Parallel Schur Complement Algorithms for the Solution of Sparse Linear Systems and Eigenvalue Problems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

TIANSHI XU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

YOUSEF SAAD

July, 2023

Acknowledgements

I would like to express my most profound appreciation to my advisor, Professor Yousef Saad, for his guidance, support, and encouragement throughout my research. His expertise, insight, understanding, and patience considerably influenced my Ph.D. journey. I consider myself lucky to have him as my advisor.

I would like to thank the other committee members, Professor Daniel Boley, Professor Ju Sun, and Professor John Sartori. Their willingness to dedicate their time and expertise to serve on my committee has been a significant factor in my academic journey. I extend my sincere gratitude to Professor Mohamed Mokbel for his kindness in serving on my committee for the preliminary exam and the time he devoted while on it.

I also wish to thank my coworkers. First, I would like to thank my lab mates at the scientific computing lab: Vasileios Kalantzis, Yuanzhe Xi, Shashanka Ubaru, Geoffrey Dillon, Mohamed El-Guide, Ziyuan Tang, Zechen Zhang, and Camden Sikes. I am particularly indebted to Vasileios Kalantzis, Yuanzhe Xi, Geoffrey Dillon, and Ziyuan Tang. Their partnership has truly made a difference in this process, and for that, I am profoundly grateful. Second, I would like to thank Daniel Osei-Kuffuor and Ruipeng Li at the Lawrance Livermore National Laboratory, who mentored me during my internships and provided significant help in my academic journey. Third, to Anthony Austin. Our collaborative efforts have significantly elevated the caliber of this research and provided me with invaluable learning experiences and insights throughout the process. And finally, to other collaborators: Shifan Zhao, Hua Huang, Huan He, Difeng Cai, and Lucas Erlandson.

I am grateful to the University of Minnesota. The opportunities, resources, and academic environment the institution provides have been instrumental in shaping this research. I am particularly thankful for the Department of Computer Science and Engineering, the Minnesota Supercomputing Institute, Information Technology, International Student and Scholar Services, and Boynton Health, which have greatly facilitated my research. It has been an honor to pursue my studies and conduct my research under the auspices of this esteemed institution.

I would also like part of the credits to go to my friends back home: Ye Tao, Jingxin Shi, Wenkai Li, and Sitao Wang, for their impact on my life.

Finally, I want to express my sincere gratitude to my family for their unflagging love and support throughout my life; this accomplishment would not have been possible without them, especially my wife, Sihong Zhang, for inspiring me to pursue my dreams.

Dedication

This work is dedicated to my family for their unwavering support and belief in my abilities. In loving memory of Xiudao Xu and Yihui Liu, whose love and support will forever inspire me.

Abstract

Large sparse matrices arise in many applications in science and engineering, where the solution of a linear system or an eigenvalue problem is needed. While direct methods are still preferable when solving linear systems arising from two-dimensional models, iterative methods are widely used when solving linear systems arising from three-dimensional models due to their superiority in memory efficiency and computational efficiency. Meanwhile, iterative methods are also widely used for solving eigenvalue problems since no direct methods are available in general. Many efforts have been spent in developing iterative methods either for general problems or for specific applications. Among those methods, the Krylov subspace method is one of the most successful types. For finding the approximation solution of linear systems, incomplete LU (ILU) factorization preconditioned Krylov subspace methods are one of the most popular algorithms known for their robustness. On the other hand, the filtering strategies and Krylov subspace methods are one of the most efficient combinations for computing the entire spectrum of matrix pencils. Due to the increasingly larger size of matrix problems and the architecture of modern supercomputers, parallel computing has become an important component of numerical linear algebra. Domain decomposition (DD) methods partition the original problem into an interface problem and multiple decoupled subproblems that only depend on the solution of the interface problem. Those subproblems can be computed in parallel once the interface problem is solved. In the matrix representation of the problem, the interface problem is usually related to the Schur complement. Both ILU factorization and eigenvalue problems can be accelerated using the DD-based Schur complement approaches.

This dissertation focuses on the parallel DD-based Schur complement approach for the solution of large sparse linear systems and eigenvalue problems on distributed memory systems, especially those equipped with GPUs. The unique contributions of this dissertation include a two-level Galerkin Schur complement preconditioner, a Schur complement low-rank preconditioner, and a polynomial-based Schur complement low-rank preconditioner. We also introduce an efficient parallel algorithm for computing several extreme eigenvalues and a parallel block Givens QR decomposition algorithm.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation discusses two problems. The first problem is finding the solution to the linear system

$$A\mathbf{x} = \mathbf{b}, \tag{1.1}$$

where the $A \in \mathbb{C}^{n\times n}$ is the coefficient matrix, $\mathbf{b} \in \mathbb{C}^n$ is the right-hand side vector, and $\mathbf{x} \in \mathbb{C}^n$ is the solution vector. This dissertation focuses on the situation where A is large and sparse and discusses the so-called general-purpose solvers where the only inputs are A and b. This problem arises in many fields of science and engineering, especially those that require the solution of partial differential equations (PDEs). Those problems can first be discretized via finite difference methods, finite element methods, or finite volume methods. Typically most unknowns are only coupled with very few unknowns that are physically close by in the original domain, and the demand for high accuracy requires a fine discretization that leads to a large number of unknowns. Thus, the resulting coefficient matrices are both large and sparse. Sparse matrices can also come from problems that do not involve PDEs, including problems in circuit simulation, power systems, and some optimization problems. It is also worth mentioning that sparse matrices can be used to approximate dense problems. For example, in data science, the covariance matrices and kernel matrices can have many entries close to zero, making them also suitable for sparse algorithms [1].

Direct methods [2] and iterative methods [3] are two major types of methods for solving Equation 1.1. Direct methods typically compute a form of LU factorization of the coefficient matrix A and get the solution by performing two triangular solves. For symmetric positive definite (SPD) matrices and symmetric indefinite matrices, the Cholesky factorization and LDL factorization are typically computed, respectively. A reordering step is typically applied to reduce the number of nonzero entries in the factors and increase the concurrency during the factorization. Topological sort is commonly used to parallelize the triangular solve steps. Given A and b, the direct solvers can get a solution with high accuracy within a fixed finite step of computations. Iterative methods, on the other hand, start from an initial approximate solution and generate a sequence of approximate solutions using some or all of the previous solutions. If the method is convergent, the sequence of approximation solution should converge to the exact solution. However, the sequence might never reach the exact solution within a finite step, even if there are no rounding errors.

The direct solvers are almost unbeatable for problems arising from two-dimensional models. It has been shown that when the system is defined on a close-planar graph, the serial computational complexity and memory complexity using the nested dissection algorithm are $O(n^{3/2})$ and $O(n \log n)$ respectively [4], while the computation could be further accelerated with parallel computing strategies. In fact, with one of the state-of-the-art sparse direct solvers built in MATLAB, one can solve pretty large two-dimensional problems in seconds, even using laptops. However, the high cost of using direct solvers on three-dimensional problems makes iterative solvers almost mandatory for these problems. The memory complexity of iterative methods can be as low as $O(n)$, while the computational cost is also generally much lower. For example, when using the basic iterative methods, including the Jacobi iteration, the Gauss-Seidel iteration, and the successive overrelaxation (SOR), no extra memory is required except for two vectors of length n holding the approximation solution at the current step and the previous step.

Algebraic multigrid (AMG) is a class of iterative solvers widely used nowadays. The intuition behind AMG is that the smooth component of errors belonging to the near null space of A is difficult to be eliminated using iterative solvers. AMG addresses this issue by restriction and interpolation to eliminate the smooth error components on smaller problems. The optimal convergence of AMG for PDEs with elliptic properties makes it preferable in many applications.

Krylov subspace methods are another important class of iterative solvers that are projection methods utilizing the Krylov subspace. Solvers in this class include the conjugate gradient (CG) method for SPD coefficient matrices and the generalized minimal residual (GMRES) method for general coefficient matrices. The fast convergence of Krylov subspace methods is achieved when the eigenvalues of A are well clusters, and thus the Krylov subspace methods alone are not enough in practice unless the original problem is simple. The idea behind preconditioning strategies is to build a new problem with the same solution while the eigenvalues of the coefficient matrix are better clustered. For instance, a left preconditioner M transforms the original Equation 1.1 into a preconditioned system

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}. \tag{1.2}$$

If setting up and applying M^{-1} is inexpensively, and M is a good approximation of A, the Krylov subspace methods are much more efficient on this new system. In general, the explicit formula of M and M^{-1} is not necessary. The only required operator related to M is the application of M^{-1} on a vector. Some unique versions of the Krylov subspace methods, like the Flexible GMRES (FGMRES), should be used if M^{-1} is subject to variations, i.e., not fixed each time it is applied.

