS}_{2;0,2}$ and $S^{SS}_{2;2,0}$, thus keeping the sparsity of the skeleton matrices intact. Note that in this case there is only one low-rank block in the particular row/column. However, in general it is necessary to concatenate the products for all low-rank blocks in the row/column that is being recompressed, which for example can be done efficiently using the technique described in [63].

Once the redundant part of the skeleton matrices are factorized, we merge and permute the remaining blocks then recursively apply the same procedure to the upper levels until we reach a level that has no admissible blocks. Once such level is reached, we factorize the remaining blocks as a single dense matrix, similar to the BLR^2 case in Equation (5.10). The overall cost of the algorithm is $\mathcal{O}(n)$ as long as the numerical rank of the off-diagonal low-rank blocks grow independently of the matrix size (see the discussion in [63]). At the end of the whole process, we are left with an approximate factorization in the form of

$$\tilde{A} \approx \mathcal{L}\mathcal{D}\mathcal{L}^T, \tag{5.18}$$

where $\mathcal{L}$ is a product of lower triangular and orthogonal matrices and $\mathcal{D}$ is a diagonal matrix. Note that here we only have an approximate factorization in contrast to the (almost) exact factorization in the HSS case shown in Equation (5.14). This is the cost of incorporating the fill-in blocks into the shared bases that ultimately amounts to approximating the Schur's complement.

5.3.2 Slicing the Spectrum with Generalized LDL Decomposition of $\mathcal{H}^2$-Matrices

Since the matrix $\mathcal{L}$ in Equation (5.18) is a product of lower triangular and orthogonal matrices, then it must be invertible. Thus, we can also use the bisection eigenvalue solver described in Algorithm 15 to compute the k-th eigenvalue of symmetric $\mathcal{H}^2$ matrices by replacing the LDL decomposition with

the corresponding generalized LDL decomposition. We call this method the $\mathcal{H}^2$-*Bisection*. This method requires $\mathcal{O}(n)$ flops per bisection iteration, leading to an overall cost of

$$\mathcal{O}(n \log_2((b-a)/\epsilon_{ev})) \tag{5.19}$$

flops to find the k-th eigenvalue with an accuracy of ϵ_{ev} from an initial interval of $[a, b]$.

The main advantage of the $\mathcal{H}^2$-Bisection is that it can maintain the optimal (near) linear cost per iteration even on the more general structured low-rank matrices originating from 3D problems. This allows us to solve a wide range of practical problems where the HSS-Bisection is hampered by high off-diagonal compression rank, which will be shown later in Section 5.3.4. Consequently, it comes with the following limitations:

- Because of the recompression step that depends on the shifted diagonal entry, reusing the work for inertia evaluation of different shifts as in the HSS-Bisection is not possible due to the shared bases being modified during the factorization.

- Since the generalized LDL decomposition shown in Equation (5.18) is approximative, we must make sure that the decomposition is accurate enough to produce the correct inertia evaluation of the shifted matrices. As discussed in [71], the generalized LDL decomposition is sufficiently accurate if it satisfies

$$\left\| \left(\tilde{A} - \mu I \right) - \mathcal{L}_\mu \mathcal{D}_\mu {\mathcal{L}_\mu}^T \right\| \leq \min_j \left| \lambda_j(\tilde{A}) - \mu \right|, \tag{5.20}$$

  where $\tilde{A} - \mu I \approx \mathcal{L}_\mu \mathcal{D}_\mu {\mathcal{L}_\mu}^T$ is the generalized LDL decomposition of the shifted matrix and $\lambda_j(\tilde{A})$ is the exact j-th eigenvalue of $\tilde{A}$. Therefore, we need an error bound of the form

$$\left\| \tilde{A} - \mathcal{L}\mathcal{D}\mathcal{L}^T \right\| \leq \epsilon. \tag{5.21}$$

  In addition, we also need this bound for all shifted matrices $\tilde{A} - \mu I$. To the best of our knowledge, such a bound is not available in the current literature on hierarchical matrices, especially for an $\mathcal{H}^2$ matrix $\tilde{A}$. Although it has been mentioned in [63] that the accuracy of the similar $\mathcal{H}^2$-ULV decomposition is directly controlled by the truncation rank used in the recompression step, there has been no discussion about the theoretical error bound. Thus at the moment, we do not have theoretical proof that the proposed generalized $\mathcal{H}^2$-LDL decomposition is accurate for slicing the spectrum in general. Fortunately, many numerical experiments have shown that $\mathcal{H}^2$-ULV decomposition produces satisfying accuracy [62, 63, 64]. Thus, we provide evidence that our method is sufficiently accurate through the numerical experiments shown in Section 5.3.4.

5.3.3 Parallelization

If only one eigenvalue is desired, one possible location to exploit parallelism is the generalized $\mathcal{H}^2$-LDL factorization, just like what we did for the HSS-Bisection in Section 5.2.4. However, in our generalized $\mathcal{H}^2$ LDL decomposition, the dependencies between the fill-in recompression and diagonal factorization limit the parallelization within one level to the triangular solves and schur complements involving dense blocks. In order to extract the parallelism across multiple levels, it requires using a task-based runtime system to resolve the task dependencies. As such runtime systems often impose implementation challenges on distributed memory systems, we would like to address it in our future works.

On the other hand, if more than one eigenvalue is desired, we can parallelize the bisection part to compute them efficiently. There are two key ideas for this:

- Some of the computed inertia can be reused for the computation of other eigenvalues. For example, in the beginning, the same starting interval $[a, b]$ is assumed for the computation of all target eigenvalues. Suppose after the first inertia evaluation we obtain $\nu(A - \mu I) = v$. Then we know that the interval $[a, \mu)$ contains the first v eigenvalues and $[\mu, b]$ contains the remaining $n - v$ eigenvalues. Thus, for the computation of the k-th eigenvalue where $k \leq v$, we can narrow the starting interval to $[a, \mu)$. The same applies for $k > v$ where the starting interval of $[\mu, b]$ can be used.

- The bisection of disjoint intervals are independent of each other so they can be done in parallel. For this, each process needs to hold an instance of the matrix along with the interval of interest. Although this is quite prohibitive for dense matrices that require $\mathcal{O}(n^2)$ storage, for $\mathcal{H}^2$-matrices that require only $\mathcal{O}(n)$ storage this is still feasible even for relatively large matrices.