The second problem addressed in this dissertation is the eigenvalue problem

$$A\mathbf{u} = \lambda M\mathbf{u}. \tag{1.3}$$

where the $A, M \in \mathbb{C}^{n\times n}$. This dissertation focuses on the situation where A is real symmetric, M is real symmetric positive definite, and only a handful of the algebraically smallest eigenvalues and their associated eigenvectors within a given interval $[\alpha, \beta]$. This problem arises in applications like spectral clustering and low-frequency response analysis and can be used to construct low-rank approximations.

Iterative methods like Krylov subspace methods or subspace iteration, together with projection methods like Rayleigh-Ritz projection, are the most widely used methods for computing eigenvalues. Strategies such as shift-and-invert, polynomial filtering, and rational filtering can be used to compute eigenvalues within a specific region of interest. The shift-and-invert methods change the original problem so that the eigenvalues close to the shift become the largest in magnitude and, thus, are easy to compute. The filtering strategy is a more powerful tool that changes a small group of eigenvalues within a target region of the original problem into new values close to a target value and changes the remaining eigenvalues close to zero.

1.1 Summary of Dissertation

This dissertation is organized as follows.

- Chapter 2 of the dissertation gives the background knowledge of iterative methods for sparse linear systems and for eigenvalue problems, the domain decomposition (DD) methods, and some data structures used in this dissertation.

- Chapter 3 presents several Schur complement algorithms for linear systems as well as their implementations that the author contributed to. The first section gives the background knowledge of parallel two-level Schur complement methods for linear systems. The second section shows the benefit of using modified ILU to achieve faster convergence. The work presented in this section is based on the manuscript that is currently under review [5]. The algorithms discussed in this section are implemented in the linear solver package hypre [1] . The author contributed to designing the algorithm and was the main developer of these algorithms in the package.

  The third section presents a parallel multilevel Schur complement method with low-rank correction for linear systems. The work presented in this chapter is based on our recent paper [6]. The algorithms discussed in the chapter are implemented in the package `parGeMSLR` [2] . The author contributed to designing the algorithm and was the main developer of these algorithms in the package.

  The last section presents a parallel multilevel Schur complement method with polynomial low-rank correction for linear systems. The work presented in this section is based on the manuscript that is currently in preparation. The author contributed to designing the algorithm and was the main developer of the package, which will be released in the future.

[1] `https://github.com/hypre-space/hypre`
[2] `https://github.com/Hitenze/pargemslr`

- Chapter 4 presents a Schur complement method for eigenvalue problems and a parallel QR algorithm. The first section discusses a parallel Schur complement method for computing several algebraic smallest eigenvalues of large and sparse generalized eigenvalue problems. The work presented in this section is based on the manuscript that has been accepted [7]. The author contributed to designing the algorithm and was the main developer of the package [3] .

  The second section presents a QR decomposition algorithm based on the block Givens rotation, which can be used in the algorithm discussed in the first section of this chapter. The manuscript for the work described in this section is in preparation.

1.2 Funding and access to the computation resources

The research output presented in this dissertation was supported mainly by the National Science Foundation (NSF) grant DMS-1912048, with certain exceptions.

The research output presented in Section 3.2 Section 3.3 was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Work at LLNL was funded by TotalEnergies SE through the FC-MAELSTROM Project (LLNL-JRNL-813686).

The research output presented in Section 3.4 was supported by NSF grant DMS-1912048. This research was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-JRNL-830724).

The research output presented in Section 4.2 was supported by NSF grant DMS-1912048. The authors acknowledge the Minnesota Supercomputing Institute (MSI) at the University of Minnesota for providing resources that contributed to the research results reported within this section (`http://www.msi.umn.edu`).

[3] `https://github.com/Hitenze/schurcheb`

Chapter 2

Background

2.1 Iterative Methods for Sparse Linear Systems

We begin this chapter by considering the problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where the $A \in \mathbb{C}^{n\times n}$ is the coefficient matrix, $\mathbf{b} \in \mathbb{C}^n$ is the right-hand side vector, and $\mathbf{x} \in \mathbb{C}^n$ is the solution vector.

Basic iterative methods are based on the splitting $A = M - N$ where M is a non-singular matrix. We can then transform the original equation into $M\mathbf{x} = N\mathbf{x} + \mathbf{b}$. Left multiplying both sides of the equation with M^{-1} we have $\mathbf{x} = M^{-1}(M\mathbf{x} + \mathbf{b} - A\mathbf{x}) = \mathbf{x} + M^{-1}(\mathbf{b} - A\mathbf{x})$. Using a fixed point iteration strategy with the initial approximate solution or initial guess $\tilde{\mathbf{x}}_0$, the kth step of the iteration can be written as

$$\tilde{\mathbf{x}}_k = \tilde{\mathbf{x}}_{k-1} + M^{-1}(\mathbf{b} - A\tilde{\mathbf{x}}_{k-1}). \tag{2.2}$$

Here the vector $\mathbf{r}_k \equiv \mathbf{b} - A\tilde{\mathbf{x}}_k$ is typically referred to as the residual vector, which is the backward error, while the forward error is $\hat{\mathbf{e}}_k \equiv \mathbf{x}^\star - \tilde{\mathbf{x}}_k$ where $\mathbf{x}^\star$ is the exact solution. We use the hat symbol over $\mathbf{e}$ to distinguish the forward error vector from the columns of an identity matrix.

Some common basic iterative methods are the Richardson iteration with $M = I/\alpha$ being the scaled identity matrix, the Jacobi method with M being the diagonal of A, the

Gauss-Seidel method with M being the upper triangular part of A, and the Successive Over Relaxation (SOR) which is a variant of the Gauss-Seidel method.

In realistic applications, basic iterative methods are likely to diverge or converge slowly. Besides, for those methods that require parameters like Richardson iteration and SOR, selecting near-optimal parameters can be difficult. As a result, better algorithms are desired.

The basic iterative method with splitting $A = M - N$ can be viewed as the Richardson iteration with $\alpha = 1$ on $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ with new coefficient matrix $M^{-1}A$ and new right-hand side $M^{-1}\mathbf{b}$. The new system $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ is called a preconditioned system, and M can be referred to as a preconditioner. Basic iterative methods with $M \neq I$ can be viewed as preconditioned iterations using the Richardson iteration with $\alpha = 1$. We will discuss preconditioned iteration in Section 2.1.4. For now, we consider the case when $M = I$.

During each step of the Richardson iteration we exact information from A by applying a matrix-vector multiplication with a vector depending on all previous vectors. Applying m iterations can be seen as an operator Φ_m on a subspace defined as

$$\mathcal{I}_{m,\mathbf{z}}(A,\mathbf{r}) \equiv \operatorname{span}\{\mathbf{r}, A\mathbf{z}_1, A\mathbf{z}_2, \ldots, A\mathbf{z}_{m-1}\}, \tag{2.3}$$

where $\mathbf{z}_i \equiv \mathbf{z}_i(\mathbf{r}, A\mathbf{z}_1, A\mathbf{z}_2, \ldots, A\mathbf{z}_{i-1})$ is a function of all previous vectors. Richardson iteration simply uses $\mathbf{z}_i = A^{i-1}\mathbf{r}$. From this perspective, there are two potential improvements over the Richardson iteration.

1. Choosing an update vector that is optimal in some sense once given $\mathcal{I}_k(A,\mathbf{r})$. The projection method, which will be discussed in the next section, can be viewed as a method of selecting the optimal update with some constraints.

2. Choosing better $\mathbf{z}_i$ functions. The desired $\mathbf{z}_i$ should be easy to obtain while providing a near-optimal subspace among all $\mathcal{I}_{m,\mathbf{z}}(A,\mathbf{r})$. In fact, the subspace for Richardson iteration, which is also known as the Krylov subspace, is a near-optimal choice. This will be discussed later in Section 2.1.2.

2.1.1 Projection methods

Projection methods approximate the problem $\mathbf{b} - A\mathbf{x} = \mathbf{0}$ by defining two subspaces $\mathcal{K}$ and $\mathcal{L}$ of $\mathbb{C}^n$ with same dimension, and find an approximate solution $\tilde{\mathbf{x}} \in \mathcal{K}$ such that $\mathbf{b} - A\tilde{\mathbf{x}} \perp \mathcal{L}$.

If we have V a basis of $\mathcal{K}$ and W a basis of $\mathcal{L}$, we can have the following projected problem

$$W^H AV\tilde{\mathbf{y}} = W^H\mathbf{b}, \tag{2.4}$$

where $\tilde{\mathbf{x}} = V\tilde{\mathbf{y}}$.

When starting from a nonzero initial guess $\tilde{\mathbf{x}}_0$, instead of searching $\tilde{\mathbf{x}}$ within $\mathcal{K}$, we can search an approximate solution within the affine space $\tilde{\mathbf{x}}_0 + \mathcal{K}$. Define the residual $\tilde{\mathbf{r}}_0 := \mathbf{b} - A\tilde{\mathbf{x}}_0$, we search for an update $\tilde{\mathbf{e}} \in \mathcal{K}$ such that $\tilde{\mathbf{r}}_0 - A\hat{\mathbf{e}} \perp \mathcal{L}$, and compute the new approximation solution as $\tilde{\mathbf{x}} = \tilde{\mathbf{x}}_0 + \tilde{\mathbf{e}}$. With bases V and W we have $\tilde{\mathbf{x}} = \tilde{\mathbf{x}}_0 + V(W^H AV)^{-1}W^H\tilde{\mathbf{r}}_0$. The entire algorithm is shown in Algorithm 1 [3]. An illustration of the projection is visualized in Figure 2.1.

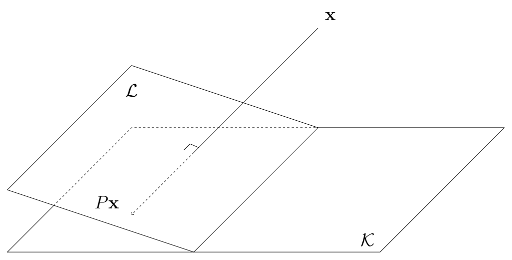


Figure 2.1: A projection operator P that projects onto $\mathcal{K}$, orthogonal to $\mathcal{L}$.

Algorithm 1 Projection method for linear systems

```
1: Choose initial guess $\tilde{\mathbf{x}}$
2: while Not converged do
3:     Select subspaces $\mathcal{K}$ and $\mathcal{L}$
4:     Compute bases $V$ of $\mathcal{K}$ and $W$ of $\mathcal{L}$
5:     Update $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} + V(W^H AV)^{-1}W^H(\mathbf{b} - A\tilde{\mathbf{x}})$
6: end while
```

In our context, we select $\mathcal{K} = \mathcal{I}_{m,\mathbf{z}}(A, \mathbf{r})$. Two of the most commonly used selections of $\mathcal{L}$ given $\mathcal{K}$ are $\mathcal{L} = \mathcal{K}$ and $\mathcal{L} = A\mathcal{K}$. If A is positive definite and $\mathcal{L} = \mathcal{K}$, or A is nonsingular and $\mathcal{L} = A\mathcal{K}$, the matrix $W^H AV$ is guaranteed to be nonsingular. For SPD matrices, the selection $\mathcal{L} = \mathcal{K}$ computes the approximation solution $\tilde{\mathbf{x}} \in \tilde{\mathbf{x}}_0 + \mathcal{K}$ such that the A-norm of the error vector $||\mathbf{x}^\star - \tilde{\mathbf{x}}||_A \equiv \langle A(\mathbf{x}^\star - \tilde{\mathbf{x}}), \mathbf{x}^\star - \tilde{\mathbf{x}} \rangle^{1/2}$ is minimized. On the other hand, the selection $\mathcal{L} = A\mathcal{K}$ computes the approximate solution $\tilde{\mathbf{x}} \in \tilde{\mathbf{x}}_0 + \mathcal{K}$ such that the 2-norm of the residual vector $||\mathbf{b} - A\tilde{\mathbf{x}}||_2$ is minimized [3]. This type of method belongs to the class of minimal residual algorithms. Compared to the results obtained using the Richardson iteration, projection methods using the above choices of $\mathcal{L}$ typically obtain a more accurate approximate solution with the same subspace $\mathcal{K}$.

2.1.2 Krylov subspace methods

In this section, we discuss the selection of the subspace $\mathcal{I}_{m,\mathbf{z}}(A, \mathbf{r})$. One of the simplest subspaces in this form is the one used in the Richardson iteration. Recall that each step of the Richardson iteration computes $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} + \alpha(\mathbf{b} - A\tilde{\mathbf{x}})$. Thus, the residual is updated as $(\mathbf{b} - A\tilde{\mathbf{x}}) \leftarrow (I - \alpha A)(\mathbf{b} - A\tilde{\mathbf{x}})$. Starting from $\mathbf{x}_0$, the approximate solution after applying m steps of the Richardson iteration can be written as

$$\tilde{\mathbf{x}}_m \leftarrow \tilde{\mathbf{x}}_0 + \left[\sum_{i=0}^{m-1} (I - \alpha A)^i \right] (\mathbf{b} - A\tilde{\mathbf{x}}), \tag{2.5}$$

where $\sum_{i=0}^{m-1} (I - \alpha A)^i$ is a polynomial of A with degree up to $m-1$. Thus, we can view the Richardson iteration as a method to search for the approximate solution within the subspace

$$\mathcal{K}_m(A, \mathbf{r}) := \text{span}\{\mathbf{r}, A\mathbf{r}, \ldots, A^{m-1}\mathbf{r}\}. \tag{2.6}$$

The subspace in this form is known as the Krylov subspace. Searching for an approximate solution in Krylov subspace can be seen as building a polynomial approximation of A^{-1}.

Although the Krylov subspace seems very simple, it is very useful in practice [8, 9]. Minimal residual algorithms using Krylov subspace are near optimal for a wide range of classes of matrices over a wide type of algorithms[10, 11]. In [11], Chou studied the general framework of using algorithm operator Φ with information operator $\mathcal{I}_{\mathbf{z}}$ to solve linear systems. Here the information operator is an operator of the dimension m, the matrix A, and the right-hand side $\mathbf{b}$ where $\mathcal{I}_{\mathbf{z}}(m, A, \mathbf{b}) = \mathcal{I}_{m,\mathbf{z}}(A, \mathbf{b})$ is defined in Equation 2.3; and the algorithm operation is an iterative method that uses only the information provided. The author evaluated the total cost of computing the information and applying the algorithm. It was shown in the paper that for many common classes of matrices that are important in applications, the minimal residual algorithms using Krylov subspace have almost optimal costs.

Simplicity and near optimality make projection methods using $\mathcal{K} = \mathcal{K}_m(A, \mathbf{r})$ one of the most common types of iterative methods in practice, known as Krylov subspace methods. The majority of Krylov subspace methods fall into the following three main categories.

1. $\mathcal{L} = A\mathcal{K}_m(A, \tilde{\mathbf{r}}_0)$. With this selection, the resulting approximation solution has minimal residual 2-norm $||\mathbf{b} - A\mathbf{x}||_2$. The minimum residual (MINRES) method and generalized minimal residual (GMRES) method belong to this category.

2. $\mathcal{L} = \mathcal{K}_m(A, \tilde{\mathbf{r}}_0)$. With this selection, the resulting approximation solution has minimal error A-norm $||\mathbf{x}^\star - \tilde{\mathbf{x}}||_A$ when A is SPD where $\mathbf{x}^\star$ is the exact solution. The conjugate gradient (CG) method and full orthogonalization (FOM) method belong to this category.

3. $\mathcal{L} = \mathcal{K}_m(A^H, \tilde{\mathbf{r}}_0)$. With this selection, the resulting methods use biorthogonal sequences instead of orthogonal sequences. The biconjugate gradient (BCG) method belongs to this category.