Thus, given the starting interval $[a, b]$ and a range of eigenvalue indices $[k_0, k_1]$ to be computed, we consider two approaches to compute the eigenvalues in parallel on distributed memory systems: one that distributes the eigenvalue indices equally to every process, and another that uses a master-worker model. In both approaches, we assume that every process holds an instance of the $\mathcal{H}^2$ matrix whose eigenvalues will be computed.

In the first parallel algorithm, we distribute the eigenvalue index range $[k_0, k_1]$ equally to P processes. Every process will first compute the smallest and largest eigenvalue from the set it is responsible for. After that, it computes the remaining inner eigenvalues using the smallest and largest eigenvalues as the starting interval. This method does not involve any communication during the eigenvalue computation since every process knows the set of indices it has to compute. However, this method may suffer from load imbalance when the eigenvalues are not uniformly distributed along the spectrum since some processes may be responsible for larger intervals than others even

though the number of eigenvalues to be computed are the same. Hence, this method works best if the number of processes is close to the total number of eigenvalues to be computed. That way, every process would be responsible for a smaller part of the spectrum, thus reducing the overall load imbalance.

For the second algorithm, we adopt a master-worker model similar to the one discussed in [65]. The master distributes the works by giving each idle worker an interval and a range of eigenvalue indices. The worker computes all eigenvalues within the given interval if there are no more than $2m$ eigenvalues to be computed. Otherwise, the worker will only compute the inner m eigenvalues. The worker first computes the smallest and the largest eigenvalues within the set and notify the master that it will not compute the eigenvalues within the other two intervals, i.e. $[a, \tilde{\lambda}_k]$ and $[\tilde{\lambda}_{k+m-1}, b]$ where $[a, b]$ is the interval received by the worker and $\tilde{\lambda}_k$ and $\tilde{\lambda}_{k+m-1}$ are the smallest and largest eigenvalues from the chosen set of m inner eigenvalues, respectively. After that, the worker continues to compute the rest of the eigenvalues in the set. The master manages a list of all free intervals and distributes them immediately to idle workers. The value m is set to be small enough to balance the load among workers. This approach works well when computing a large number of eigenvalues, as reported in [65, 16].

5.3.4 Numerical Results

In this section, we use several examples to demonstrate the performance and accuracy of our proposed $\mathcal{H}^2$-Bisection method. Implementations were written in C++ where floating point calculations were performed in double precision. Single-threaded BLAS and LAPACK routines from Intel MKL were used for inner kernels involving dense matrices. Distributed memory parallelization was performed using Intel MPI. Experiments were conducted on a system described in Table 4.8.

The following methods have been compared:

- **$\mathcal{H}^2$-Bisection**: Slicing the spectrum with our generalized $\mathcal{H}^2$-LDL factorization described in Section 5.3.2.
- **HSS-Bisection**: Slicing the spectrum with generalized HSS-LDL factorization [88].
- **LAPACK** `dsyevx`: Dense symmetric eigenvalue solver that computes selected eigenvalue(s) with controllable accuracy ϵ_{ev} [8].
- **ScaLAPACK** `pdsyev`: Distributed-memory parallel dense symmetric eigenvalue solver that computes all eigenvalues in double precision accuracy [18].
- **ELPA**: Distributed memory parallel dense eigenvalue solver that computes all eigenvalues using the 2-stage solver presented in [66].

In order to ensure that the bisection methods produce accurate inertia evaluation, we used the low-rank compression threshold that was smaller than the bisection threshold, e.g. $\epsilon_{\mathcal{H}^2} = 10^{-2}\epsilon_{ev}$. The eigenvalue error was obtained by comparing the approximated eigenvalue with the result from LAPACK dense symmetric eigenvalue solver `dsyev`. We also provide the time to construct the $\mathcal{H}^2$ matrices using the linear time construction scheme presented in [25] with the Anchor-Net sampling algorithm.

5.3.4.1 Computing Some or All Eigenvalues of Synthetic Matrices

Here we used the Laplace kernel to generate the rank-structured matrix

$$a_{i,j} = \frac{1}{|x_i - x_j| + 10^{-3}},$$

where $|x_i - x_j|$ denotes the Euclidean distance between two points x_i and x_j. The points x_i were uniformly distributed on the circumference of a unit circle.

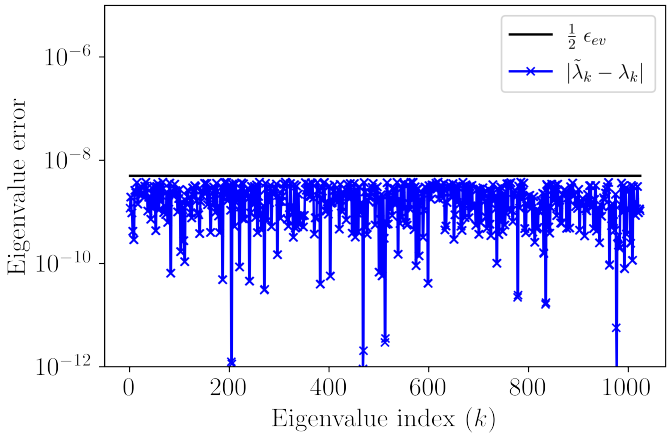


Figure 5.13: Eigenvalue errors for a 1024×1024 synthetic Laplace matrix ($\epsilon_{ev} = 10^{-8}$)

We first show that our method produces accurate eigenvalues according to the bisection threshold. If every inertia evaluation is sufficiently accurate, the bisection will choose the correct interval that contains the target eigenvalue. Otherwise, it will choose the wrong interval that does not contain the target eigenvalue, leading to an error larger than the theoretical bound in Equation (2.55). Figure 5.13 shows that all of the eigenvalue errors lie below the theoretical error bound. Note that the eigenvalue error corresponds to the distance of the actual eigenvalue to the midpoint of our bisection interval in the last iteration. So different choices of starting interval may give slightly different errors, and in some cases, might result in an eigenvalue error that is much smaller than the desired accuracy, as shown in Figure 5.13 where some of the eigenvalue errors are smaller than 10^{-10} even when $\epsilon_{ev} = 10^{-8}$. Figure 5.14 also shows that our method consistently produced accurate eigenvalues as we decreased the bisection threshold. We did not observe a high rank growth that is