In practice, an orthogonal basis $V_m = [\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{m-1}]$ of the Krylov subspace is typically computed using the Arnoldi procedure [12] starting by setting $\mathbf{v}_0$ as the normalized initial residual $\mathbf{r}_0 / ||\mathbf{r}_0||_2$ with $\mathbf{r}_0 \equiv \mathbf{b} - A\tilde{\mathbf{x}}_0$. The ith step of the Arnoldi procedure computes matrix-vector multiplication $A\mathbf{v}_{i-1}$ with the current Arnoldi vector, orthonormalizes the result against all previous Arnoldi vectors, and then sets the result as $\mathbf{v}_i$. If A is Hermitian, theoretically, we only need to normalize against the previous two Arnoldi vectors, which are also called Lanczos vectors in this case. This simplified algorithm for real symmetric or Hermitian matrices is known as the Lanczos algorithm [13]. However, full orthogonalization against all previous vectors is sometimes required in practice due to the round-off errors in numerical computation.

With the Arnoldi procedure, the Krylov subspace methods can choose m adaptively. Given m, we do not need to form the entire V_m before applying the projection as shown in Alrogirhm 1. After each step of the Arnodi procedure, the residual norm of the updated approximate solution can be computed inexpensively, so that we can check convergence after each matrix-vector multiplication. Besides, if the condition of the residual norm is satisfied, the approximation solution can be computed using all the currently available information. To avoid the large memory requirement for storing V_m when m is large, many Krylov subspace methods introduce a restart step that clears the entire V_m and restarts the algorithm by using the current approximate solution as the initial guess.

We end this section by summarizing the convergence property of CG and GMRES. For SPD problems, denote by $\kappa_2(A)$ the 2-norm condition number of the coefficient matrix A, a widely used inequality regarding the convergence property of CG is

$$||\mathbf{x}^\star - \mathbf{x}_m||_A \leq 2 \left[\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right]^m ||\mathbf{x}^\star - \mathbf{x}_0||_A \, . \tag{2.7}$$

On the other hand, it is not possible to show general convergence results for GMRES. In fact, Greenbaum et.al show in [14] that for any nonincreasing convergence real positive sequence, there exists a linear system such that applying GMRES on it obtains this sequence as its residual norm after each step. Convergence results similar to the one for CG can be obtained when the coefficient matrix is normal or diagonalizable. Here we only show one of the results when the coefficient matrix is diagonalizable. More results can be seen in [3, § 6.11.4]. For diagonalizable matrix $A = X\Lambda X^{-1}$ where Λ is a diagonal matrix, if all eigenvalues of A are located in the ellipse $E(c, d, a)$ which excludes the origin, the residual norm at the mth steps of GMRES satisfies

$$||\mathbf{b} - A\tilde{\mathbf{x}}_m||_2 \leq \kappa_2(X) \frac{C_m\left(\frac{a}{d}\right)}{\left|C_m\left(\frac{c}{d}\right)\right|} ||\mathbf{b} - A\tilde{\mathbf{x}}_0||_2 \, , \tag{2.8}$$

where C_m is the Chebyshev polynomial of degree m of the first kind. This bound is only useful when A is close to normal so that $\kappa_2(X) \approx 1$.

Although any nonincreasing GMRES residual norm sequence is possible for A with any eigenvalues, in general, the convergence is faster when the 2-norm condition number of A is small or the eigenvalues of A are clustered away from zero.

2.1.3 Algebraic multigrid

Another type of widely used iterative method is the algebraic multigrid (AMG) [15]. The intuition behind AMG is that for basic iterative methods, the error components in the near null space of A typically vanish slowly. For example, when using the Richardson iteration with $\alpha = 1$, the approximate solution is updated with the residual each step, which is an update of nearly $\mathbf{0}$ if the error is in the near null space of A. The AMG community typically refers to such error $\hat{\mathbf{e}} = \mathbf{x}^\star - \tilde{\mathbf{x}}$ as a "smooth error."

If we associate the matrix A with an abstract grid Ω defined by its adjacency graph where the grid points are the vertices and the connections are the edges, the assumption of AMG is that the smooth error is no longer smooth on a coarser grid. Thus, these error components can be removed by a correction step on the coarse grid [16, 17]. Define the original matrix and grid as $A_0 \equiv A$ and $\Omega_0 \equiv \Omega$, the AMG constructs a sequence of progressively smaller grids $\Omega_0 \supset \Omega_1 \supset \Omega_2 \ldots \supset \Omega_l$. A sequence of corresponding restriction operator $I_0^1, I_1^2, \ldots, I_{l-1}^l$ and interpolation operation $I_1^0, I_2^1, \ldots, I_l^{l-1}$ are then defined where typically $I_k^{k+1} = (I_{k+1}^k)^T$. The restriction operator I_k^{k+1} maps vectors on Ω_k to Ω_{k+1}, while the interpolation operator I_{k+1}^k maps vectors on Ω_{k+1} back to Ω_k. The matrices associated with coarser grids are defined recursively as $A_{k+1} = I_k^{k+1} A_k I_{k+1}^k$. A classic V-cycle AMG algorithm is shown in Algorithm 2. After several iterations on the finer grid, the remaining error is likely to be dominated by the smooth components. The residual is then restricted to the coarser grid to accelerate convergence. The result on the coarse grid is then interpolated back to the finer grid, hoping that the smooth component is effectively removed.

Again, it is not easy to show general convergence results for AMG. AMG is provably optimal for some specific types of problems. In [18], Antonio and Marco show that V-cycle multigrid is optimal for matrices belonging to circulate, tau, or Hartley algebras under certain constraints, in the sense that solving a linear system has the same order of cost as applying a matrix-vector multiplication. In practice, AMG works reasonably well for Poisson-like problems on meshes that are close to regular.

Algorithm 2 V-cycle AMG

```
 1: Choose initial guess $\tilde{\mathbf{x}}$
 2: Iterates $\mu_1$ steps to get $\tilde{\mathbf{x}} \approx A_0^{-1}\mathbf{b}$ using $\tilde{\mathbf{x}}$ as initial guess
 3: Set $\mathbf{r}_1 \leftarrow I_0^1(\mathbf{b} - A_0\tilde{\mathbf{x}})$
 4: Set $\tilde{\mathbf{e}}_1 \leftarrow \mathbf{0}$
 5: for $i = 1$ to $l - 1$ do
 6:     Iterates $\mu_1$ steps to get $\tilde{\mathbf{e}}_i \approx A_i^{-1}\mathbf{r}_i$ using $\tilde{\mathbf{e}}_i$ as initial guess
 7:     Compute $\mathbf{r}_{i+1} \leftarrow I_i^{i+1}(\mathbf{r}_i - A_i\tilde{\mathbf{e}}_i)$
 8:     Set $\tilde{\mathbf{e}}_{i+1} \leftarrow \mathbf{0}$
 9: end for
10: Solve $\mathbf{e}_l = A_l^{-1}\mathbf{r}_l$
11: for $i = l - 1$ to 1 do
12:     Correct $\tilde{\mathbf{e}}_i \leftarrow \tilde{\mathbf{e}}_i + I_{i+1}^i\tilde{\mathbf{e}}_{i+1}$
13:     Iterates $\nu_2$ steps to get $\tilde{\mathbf{e}}_i \approx A_i^{-1}\mathbf{r}_i$ using $\tilde{\mathbf{e}}_i$ as initial guess
14: end for
15: Correct $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} + I_1^0\tilde{\mathbf{e}}_1$
16: Iterates $\nu_2$ steps to get $\tilde{\mathbf{x}} \approx A_0^{-1}\mathbf{b}$ using $\tilde{\mathbf{x}}$ as initial guess
```

2.1.4 Preconditioned iteration

Krylov subspace methods converge slowly for many typical applications. Although counterexample exists, in practice, these methods usually suffer from slow convergence when the eigenvalues of the coefficient matrix are not clustered. As is stated in [3]: *"Preconditioning is a key ingredient for the success of Krylov subspace methods in these applications."* In general, preconditioning strategies modify the original linear system and solve a new system that is much easier to solve using Krylov subspace methods.

For Equation 2.1, the left preconditioning uses a preconditioning matrix or preconditioner M and modifies the original system into the left preconditioned system

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}, \tag{2.9}$$

where $M^{-1}A$ is the preconditioned matrix. As we discussed earlier, the basic iterative methods can be seen as a left preconditioned Richardson iteration. An efficient preconditioner should have two characteristics. First, M should be a good approximation of A so that the resulting eigenvalues of $M^{-1}A$ cluster better. Second, the operator M^{-1} can be applied inexpensively. Otherwise, the cost per iteration would be high.

There are two other common preconditioning strategies, including right preconditioning and split preconditioning. The right preconditioned system takes the form $AM^{-1}(M\mathbf{x}) = \mathbf{b}$, while the split preconditioned system takes the form $M_1^{-1}AM_2^{-1}(M_2\mathbf{x}) = M_1^{-1}\mathbf{b}$, with preconditioned matrix AM^{-1} and $M_1^{-1}AM_2^{-1}$, respectively.

We refer to iterative methods with the preconditioned system as preconditioned iteration. It is worth mentioning that neither the preconditioning matrix itself nor the preconditioned matrix is explicitly needed during the preconditioned iteration using Krylov subspace methods. All we need is the operation of applying M^{-1} to a vector. For some Krylov subspace methods, M can even change from iteration to iteration. The flexible GMRES (FGMRES) algorithm falls into this category. Also, although CG requires A to be symmetric when M is SPD, it is still possible to use CG with preconditioning and right preconditioning by slightly modifying the algorithm. In the following sections, we will introduce two classes of preconditioners that are used in this dissertation: the ILU preconditioner for general matrices, and the factorized approximate inverse preconditioner for SPD matrices.

2.1.5 Incomplete LU preconditioner

One of the most widely used types of general-purpose preconditioners are ILU-based preconditioners. For many sparse matrices, only small portions of entries in its LU factorization have a large magnitude. The ILU factorization drops some entries during the Gaussian elimination to obtain $\tilde{L}\tilde{U} \approx A$, which can be done either by pre-selecting a pattern or selecting the pattern during the computation. We store sparse matrices mainly in the Compressed Sparse Row (CSR) format, which will be discussed in detail in 2.3.1. CSR format allows fast access to each row of the matrices, thus the row-wise ILU factorization summarized in Algorithm 3 is of specific interest, where $\tilde{U}$ and the strictly lower triangular part of $\tilde{L}$ are stored in the upper and strictly lower triangular part of A respectively after the factorization, and the diagonal of $\tilde{L}$ are all ones. When applying the ILU preconditioner, triangular solves with $\tilde{L}$ and $\tilde{U}$ are required. When A is Hermitian positive definite, the incomplete Cholesky factorization can be used to build an approximation of A in the form $\tilde{L}\tilde{L}^H \approx A$ or $\tilde{U}^H\tilde{U} \approx A$.

The most widespread pattern-selecting strategy for ILU is based on the level of fill. The heuristic behind this strategy is that if we assume that the off-diagonal nonzero

Algorithm 3 Row-wise static pattern ILU factorization

```
1: Choose a pattern $P$
2: For each $(i,j) \in P$ set $a_{i,j} \leftarrow 0$.
3: for $i = 2$ to $n$ do
4:     for $k = 1$ to $i-1$ and if $(i,k) \notin P$ do
5:         $a_{i,k} \leftarrow a_{i,k}/a_{k,k}$
6:         for $j = k+1$ to $n$ and if $(i,j) \notin P$ do
7:             $a_{i,j} \leftarrow a_{i,j} - a_{i,k}a_{k,j}$
8:         end for
9:     end for
10: end for
```

entries of a diagonally dominant matrix A have the same magnitude $\epsilon < 1$ and if we assign them an initial level of fill $lev = 0$, step 7 in Algorithm 3 can be seen as an update

$$a_{i,j} \approx \epsilon^{lev_{i,j}+1} + \epsilon^{lev_{i,k}+1}\epsilon^{lev_{k,j}+1}, \tag{2.10}$$

where $lev_{i,j}$ is the current level of fill of the (i,j)th entry of A during the factorization. Thus, we can use the level of fill as an estimate of the magnitude of an entry in the LU factorization without actually computing its value. The commonly seen definition of the initial level of fill is

$$lev_{i,j} = \begin{cases} 0 & \text{if } a_{i,j} \neq 0 \text{ or } i = j, \\ \infty & \text{otherwise,} \end{cases} \tag{2.11}$$

and the update of the level of fill is

$$lev_{i,j} = \min(lev_{i,j}, lev_{i,k} + lev_{k,j} + 1), \tag{2.12}$$

during step 7 in Algorithm 3.

The ILU algorithm that keeps entries with the level of fill up to k is known as ILU(k). One of the simplest forms of ILU is the ILU(0) factorization, which is equivalent to choosing P to be the zero pattern of A. The advantage of this choice is that it requires no extra computation to select the pattern. ILU(0) is also referred to as ILU factorization with no fill-in. For $k \geq 1$, a so-called symbolic factorization step is typically applied before the actual factorization, which computes only the level of fill without computing the L and U factors.

Another commonly used algorithm is the threshold-based ILU algorithm, which determines the pattern during the factorization. This algorithm is usually referred to as ILUT, which first computes the value during the update step and ignores an update if the value is less than a given threshold. ILUT typically requires another parameter to control the maximum number of nonzeros in each row of the L and U factors.

The above strategies only consider the magnitude of the entries of L and U, which strongly relate to the approximation error $A - \tilde{L}\tilde{U}$. However, the errors of the inverse of the factors $L^{-1} - \tilde{L}^{-1}$ and $U^{-1} - \tilde{U}^{-1}$ are also important [19, 20], especially when L or U is ill-conditioned. It can be shown that dropping small entries during the factorization may introduce error matrices with arbitrarily large norms. An ILU strategy based on the crout version of the Gaussian elimination introduced in [20] can be used to address this issue by using a greedy algorithm to estimate the influence of dropping an entry to the error norm.

When A is an M-matrix, the ILU factorization exists for any pattern P, and the basic iteration using $M = \tilde{L}\tilde{U}$ will converge to the solution. For general applications, some diagonal entries of A might become very small or even zero during the factorization, which would potentially make the algorithm fail. In practice, strategies including pivoting and diagonal shift are widely used to improve the robustness of the algorithm.

2.1.6 Approximate inverse preconditioners

As mentioned in the previous section, the accuracy of the inverse of the LU factors $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ can be important if L and U are not well-conditioned. Besides, applying ILU preconditioners requires triangular solves, which can potentially influence parallel performance, especially on manycore architectures.

A possible workaround is to build the approximation of A^{-1} instead of A. Preconditioners using the former approach are usually called explicit preconditioners, since applying them does not require an additional inverse step, while preconditioners the latter approach are called implicit preconditioners.

Since A^{-1} is usually not easily available, approximation to A^{-1} can be built by finding a sparse matrix M or pair of sparse matrices L and U with small $||AM - I||$, $||I - MA||$ or $||I - LAU||$. We will only discuss the right preconditioner and the split preconditioner since the left preconditioner is similar to the right preconditioner. Note that the definition of M, L, and U in many papers on approximate inverse and in this section is different from what we defined earlier. When talking about approximate inverse preconditioners, those matrices typically refer to the inverse of the preconditioner.

Due to the higher cost of minimizing 1-norm or 2-norm, most strategies find M by minimizing the F-norm under certain constraints. When building the right preconditioner, one straightforward approach is to directly apply optimization methods on $||AM - I||_F^2$. When using descent-type methods with initial guess $\tilde{M}$, once we select the direction G, the minimal residual approach selects the stepsize α by letting $R-\alpha AG$ orthogonal to AG with respect to the square norm $\langle \cdot, \cdot \rangle$ defined as $\langle X, Y \rangle = \mathrm{Tr}(Y^H X)$ where $R = I - A\tilde{M}$ is the residual. Thus, α can be computed via

$$\alpha = \frac{\langle R, AG \rangle}{\langle AG, AG \rangle} = \frac{\mathrm{Tr}[(AG)^H R]}{\mathrm{Tr}[(AG)^H AG]}. \tag{2.13}$$

The approximate inverse $\tilde{M}$ would typically become dense after a few iterations, and thus proper dropping strategies need to be applied. Applying dropping strategies to M can control the total number of nonzeros in M but can not guarantee the decent property of the algorithm. On the other hand, applying dropping strategies to G can enforce each step of the iteration as a descent step while having less control over the total number of nonzeros in M. Two common choices of G include the residual direction R and the steepest descend direction $-2A^H R$. The steepest descent version of the algorithm is summarized in Algorithm 4.