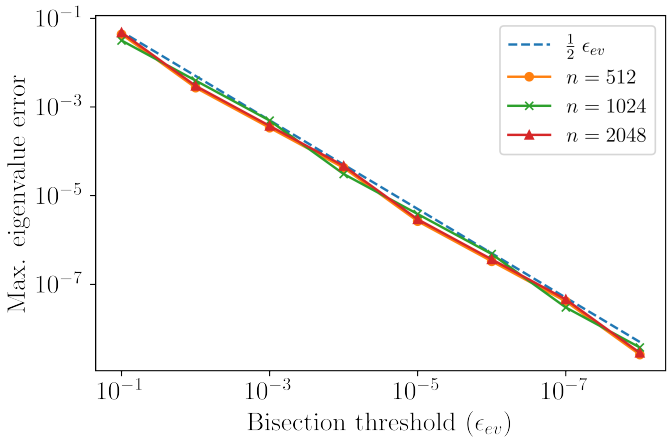
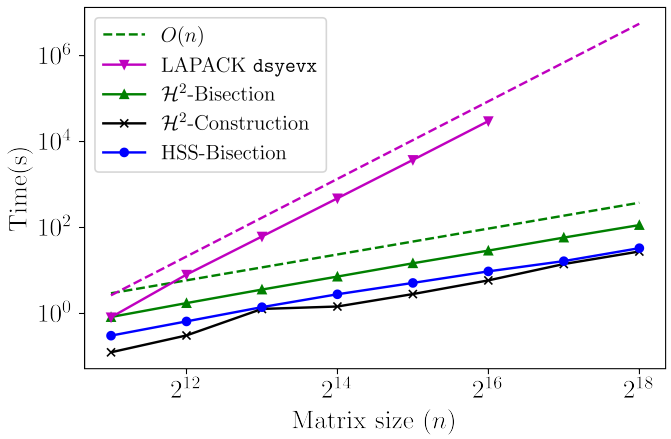


Figure 5.14: Maximum eigenvalue errors when computing all eigenvalues of synthetic Laplace matrices

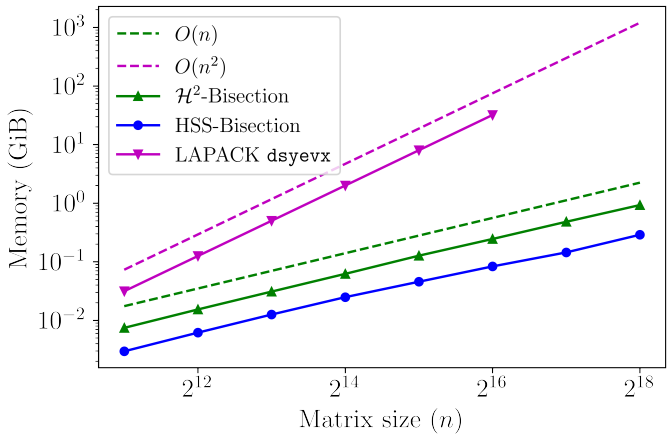
proportional to the matrix size when using shifts near the actual eigenvalues as reported in [14] for $\mathcal{H}$-LDL factorization, but we do not have a theoretical bound for the ranks.

Next, we show the performance of our method to compute the k-th eigenvalue where $k = n/2$ in Figure 5.15. A constant starting interval of $[0, 2048]$ was used for the bisection methods, which we found still contain the target (median) eigenvalue even if we increased the matrix size. Figure 5.15a shows that with constant accuracy and size of starting interval, our $\mathcal{H}^2$ bisection method scaled linearly with respect to the matrix size in terms of both computation time and memory consumption, which is in accordance with our estimate in Section 5.3.2. The HSS bisection method also showed a similar linear scaling since the test matrices originate from simple 2D geometries where HSS could still maintain a constant rank as we increased the problem size. We observed a maximum compression rank of 17 for the $\mathcal{H}^2$-Bisection and 50 for the HSS-Bisection. With this small difference in the compression rank, our method was slightly slower than the HSS one due to the hidden constant number of operations coming from the fill-in recompression steps that did not occur in generalized HSS-LDL factorization. Moreover, both bisection methods outperformed LAPACK `dsyevx` in terms of computation time and memory consumption. For the matrix of order 65,536, our bisection method was already about 3 orders of magnitude faster than LAPACK `dsyevx`. For the large matrix of order $n = 262,144$, our method required only about 0.28 GiB of memory, whereas LAPACK `dsyevx` required about 512 GiB, which exceeded the memory capacity per node of the system that we were using.

Finally, we show the scalability of the parallel algorithms discussed in Section 5.3.3 in computing 100 eigenvalues of a given $n \times n$ matrix where $k_0 = \lfloor n/2 \rfloor - 50$ and $k_1 = k_0 + 99$. The starting interval of $[0, 2048]$ was used for our bisection method. Small values of m were chosen for the master-worker model to allow for load balancing. We used up to 144 MPI processes, where each process was mapped



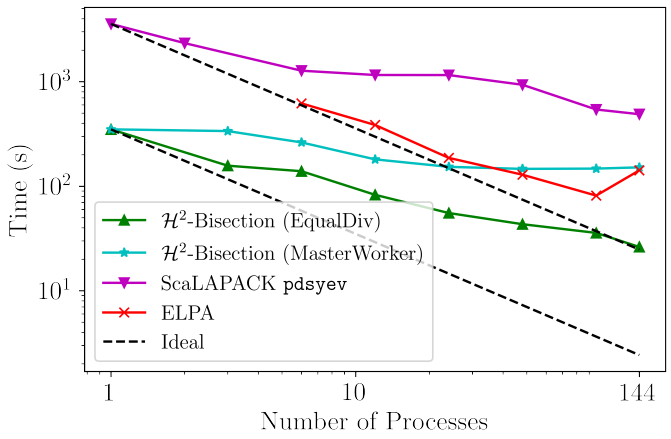
(a)



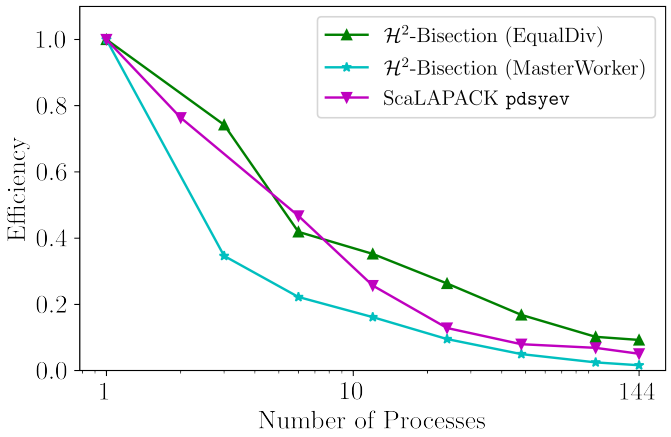
(b)

Figure 5.15: Computing the k-th eigenvalue of synthetic Laplace matrices with $\epsilon_{ev} = 10^{-5}$ using a single CPU core: (a) computation time; (b) memory consumption

to one core. The subroutine `MPI_Dims_create` was used to determine the size of the 2D process grid for ScaLAPACK and ELPA. Fig. 5.16a shows that our first parallel algorithm was faster than both ScaLAPACK and ELPA when using up to 144 processes. On the other hand, our second algorithm with the master-worker model did not scale as much, making it only outperform ScaLAPACK for this particular case. This is because the 100 target eigenvalues were closely clustered within a small interval so that the generation of tasks became a bottleneck and the workers spent most of their time waiting for task. The same reason is also related to the low efficiency shown in Fig. 5.16b. When the target eigenvalues are clustered in an interval that is much smaller than the initial bisection interval, reusing the inertia from the computation of smallest and largest target eigenvalues will dramatically reduce the size of the starting interval for computing the subsequent eigenvalues, which naturally will also reduce their computation times. The smaller the interval containing the target eigenvalues than the initial bisection interval is, the more the speedup factor will decay. One way to address



(a)



(b)

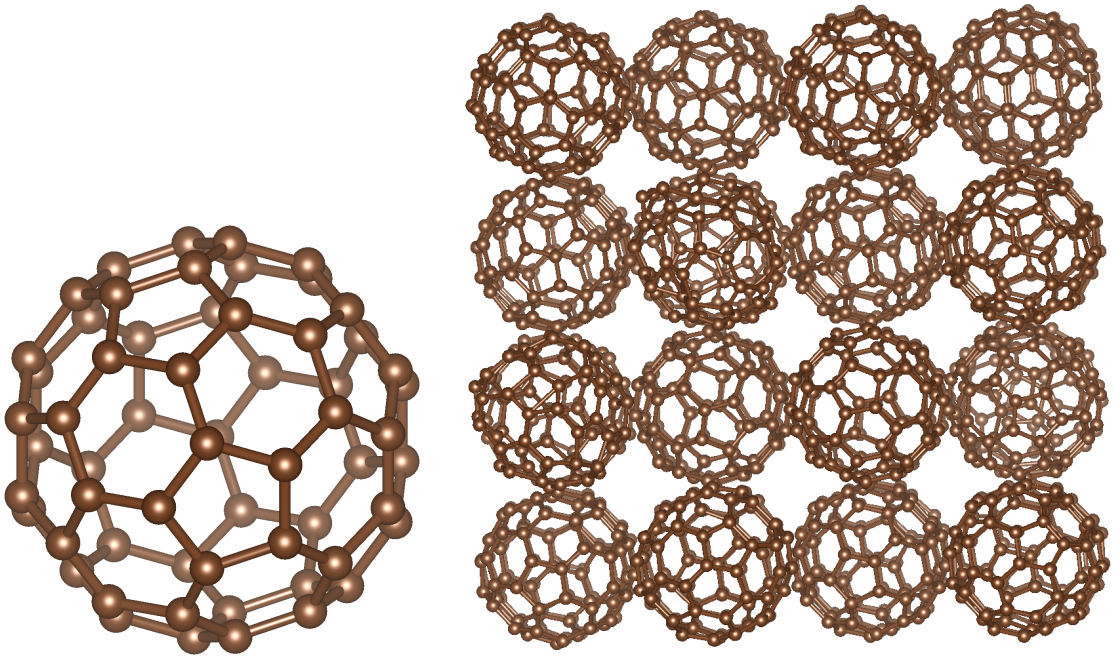
Figure 5.16: Computing 100 interior eigenvalues of order 32,768 synthetic Laplace matrix with $\epsilon_{ev} = 10^{-5}$ using up to 3 compute nodes: (a) computation time; (b) process efficiency

this is by supplying a sufficiently small initial bisection interval that contains all target eigenvalues.

5.3.4.2 Computing the Target Eigenvalue for Electronic Structure Calculations

Here we tested our method to solve the k-th eigenvalue problem arising from electronic structure calculations of carbon nanomaterials composed of *fullerene* (C_{60}) allotropes [69]. A single fullerene takes the shape of a ball made up of 60 carbon atoms, as shown in Figure 5.17. The real symmetric matrices corresponding to the standard eigenvalue problems were generated using ELSES quantum mechanical simulator [53]. The test materials that we used were formed by 64, 128, 256, and 512 fullerenes, which correspond to the matrix sizes of 15360, 30720, 61440, and 122880, respectively. We used the Hilbert space-filling curve to hierarchically partition the mesh where each fullerene ball formed the leaf block of the $\mathcal{H}^2$-matrix.

The target eigenvalue index k is a material-specific value that is determined by the number of

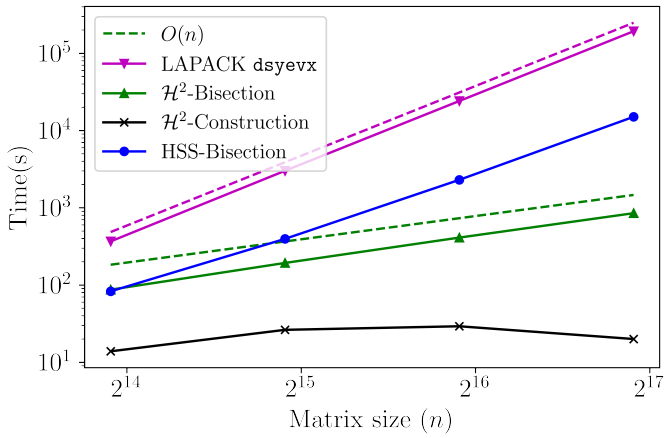


(a) (b)

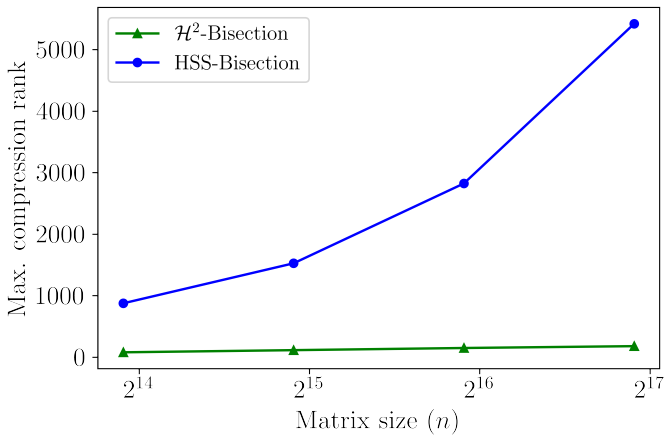
Figure 5.17: (a) A single fullerene mesh; (b) Arrangement of 32 fullerenes (2 identical layers of 4 × 4 fullerenes)

electrons in the material. In a typical case, the index is defined as $k = \lceil n/2 \rceil$ where n is the matrix size. This target eigenvalue is important in determining the electronic properties of the material. For example, the difference between the k-th and $(k+1)$-th eigenvalue, which is referred to as the energy gap, can be used to determine the conductivity of metallic materials [61].