Algorithm 4 Global steepest descent approximate inverse right preconditioner

```
1: Choose initial guess $\tilde{M}$
2: while Not converged do
3:     Compute decent direction $G = A^H(I - AM)$
4:     Apply dropping strategy to $G$
5:     Compute step size $\alpha = ||G||_F^2 \,/\, ||AG||_F^2$
6:     Update $\tilde{M} \leftarrow \tilde{M} + \alpha G$
7:     Apply dropping strategy to $\tilde{M}$
8: end while
```

The objective function $||AM - I||_F^2$ can also be written in column version as

$$||I - AM||_F^2 = \sum_{j=1}^{n} ||\mathbf{e}_j - A\mathbf{m}_j||_2^2 \,, \tag{2.14}$$

where $\mathbf{e}_j$ is the jth column of the identity matrix and $\mathbf{m}_j$ is the jth column of M. Thus, instead of a global minimization on $||AM - I||_F^2$, we can apply column-wise minimization on each $||A\mathbf{m}_j - \mathbf{e}_j||_2^2$. We can solve n independent linear system $A\mathbf{m}_j = \mathbf{e}_j$ using MR or GMRES. Again, proper dropping strategies need to be applied to the approximate solution during each step to avoid the use of dense vectors. Besides, the basis of the Krylov subspace also needs to be stored in sparse mode to avoid $O(n^2)$ complexity.

The algorithms we discussed above can not easily guarantee that the preconditioner is nonsingular. A simple workaround is to use the split preconditioner. We end this section by showing a column-wise descent-type algorithm for Hermitian positive definite matrices in Algorithm 5, which computes the factorized approximate inverse in the symmetric form $\tilde{U}\tilde{U}^H \approx A^{-1}$.

Algorithm 5 Column-wise Hermitain factorized approximate inverse

```
 1: Choose initial guess $\tilde{U}$
 2: while Not converged do
 3:     for $i = 1$ to $n$ do
 4:         Compute $\mathbf{r}_i := \mathbf{e}_i - A\tilde{\mathbf{u}}_i$
 5:         Set $\mathbf{r}_i(i+1:n) = 0$                          ▷ in MATLAB notation
 6:         Apply dropping strategy to $\mathbf{r}_i$
 7:         Compute $\alpha = \mathbf{c}_i^H\mathbf{r}_i/\mathbf{c}_i^H\mathbf{c}_i$ where $\mathbf{c}_i = A\mathbf{r}_i$
 8:         Update $\tilde{\mathbf{u}}_i \leftarrow \tilde{\mathbf{u}}_i + \alpha\mathbf{r}_i$
 9:         Apply dropping to $\tilde{\mathbf{u}}_i$
10:     end for
11: end while
```

2.2 Numerical Methods for Eigenvalue Problems

Next, we consider the second problem of solving the eigenvalue problem

$$A\mathbf{u} = \lambda M\mathbf{u}. \tag{2.15}$$

where the $A, M \in \mathbb{C}^{n\times n}$. Typically we search for an eigenvector with a unit norm. It is common to refer to the matrix pair for generalized eigenvalue problems as matrix pencil (A, M), and denote by $\Lambda(A, M)$ the set of all eigenvalues of (A, M). In this dissertation, we are particularly interested in the computation of very few eigenvalues either within a specific region or with the largest magnitude.

We start from standard eigenvalue problems with $M = I$. One basic algorithm for computing a single eigenvalue with the largest magnitude of (A, I) is the power method. Starting from an initial vector $\tilde{\mathbf{v}}$, each step of the power method updates the approximate eigenvector as $\tilde{\mathbf{v}} \leftarrow A\tilde{\mathbf{v}}$ and scales the result so that the largest magnitude of all entries in $\tilde{\mathbf{v}}$ is exactly one. The idea behind the power method is very simple. Assume that the matrix A is diagonalizable and the starting vector can be written as $\sum_{j=1}^{n} \alpha_j \mathbf{u}_j$ where $\mathbf{u}_j$ is the jth eigenvector. After m iterations, the resulting vector is a vector on the direction of $\sum_{j=1}^{n} \alpha_j \lambda_j^m \mathbf{u}_j$ where λ_j is the jth eigenvalue with the magnitude of its largest entry equals to one. Assume there is only one eigenvalue λ_i with the largest magnitude, and it is semi-simple. If the initial vector is not orthogonal to the invariant subspace associated with λ_i, the power method will converge since for any $j \neq i$, we have $\lim_{m\to\infty}(|\lambda_j|/|\lambda_i|)^m = 0$.

The deflation strategy can be used together with the power method to compute more than one eigenvalue. After computing an eigenpair λ_1 and $\mathbf{u}_1$, a single vector deflation uses a new matrix $A_1 := A - \gamma \mathbf{u}_1 \mathbf{v}^H$ where $\mathbf{v}$ is a vector such that $\mathbf{v}^H \mathbf{u}_1 = 1$ and γ is the shift. The shifted matrix A_1 will have the same eigenvalues as A except that λ_1 is replaced by $\lambda_1 - \gamma$. Typical choices of $\mathbf{v}$ are $\mathbf{u}_1$ and $\mathbf{w}_1$ where $\mathbf{w}_1$ is the left eigenvector associated with λ_1. The deflation strategy can be easily extended to the situation with multiple vectors. One possible strategy is the Schur Wielandt deflation. Define $\mathbf{q}_1 = \mathbf{u}_1$. When each new eigenvalue λ_i is computed, the corresponding eigenvector $\mathbf{u}_i$ is orthonormalized against all previous Schur vectors to get a new Schur vector $\mathbf{q}_i$, and the matrix is updated as $A_i := A_{i-1} - \gamma_i \mathbf{q}_i \mathbf{v}_i^H$.

The power method computes a single eigenpair at a time. At each step, the power method can be seen as a method that maintains a subspace $\mathcal{K}$ of dimension one and searches for approximate eigenvectors within it. Similar to our previous discussion on linear systems, the eigenvalue problems can be solved more efficiently if we select a good approximate subspace and use the idea of projection.

2.2.1 Projection methods

Similar to projection methods for linear systems, the projection methods for eigenvalue problems define two subspaces $\mathcal{K}$ and $\mathcal{L}$, and search for an approximate eigenpair $\tilde{\lambda}$, $\tilde{\mathbf{u}}$, such that the residual $A\tilde{\mathbf{u}} - \tilde{\lambda}\tilde{\mathbf{u}}$ is perpendicular to $\mathcal{L}$.

In this dissertation, we only consider the orthogonal projection methods with $\mathcal{L} = \mathcal{K}$. If we have an orthogonal basis V of $\mathcal{K}$, each approximation solution can be written as a linear combination of columns of $\mathcal{K}$ as $\tilde{\mathbf{u}} = V\mathbf{y}$. The orthogonal constraint can then be written as $V^H(AV\mathbf{y} - \tilde{\lambda}V\mathbf{y}) = 0$, which leads to a new eigenvalue problem

$$V^H AV\mathbf{y} = \tilde{\lambda}\mathbf{y}. \tag{2.16}$$

One well-known procedure for computing a group of approximation given $\mathcal{K}$ is the Rayleigh-Ritz procedure shown in Algorithm 6. Note that we can also compute approximate Schur vectors by replacing the eigenvectors with Schur vectors in step 2 of the algorithm.

Algorithm 6 Rayleigh-Ritz procedure

```
1: Compute a basis $V$ of $\mathcal{K}$
2: Compute several eigenpairs $\tilde{\lambda}_i$, $\mathbf{y}_i$ of matrix pencil $(V^H AV, I)$
3: Compute approximate eigenvectors $\tilde{\mathbf{u}}_i = V\mathbf{y}_i$
```

Similar to projection methods for linear systems, orthogonal projection methods for eigenvalue problems also demonstrate some optimality properties. For instance, it can be shown that for standard symmetric eigenvalue problems, given a basis V of a rank m subspace $\mathcal{K}$, if we use the Rayleigh-Ritz procedure to find m approximate eigenvalues, the minimal $||AV - VR||_2$ for all $R \in \mathbb{R}^{m\times m}$ is obtained by $R = V^T AV$. On the other hand, let the eigendecomposition of $V^T AV$ be $Y^T\tilde{\Lambda}Y$, the minimal $||AZ - ZD||_2$ for all $Z \in \mathbb{R}^{n\times m}$ and diagonal matrix $D \in \mathbb{R}^{m\times m}$ is obtained by the orthonormal matrix $Z = VY$ and $D = \tilde{\Lambda}$. If $\mathcal{K}$ is invariant under A, then all the approximate eigenpairs are exact.

2.2.2 Subspace iteration

The accuracy of projection methods depends on the selection of subspace $\mathcal{K}$. Projection methods cannot find good approximate eigenvectors if $\mathcal{K}$ is away from all of the eigenvectors associated with the desired eigenvalues. In this dissertation, we focus on approaches extracting information from A only through matrix-vector multiplications.

One of the simplest strategies is random projection methods, which select $\mathcal{K}$ randomly. Iterative methods can be used to improve $\mathcal{K}$. A straightforward approach is subspace iteration [21], which can be seen as a block version of the power method. The idea is to apply A on a basis V of the current subspace $\mathcal{K}$ as $A^l V$, build a new basis of the subspace spanned by the columns of $A^l V$, and repeat by setting V as the new basis. A common strategy is to use the QR factorization of $A^l V$ to find a new basis. The new basis of the subspace can be used together with projection methods to obtain better approximation. A subspace iteration with orthogonal projection is summarized in Algorithm 7.

Algorithm 7 Subspace iteration with projection

```
1: Choose initial vectors $V$ and initial integer $l$
2: while Not Cconverged do
3:     Compute iteration $\tilde{V} \leftarrow A^l V$
4:     Compute the QR factorization $QR = \tilde{V}$
5:     Compute the QR factorization $ZS = Q^H AQ$
6:     Update $V \leftarrow QY$
7:     Update $l$
8: end while
```

It is also natural to generalize the deflation strategy discussed earlier for the power method to use in subspace iteration. One strategy is to update the matrix-vector multiplication using the same strategy. Deflation can also be done by using the locking strategy. Once an eigenvector has converged, we no longer need to apply the matrix-vector multiplication with it. The vector is then swapped with the first unlocked column of V and locked, which reduces the computation cost.

We end this section with a convergence result associated with the above algorithm [22]. If the $m+1$ eigenvalues of A with the largest magnitude in descending order are $\lambda_1, \lambda_2, \dots, \lambda_{m+1}$ and denote by P the spectral projector associated with the first m of them. If the projection of initial vectors $P\mathbf{v}_1, P\mathbf{v}_2, \ldots, P\mathbf{v}_m$ are linearly independent, we can find a vector $\mathbf{s}_i \in \operatorname{span}\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ such that $P\mathbf{s}_i = \mathbf{u}_i$ for any eigenvector $\mathbf{u}_i$ assciated with λ_i for $i = 1, 2, \ldots, m$. Moreover, denote by $\mathcal{K}_k$ the subspace after the kth iteration and denote by P_k the orthogonal projector onto $\mathcal{K}_k$, we have

$$||(I - P_k)\mathbf{u}_i||_2 \leq ||\mathbf{u}_i - \mathbf{s}_i||_2 \left(\left| \frac{\lambda_{m+1}}{\lambda_i} \right| + \epsilon_k \right)^k, \tag{2.17}$$

with $\lim_{k\to\infty} \epsilon_k = 0$.

2.2.3 Krylov subspace methods

Krylov subspace methods are another successful class of methods for solving eigenvalue problems. While subspace iteration can be seen as a block version of the power method, Krylov subspace methods can be viewed as projection methods on the space spanned by the sequence of approximate eigenvectors generated during the power method, which is exactly the Krylov subspace defined in Equation 2.6 with $\mathbf{r} = \mathbf{u}$ is the initial vector. The Krylov subspace is also provably near optimal for eigenvalue problems over a wide range of classes of matrices [11].

The Arnoldi procedure is again the typical choice for building the Krylov subspace. If the algorithm does not break, m steps Arnoldi procedure generates orthnormal matrix $V_{m+1} \in \mathbb{C}^{n\times m+1}$ and $\hat{H}_m \in \mathbb{C}^{m+1\times m}$ such that $AV_m = V_{m+1}\hat{H}_m$ where V_m is the first m columns of V_{m+1} and the first column $\mathbf{v}_1$ of V_{m+1} is the normalized initial vector. Denote by H_m the first m rows of $\hat{H}_m$, we can see that H_m is a Hessenberg matrix, and the only nonzero entry in the last row of $\hat{H}_m$ is its last entry $h_{m_1,m}$. Thus, the approximation can also be written as

$$AV_m = V_m H_m + h_{m+1,m}\mathbf{v}_{m+1}\mathbf{e}_m^H, \tag{2.18}$$

where $\mathbf{e}_m^H$ is the mth column of the identity matrix. Since $V_m^H A V_m = H_m$, when applying orthogonal projection onto $\mathcal{K}_m(A, \mathbf{v}_1)$, the approximate eigenvalues are exactly the eigenvalues of H_m, and the approximate eigenvector associated with λ_i is $\mathbf{u}_i = V_m\mathbf{y}_i$ where $\mathbf{y}_i$ is the eigenvector of H_m associated with λ_i.

We do not need to actually form the Ritz vectors to compute the residual norm.

The residual norm $||(A - \lambda_i I)\mathbf{u}_i||_2$ can be computed using the last entry of $\mathbf{y}_i$ [22]. This can be easily seen by multiplying $\mathbf{y}_i$ on both sides of Equation 2.18.

For Hermitian matrices, the Arnoldi procedure can be simplified into the Lanczos procedure, while in practice, full orthogonalization is again typically required. For non-Hermitian matrices, another major type of Krylov subspace method is the non-Hermitian Lanczos algorithm based on biorthogonal bases.

In practice, Krylov subspace methods for eigenvalue problems again, in general, require a restart step. This is because unless the initial vector is exactly in an invariant subspace with a small dimension, Krylov subspace methods will not converge within very few steps if relatively high accuracy is wanted. A large step of iteration indicates a large amount of memory to store V_{m+1}, which makes the algorithm not practical. For the Arnoldi method and the Lanczos method with full orthogonalization, the computational cost also increases as m increases. Thus, restarting the iteration with a new initial vector obtained using the information from the previous loops is useful in practice.

Common restart options are explicit restart, implicit restart, and thick restart. The explicit restart strategy simply selects the new initial vector as a linear combination of current approximate Ritz vectors. Unlike the explicit restart strategy, the implicit restart strategy and the thick restart strategy restart with an intermediate state V_{k+1} and $H_{k+1,k}$ with multiple vectors. The implicit restart strategy $m - k$ most unwanted eigenvalues and use the implicit shifted QR algorithm to apply filtering on the current results, while the thick restart strategy selects k most wanted eigenvalues and constructs the restart with them.

The deflation strategy can also be applied to Krylov subspace methods by either locking several vectors to the leading part of A, or directly updating the matrix-vector multiplication with A.

When A is diagonalizable, if the initial vector can be written as $\sum_{j=1}^{n} \alpha_j \mathbf{u}_j$ where $\mathbf{u}_j$ is the jth eigenvector, then we have

$$||(I - P_m)\mathbf{u}_i||_2 \leq \xi_i \epsilon_i^{(m)}, \tag{2.19}$$

where $\xi_i = \sum_{k \neq i}^{n} |\alpha_j| \,/\, |\alpha_i|$ and $\epsilon_i^{(m)} = \min_{p \in \mathcal{P}_{m-1}^{\star}} \max_{\lambda \in \Lambda(A,I) - \lambda_i} |p(\lambda)|$. Here $\mathcal{P}_{m-1}^{\star}$ is the space of all polynomials of degree up to $m - 1$ such that $p(\lambda_i) = 1$ [22].

2.2.4 Practical techniques

In this dissertation, we only study the problem of finding a few eigenvalues of interest. In this section, we discuss several strategies commonly used for this purpose.

Shift-and-invert

The first strategy is the shift-and-invert approach. This approach can be used together with the power method to compute eigenvectors closest to any desired points that are not in $\Lambda(A, I)$. If $A\mathbf{u} = \lambda\mathbf{u}$, we can write $(A - \sigma I)\mathbf{u} = (\lambda - \sigma)\mathbf{u}$ for any $\sigma \neq \Lambda(A, I)$. Since $A - \sigma I$ is nonsingular, we have $(A - \sigma I)^{-1}\mathbf{u} = (\lambda - \sigma)^{-1}\mathbf{u}$. Thus, the eigenvector with eigenvalue λ of the original problem is still an eigenvector but with a different eigenvalue $(\lambda - \sigma)^{-1}$ for the transformed eigenvalue problem $(A - \sigma I)^{-1}\mathbf{u} = \nu\mathbf{u}$. If there is only one unique eigenvalue closest to σ, the power method on the transformed problem will converge to it. According to the convergence theories discussed earlier, shift-and-invert can also be used together with subspace iteration or Krylov subspace methods to search for desired eigenpairs.