Figure 5.18a shows the time to compute the target eigenvalue for each material with a known starting interval of $[-2, 2]$ and accuracy of $\epsilon_{ev} = 10^{-3}$ which is typically enough to produce an initial estimate for electronic structure calculations. Near linear scaling with respect to the matrix size can be observed from our $\mathcal{H}^2$ bisection method. On the other hand, the HSS bisection showed computation times that grow as $\mathcal{O}(n^3)$. This is because the matrices originate from 3D geometries where HSS could not maintain a constant rank as the problem size is increased, leading to a high compression rank that grew proportionally with the matrix size as shown in Fig 5.18b. Whereas $\mathcal{H}^2$-matrix allows the off-diagonal blocks with high numerical rank to be subdivided further so that each compressed block could maintain a sufficiently small rank.



(a)



(b)

Figure 5.18: Computing the target eigenvalue for electronic state calculations with $\epsilon_{ev} = 10^{-3}$ using a single CPU core: (a) computation time; (b) maximum compression rank

Chapter 6

Conclusion

This thesis presented two important tools to solve the symmetric eigenvalue problems efficiently. The proposed methods utilize the structured low-rank property of dense matrices arising from many applications in order to obtain fast algorithms with controllable accuracy. The performance and accuracy of the proposed methods were evaluated on various kinds of problems ranging from synthetic to practical ones. Numerical results showed significant improvements over the existing methods, including the state-of-the-art dense factorization and eigensolver of LAPACK and ScaLAPACK in terms of both computation time and storage consumption. Furthermore, the proposed methods exhibit promising parallel scalability on shared and distributed memory systems, allowing them to benefit from modern supercomputer architectures. The key findings of this thesis are summarized as follows, along with some remarks on possible future directions.

In Chapter 4 we investigated the QR decomposition of BLR matrices and its application to the QR algorithm, which has been the standard method to compute all eigenvalues in the dense linear algebra community. We presented two new algorithms for the QR decomposition of BLR matrices based on the blocked and tiled Householder methods along with their parallelization on shared-memory systems. One is based on the blocked Householder method with $\mathcal{O}(mn)$ complexity, and the other one is based on the tiled Householder method with $\mathcal{O}(mn^{1.5})$ complexity. Numerical results showed that our proposed algorithms are orders of magnitude faster than the state-of-the-art dense QR decomposition of Intel MKL. Moreover, the proposed method also achieved higher parallel scalability and robustness to ill-conditioning than the existing method in [55], which to the best of our knowledge is the only existing work that studies QR decomposition of BLR matrices. Further, applying them to the QR algorithm produced an eigenvalue solver with controllable accuracy, but its performance suffered from the growing rank of the off-diagonal blocks during the QR iteration. Although this limits the application of our BLR-QR decomposition for the QR iteration, another possible application in the context of eigenvalue solver is in the tridiagonal reduction of BLR matrices. In addition, we would also like to investigate the application of our BLR-QR decomposition to solve other problems such as large-scale least squares problems on distributed memory systems.

In Chapter 5 we investigated the generalized LDL decomposition of HSS and $\mathcal{H}^2$ matrices and their application to the bisection eigenvalue solver. In the first part, we extended the existing method of [88] to distributed memory systems in order to obtain a highly parallel generalized LDL decomposition of HSS matrices with $\mathcal{O}(n)$ complexity. Employing it in the bisection method resulted in a fast and scalable algorithm to compute the k-th eigenvalue of HSS matrices in near-linear time. Numerical results showed a dramatic speedup and better parallel scalability over the vendor-optimized parallel dense eigenvalue solvers in LAPACK and ScaLAPACK, allowing our method to scale up to tens of thousands of CPU cores across hundreds of nodes. This is mainly attributed to the inherent parallelism of the generalized LDL decomposition when combined with the simple structure of the HSS matrices. In the second part, we further extended the generalized LDL decomposition of HSS matrices to $\mathcal{H}^2$ matrices based on the idea in [63]. Although this extension led to a reduced degree of parallelism compared to the HSS counterpart, it allows us to tackle the more general problem classes efficiently, especially the ones originating from 3D objects where the HSS-based method suffers from performance degradation, making it unable to attain the optimal linear complexity that the $\mathcal{H}^2$-based method achieves. This is verified by applying both methods to the bisection eigenvalue solver in order to solve the k-th eigenvalue problems arising from the electronic state calculations of carbon nanomaterials, where the $\mathcal{H}^2$-based bisection method outperformed the existing HSS-based bisection on a single node. Comparison between the two methods on multiple nodes will be addressed as a future work due to the need for distributed memory runtime system to fully exploit the parallelism of the $\mathcal{H}^2$-based method. In addition, the backward error bound for the generalized LDL decomposition of $\mathcal{H}^2$ matrices, which is not available at the moment, should be addressed in order to improve its reliability when applied to the bisection eigenvalue algorithm.

Bibliography

[1] Stars-h, 2020. URL `https://ecrc.github.io/stars-h`.

[2] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, and David Keyes. Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, pages 22–40, Cham, 2017. Springer International Publishing. ISBN 978-3-319-58667-0.

[3] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, Ali Charara, Aniello Esposito, and David Keyes. Exploiting data sparsity for large-scale matrix computations. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 721–734, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96983-1.

[4] Sivaram Ambikasaran and Eric Darve. An $O(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices. *J. Sci. Comput.*, 57(3):477–501, December 2013. ISSN 0885-7474. doi: 10.1007/s10915-013-9714-z.

[5] Patrick. Amestoy, Cleve. Ashcraft, Olivier. Boiteau, Alfredo. Buttari, Jean-Yves. L'Excellent, and Clément. Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015. doi: 10.1137/120903476.

[6] Patrick Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4): A1710–A1740, 2017. doi: 10.1137/16M1077192.

[7] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw.*, 45(1), February 2019. ISSN 0098-3500. doi: 10.1145/3242094. URL `https://doi.org/10.1145/3242094`.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for

Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

[9] M. Ridwan Apriansyah and Rio Yokota. Parallel qr factorization of block low-rank matrices. *ACM Trans. Math. Softw.*, 48(3), sep 2022. ISSN 0098-3500. doi: 10.1145/3538647. URL `https://doi.org/10.1145/3538647`.

[10] M. Ridwan Apriansyah and Rio Yokota. Computing the k-th eigenvalue of symmetric h^2-matrices. In *52nd International Conference on Parallel Processing (ICPP 2023)*, page to appear. ACM, 2023. ISBN 979-8-4007-0843-5/23/08. doi: 10.1145/3605573.3605607. URL `https://doi.org/10.1145/3605573.3605607`.

[11] Cleve Ashcraft, Alfredo Buttari, and Theo Mary. Block low-rank matrices with shared bases: Potential and limitations of the blr^2 format. *SIAM Journal on Matrix Analysis and Applications*, 42(2):990–1010, 2021. doi: 10.1137/20M1386451.

[12] Mario Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2008. ISBN 9783540771470. URL `https://books.google.co.jp/books?id=hnIJOwyz9Z4C`.

[13] Peter Benner and Thomas Mach. On the QR decomposition of H-matrices. *Computing*, 88(3): 111–129, 2010. ISSN 1436-5057. doi: 10.1007/s00607-010-0087-y. URL `https://doi.org/10.1007/s00607-010-0087-y`.

[14] Peter Benner and Thomas Mach. Computing all or some eigenvalues of symmetric $\mathcal{H}\_\ell$-matrices. *SIAM Journal on Scientific Computing*, 34(1):A485–A496, 2012. doi: 10.1137/100815323.

[15] Peter Benner and Thomas Mach. The preconditioned inverse iteration for hierarchical matrices. *Numerical Linear Algebra with Applications*, 20(1):150–166, 2013. doi: 10.1002/nla.1830.

[16] Peter Benner, Steffen Borm, Thomas Mach, and Knut Reimer. Computing the eigenvalues of symmetric h^2-matrices by slicing the spectrum. *Comput. Vis. Sci.*, 16(6):271–282, dec 2013. ISSN 1432-9360. doi: 10.1007/s00791-015-0238-y.

[17] Dario A. Bini, Luca Gemignani, and Victor Y. Pan. Fast and stable QR eigenvalue algorithms for generalized companion matrices and secular equations. *Numerische Mathematik*, 100(3): 373–408, may 2005. ISSN 0945-3245. doi: 10.1007/s00211-005-0595-4.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN 0-89871-397-8 (paperback).

[19] Steffen Borm and Lars Grasedyck. Hybrid cross approximation of integral operators. *Numer. Math.*, 101(2):221–249, aug 2005. ISSN 0029-599X. doi: 10.1007/s00211-005-0618-1. URL https://doi.org/10.1007/s00211-005-0618-1.

[20] Karen Braman, Ralph Byers, and Roy Mathias. The multishift qr algorithm. part ii: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, 23(4):948–973, 2002. doi: 10.1137/S0895479801384585.

[21] Peter Businger and Gene H. Golub. Linear least squares solutions by householder transformations. *Numer. Math.*, 7(3):269–276, June 1965. ISSN 0029-599X. doi: 10.1007/BF01436084. URL https://doi.org/10.1007/BF01436084.

[22] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, jan 2009. ISSN 0167-8191. doi: 10.1016/j.parco.2008.10.002. URL https://doi.org/10.1016/j.parco.2008.10.002.

[23] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405–422, 2003. ISSN 0955-7997. doi: 10.1016/S0955-7997(02)00152-2. Large scale problems using BEM.

[24] Steffen Börm, Nadine Albrecht, Christina Börst, Sven Christophersen, Jonas Lorenzen, Dirk Boysen, Knut Reimer, and Jessica Gördes. H2lib 3.0, 2016. URL http://www.h2lib.org.

[25] Difeng Cai, Hua Huang, Edmond Chow, and Yuanzhe Xi. Data-driven construction of hierarchical matrices with nested bases, 2022. URL https://arxiv.org/abs/2206.01885.

[26] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-scale task-based cholesky factorization toward climate and weather prediction applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379939. doi: 10.1145/3394277.3401846. URL https://doi.org/10.1145/3394277.3401846.

[27] S. Chandrasekaran and M. Gu. A divide-and-conquer algorithm for the eigendecomposition of symmetric block-diagonal plus semiseparable matrices. *Numerische Mathematik*, 96(4):723–731, 2004. ISSN 0945-3245. doi: 10.1007/s00211-002-0199-1.

[28] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A. J. van der Veen, and D. White. Some fast algorithms for sequentially semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 27(2):341–364, 2005. doi: 10.1137/S0895479802405884.

[29] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for hss representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):67–81, 2007. doi: 10.1137/050639028.

[30] Shiv Chandrasekaran, Ming Gu, Jianlin Xia, and Jiang Zhu. A fast QR algorithm for companion matrices. In Joseph A. Ball, Yuli Eidelman, J. William Helton, Vadim Olshevsky, and James Rovnyak, editors, *Recent Advances in Matrix and Operator Theory*, pages 111–143. Birkhäuser Basel, 2008. ISBN 978-3-7643-8539-2.

[31] Ali Charara, David Keyes, and Hatem Ltaief. Tile low-rank gemm using batched operations on gpus. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 811–825, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96983-1.

[32] H. Cheng, Z. Gimbutas, P. G. Martinsson, and V. Rokhlin. On the compression of low rank matrices. *SIAM J. Sci. Comput.*, 26(4):1389–1404, April 2005. ISSN 1064-8275. doi: 10.1137/030602678. URL `https://doi.org/10.1137/030602678`.

[33] Alan K. Cline, Gene H. Golub, and George W. Platzman. Calculation of normal modes of oceans using a lanczos method. In JAMES R. BUNCH and DONALD J. ROSE, editors, *Sparse Matrix Computations*, pages 409–426. Academic Press, 1976. ISBN 978-0-12-141050-6. doi: 10.1016/B978-0-12-141050-6.50029-5.

[34] J. J. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36(2):177–195, jun 1980. ISSN 0029-599X. doi: 10.1007/BF01396757.

[35] Yuli Eidelman, Israel Gohberg, and Vadim Olshevsky. The QR iteration method for hermitian quasiseparable matrices of an arbitrary order. *Linear Algebra and its Applications*, 404:305–324, jul 2005. ISSN 0024-3795. doi: 10.1016/j.laa.2005.02.037.

[36] J. G. F. Francis. The QR Transformation A Unitary Analogue to the LR Transformation—Part 1. *The Computer Journal*, 4(3):265–271, 01 1961. ISSN 0010-4620. doi: 10.1093/comjnl/4.3.265. URL `https://doi.org/10.1093/comjnl/4.3.265`.

[37] J. G. F. Francis. The QR Transformation—Part 2. *The Computer Journal*, 4(4):332–345, 01 1962. ISSN 0010-4620. doi: 10.1093/comjnl/4.4.332. URL `https://doi.org/10.1093/comjnl/4.4.332`.

[38] Alan George and Joseph W.H. Liu. Householder reflections versus givens rotations in sparse orthogonal decomposition. *Linear Algebra and its Applications*, 88-89:223 – 238, 1987. ISSN 0024-3795. doi: 10.1016/0024-3795(87)90111-X.

[39] Wallace Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):26–50, 1958. ISSN 03684245. URL http://www.jstor.org/stable/2098861.

[40] Abeynaya Gnanasekaran and Eric Darve. Hierarchical orthogonal factorization: Sparse least squares problems. *J. Sci. Comput.*, 91(2), may 2022. ISSN 0885-7474. doi: 10.1007/s10915-022-01824-9. URL https://doi.org/10.1007/s10915-022-01824-9.

[41] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2 (2):205–224, 1965. doi: 10.1137/0702016.

[42] Gene H. Golub and Henk A. van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):35–65, 2000. ISSN 0377-0427. doi: 10.1016/S0377-0427(00)00413-1. Numerical Analysis 2000. Vol. III: Linear Algebra.

[43] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. ISBN 0-8018-5414-8.

[44] L Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987. ISSN 0021-9991. doi: 10.1016/0021-9991(87)90140-9.

[45] Ming Gu and Stanley C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 16(1):172–191, 1995. doi: 10.1137/S0895479892241287.

[46] Ming Gu and Stanley C. Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM J. Sci. Comput.*, 17(4):848–869, July 1996. ISSN 1064-8275. doi: 10.1137/0917055. URL https://doi.org/10.1137/0917055.

[47] Brian C. Gunter and Robert A. Van De Geijn. Parallel out-of-core computation and updating of the qr factorization. *ACM Trans. Math. Softw.*, 31(1):60–78, March 2005. ISSN 0098-3500. doi: 10.1145/1055531.1055534. URL http://doi.acm.org/10.1145/1055531.1055534.

[48] W. Hackbusch and S. Borm. Data-sparse approximation by adaptive h2-matrices. *Computing*, 69(1):1–35, sep 2002. ISSN 0010-485X. doi: 10.1007/s00607-002-1450-4.

[49] W. Hackbusch and W. Kress. A projection method for the computation of inner eigenvalues using high degree rational operators. *Computing*, 81(4):259–268, dec 2007. ISSN 0010-485X. doi: 10.1007/s00607-007-0253-z.

[50] Wolfgang Hackbusch. A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices. *Computing*, 62:89–108, apr 1999. doi: 10.1007/s006070050015.

[51] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49 of *Springer Series in Computational Mathematics*. Springer, Berlin, Heidelberg, 12 2015. ISBN 978-3-662-47323-8. doi: 10.1007/978-3-662-47324-5.

[52] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53(2): 217–288, may 2011. ISSN 0036-1445. doi: 10.1137/090771806. URL `https://doi.org/10.1137/090771806`.

[53] T Hoshi, S Yamamoto, T Fujiwara, T Sogabe, and S-L Zhang. An order-n electronic structure theory with generalized eigenvalue equations and its application to a ten-million-atom system. *Journal of Physics: Condensed Matter*, 24(16):165502, mar 2012. doi: 10.1088/0953-8984/24/16/165502.

[54] Akihiro Ida, Hiroshi Nakashima, and Masatoshi Kawai. Parallel hierarchical matrices with block low-rank representation on distributed memory computer systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 232–240, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5372-4. doi: 10.1145/3149457.3149477. URL `http://doi.acm.org/10.1145/3149457.3149477`.

[55] Akihiro Ida, Hiroshi Nakashima, Tasuku Hiraishi, Ichitaro Yamazaki, Rio Yokota, and Takeshi Iwashita. Qr factorization of block low-rank matrices with weak admissibility condition. *Journal of Information Processing*, 27:831–839, 2019. doi: 10.2197/ipsjjip.27.831.

[56] W. Jalby and B. Philippe. Stability analysis and improvement of the block gram-schmidt algorithm. *SIAM J. Sci. Stat. Comput.*, 12(5):1058–1073, September 1991. ISSN 0196-5204.

[57] Claude-Pierre Jeannerod, Théo Mary, Clément Pernet, and Daniel S Roche. Improving the Complexity of Block Low-Rank Factorizations with Fast Matrix Arithmetic. *SIAM Journal on Matrix Analysis and Applications*, 40(4):1478–1496, November 2019. doi: 10.1137/19M1255628. URL `https://hal.inria.fr/hal-02008666`.

[58] Andrew Knyazev, Volker Mehrmann, and Jinchao Xu. Numerical solution of pde eigenvalue problems. *Oberwolfach Reports*, 10:3221–3304, nov 2013. doi: 10.4171/OWR/2013/56.

[59] Daniel Kressner and Ana Susnjara. Fast qr decomposition of hodlr matrices, 2018.

[60] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead lapack working note 178. volume 178, 06 2006. ISBN 978-3-540-75754-2. doi: 10.1007/978-3-540-75755-9_18.

[61] Dongjin Lee, Takeo Hoshi, Tomohiro Sogabe, Yuto Miyatake, and Shao-Liang Zhang. Solution of the k-th eigenvalue problem in large-scale electronic structure calculations. *Journal of Computational Physics*, 371:618–632, 2018. ISSN 0021-9991. doi: 10.1016/j.jcp.2018.06.002.

[62] Miaomiao Ma and Dan Jiao. Accuracy directly controlled fast direct solution of general $\langle^2$-matrices and its application to solving electrodynamic volume integral equations. *IEEE Transactions on Microwave Theory and Techniques*, 66(1):35–48, 2018. doi: 10.1109/TMTT.2017.2734090.

[63] Miaomiao Ma and Dan Jiao. Direct solution of general h2-matrices with controlled accuracy and concurrent change of cluster bases for electromagnetic analysis. *IEEE Transactions on Microwave Theory and Techniques*, 67(6):2114–2127, jun 2019. ISSN 1557-9670. doi: 10.1109/TMTT.2019.2914391.

[64] Qianxiang Ma, Sameer Deshmukh, and Rio Yokota. Scalable linear time dense direct solver for 3-d problems without trailing sub-matrix dependencies. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. ISBN 9784665454445.

[65] Thomas Mach. *Eigenvalue Algorithms for Symmetric Hierarchical Matrices*. PhD thesis, Chemnitz University of Technology, 04 2012.

[66] A Marek, V Blum, R Johanni, V Havu, B Lang, T Auckenthaler, A Heinecke, H-J Bungartz, and H Lederer. The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, may 2014. doi: 10.1088/0953-8984/26/21/213201.

[67] Ronald B. Morgan. Computing interior eigenvalues of large matrices. *Linear Algebra and its Applications*, 154-156:289–309, 1991. ISSN 0024-3795. doi: 10.1016/0024-3795(91)90381-6.

[68] Yuji Nakatsukasa and Nicholas J. Higham. Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the svd. *SIAM Journal on Scientific Computing*, 35(3):A1325–A1349, 2013. doi: 10.1137/120876605. URL `https://doi.org/10.1137/120876605`.

[69] Yasuaki Omata, Yuichiro Yamagami, Kotaro Tadano, Takashi Miyake, and Susumu Saito. Nanotube nanoscience: A molecular-dynamics study. *Physica E: Low-dimensional Systems and Nanostructures*, 29(3):454–468, 2005. ISSN 1386-9477. doi: 10.1016/j.physe.2005.06.009. nanoPHYS'05.

[70] Xiaofeng Ou and Jianlin Xia. Superdc: Superfast divide-and-conquer eigenvalue decomposition with improved stability for rank-structured matrices. *SIAM Journal on Scientific Computing*, 44(5):A3041–A3066, 2022. doi: 10.1137/21M1438633.

[71] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Society for Industrial and Applied Mathematics, 1998. doi: 10.1137/1.9781611971163.

[72] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse supernodal solver using block low-rank compression. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1138–1147, 2017. doi: 10.1109/IPDPSW.2017.86.

[73] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3), jul 2009. ISSN 0098-3500. doi: 10.1145/1527286.1527288. URL `https://doi.org/10.1145/1527286.1527288`.

[74] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw.*, 42(4), jun 2016. ISSN 0098-3500. doi: 10.1145/2930660. URL `https://doi.org/10.1145/2930660`.

[75] Robert Schreiber and Charles VanLoan. A storage-efficient wy representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10, 02 1989. doi: 10.1137/0910005.

[76] M. Sergent, D. Goudin, S. Thibault, and O. Aumage. Controlling the memory subscription of distributed applications with a task-based runtime system. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 318–327, 2016.

[77] Daniil V. Shantsev, Piyoosh Jaysaval, Sébastien de la Kethulle de Ryhove, Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Large-scale 3-D EM modelling with a Block Low-Rank multifrontal direct solver. *Geophysical Journal International*, 209(3):1558–1571, 03 2017. ISSN 0956-540X. doi: 10.1093/gji/ggx106. URL `https://doi.org/10.1093/gji/ggx106`.

[78] G. W. Stewart. The economical storage of plane rotations. *Numer. Math.*, 25(2):137–138, June 1976. ISSN 0029-599X. doi: 10.1007/BF01462266. URL `https://doi.org/10.1007/BF01462266`.

[79] G. W. Stewart. *Matrix Algorithms: Volume 1, Basic Decompositions*. Society for Industrial Mathematics, 1998. ISBN 0898714141.

[80] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA, fourth edition, 2009. ISBN 9780980232714 0980232716 9780980232721 0980232724 9788175968110 8175968117.

[81] Ana Susnjara and Daniel Kressner. A fast spectral divide-and-conquer method for banded matrices. *Numerical Linear Algebra with Applications*, 28(4):e2365, 2021. doi: https://doi.org/10.1002/nla.2365.

[82] Hiroshi Tanaka. Global energetics analysis by expansion into three-dimensional normal mode functions during the fgge winter. *Journal of the Meteorological Society of Japan. Ser. II*, 63(2): 180–200, 1985. doi: 10.2151/jmsj1965.63.2_180.

[83] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 1997. ISBN 9780898713619.

[84] Marc Van Barel, Raf Vandebril, Paul Van Dooren, and Katrijn Frederix. Implicit double shift QR-algorithm for companion matrices. *Numerische Mathematik*, 116(2):177–212, aug 2010. ISSN 0945-3245. doi: 10.1007/s00211-010-0302-y.

[85] Raf Vandebril, Marc Van Barel, and Nicola Mastronardi. *Matrix Computations and Semiseparable Matrices: Linear Systems*. Johns Hopkins University Press, Baltimore, 2008. ISBN 978-0-8018-8714-7. doi: 10.1353/book.16537.

[86] James Vogel, Jianlin Xia, Stephen Cauley, and Venkataramanan Balakrishnan. Superfast divide-and-conquer method and perturbation analysis for structured eigenvalue solutions. *SIAM Journal on Scientific Computing*, 38(3):A1358–A1382, 2016. doi: 10.1137/15M1018812. URL `https://doi.org/10.1137/15M1018812`.

[87] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Monographs on numerical analysis. Clarendon Press, 1988. ISBN 9780198534181. URL `https://books.google.co.jp/books?id=5wsK1OP7UFgC`.

[88] Yuanzhe Xi, Jianlin Xia, and Raymond Chan. A fast randomized eigensolver with structured ldl factorization update. *SIAM Journal on Matrix Analysis and Applications*, 35(3):974–996, 2014. doi: 10.1137/130914966.

[89] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010. doi: 10.1002/nla.691.

[90] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126921. URL https://doi.org/10.1145/3126908.3126921.

[91] Chenhan D. Yu, Severin Reiz, and George Biros. Distributed o(n) linear solver for dense symmetric hierarchical semi-separable matrices. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 1–8, 2019. doi: 10.1109/MCSoC.2019.00008.