Filtering strategies

When there are a large number of eigenvalues of interest, the shift-and-invert strategy becomes less effective. As we compute more eigenpairs, the cost of orthogonalization becomes more and more expensive and ends up being unacceptable.

Filtering strategies solve the eigenvalue problem with several matrix functions $f_i(A)$. The functions $f_i(x)$ are selected such that the function value of $f(x)$ is close to a target value for a small number of eigenvalues, and close to zero for other eigenvalues. Typically $f_i(x)$ is chosen to be a polynomial or a rational function. The eigenvalues of A are estimated first, and those functions are selected such that each $f_i(A)$ captures a similar amount of eigenvalues.

2.2.5 Generalized eigenvalue problems

So far, we have discussed algorithms for standard eigenvalue problems when $M = I$. These approaches can be easily extended to handle general eigenvalue problems. One simple strategy is to solve the problem with matrix pencil $(A^{-1}M, I)$, or $(M^{-1}A, I)$

when M is invertible. If A and M are Hermitian matrices, we can also use the Cholesky factorization to guarantee that the problem is still symmetric. For instance, we can solve the problem with matrix pencil $(L^{-1}ML^{H}-1, I)$ with the factorization $A = LL^{H}$.

2.3 Domain Decomposition

One of the key techniques used frequently in this dissertation is the domain decomposition (DD) technique. DD methods partition the original computational domain into several small subdomains. For example, if we hope to find the numerical solution of a PDE on a two-dimensional square with the help of a 6×6 grid as shown in the left panel of Figure 2.2 using the Finite Element Method (FEM) with the rectangular elements, we can partition the elements into four groups as shown in the right panel of Figure 2.2. The original problem can then be solved within those four subdomains almost independently, except for those constraints added to the grid points shared by multiple subdomains marked black in the figure.

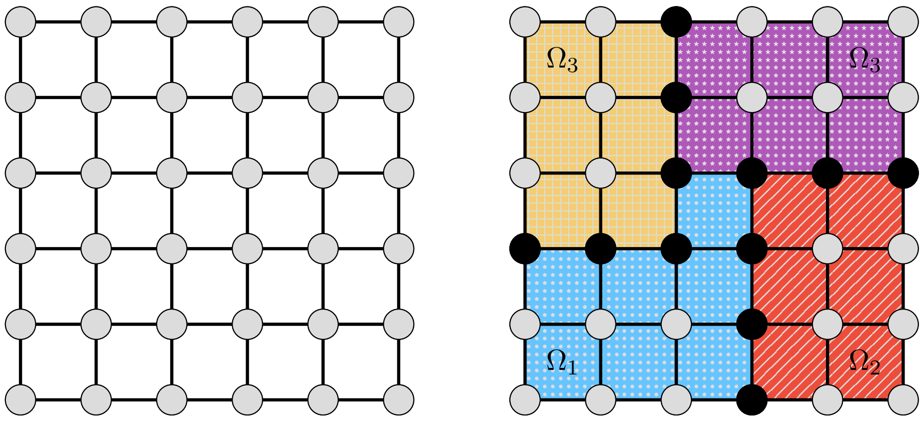


Figure 2.2: A 6×6 grid on a two-dimensional square domain (left) and the element-based partition of it into four subdomains (right). The elements with the same color belong to the same subdomain, and the black grid points are shared by multiple elements.

This dissertation focuses on two additional types of DD methods: the edge-based partition and the vertex-based partition. We specifically concentrate on algebraic DD methods, which borrow a concept similar to that applied in AMG. Instead of partitioning the physical domain of a problem, which might not always be available, we

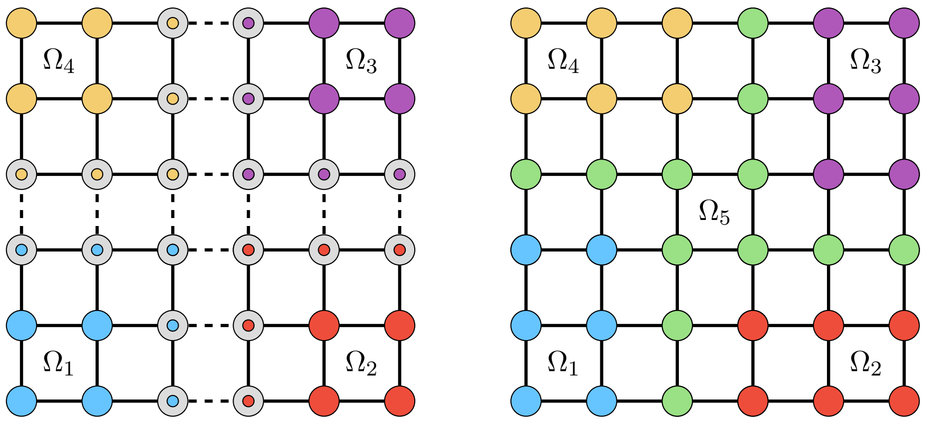


Figure 2.3: A 4-way partitioning of a 6×6 discretized domain obtained from an edge separator (left) and vertex separator (right). In the left figure, the four colors distinguish the four different subdomains. Solid-colored nodes correspond to interior variables. Nodes with a gray background correspond to interface variables. Solid lines correspond to edges between vertices of the same partition. Dashed lines correspond to edges between vertices of neighboring partitions. In the right figure, the four colors other than green distinguish the four different subdomains, and green nodes correspond to the vertex separator.

partition the adjacency graph corresponding to a given matrix A when solving linear systems and corresponding to $|A|+|M|$ when solving eigenvalue problems. More specifically, we refer to the adjacency graph of a square matrix A as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{I})$ associated with it. Here, $\mathcal{V} = 1, \ldots, n$, where n is the matrix's size, and $\mathcal{I} = (i, j) \mid i \neq j,\ A_{i,j} \neq 0$ or $A_{j,i} \neq 0$. We can perceive a linear system $A\mathbf{x} = \mathbf{b}$ and eigenvalue problem $A\mathbf{u} = \lambda M\mathbf{u}$ as a set of n equations. In the context of this adjacency graph, each vertex in the vertex set $\mathcal{V}$ corresponds to one unknown. If there is an edge (i, j) in the edge set, it implies that either the ith unknown appears in the jth equation or the jth unknown appears in the ith equation.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{I})$ be a simple undirected graph where $\mathcal{V}$ is the vertex set, and $\mathcal{I}$ is the edge set. A vertex-based partition partitions the vertex set V into several non-overlapping subsets $\mathcal{V}_i$ such that $\cup_i \mathcal{V}_i = \mathcal{V}$ as shown in the left panel of Figure 2.3. Following this partition, certain edges from the set $\mathcal{I}$ connect vertices that belong to different subsets. These edges form what is called the edge separator, represented by the dashed lines in the example. Based on this separator, a vertex can be classified as either an interior vertex if it does not serve as an endpoint for any edges in the edge separator or an interface vertex if it does.

Another type of partition is the edge-based partition. Unlike the vertex-based partition, an edge-based partition divides the edge set $\mathcal{I}$ into several non-overlapping subsets $\mathcal{I}_i$, such that $\cup_i \mathcal{I}_i = \mathcal{I}$, as shown in the right panel of Figure 2.3. The vertex separator is defined as the set of vertices shared by edges from different subsets. A vertex can again be classified as either an interior vertex if it does not in the vertex separator or an interface vertex if it does. We will discuss the difference between the edge-based partition and the vertex-based partition later in the next chapter.

2.3.1 Distributed Sparse Matrices

In the last section of the chapter, we briefly discuss the association between DD and distributed matrices and introduce several data structures used in the dissertation for storing (distributed) sparse matrices.

This dissertation focus on distributed-memory parallel algorithms. Thus, we need to store vectors and matrices in distributed format. We mainly consider the case where matrices are stored using row-distributed storage. If we have p MPI processes, a matrix A and a vector $\mathbf{x}$ is partitioned into p blocks rows A_1 to A_p and $\mathbf{x}_1$ to $\mathbf{x}_p$, and each block row is further partitioned into p block columns as shown in the following equation:

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}, \mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix}. \tag{2.20}$$

The block A_i is stored on the ith MPI process. Typically the matrix is stored together with another array storing the row numbers of the first and the last row of A_i in A.

Under the DD framework, the domain is partitioned into k subdomains where k is an integer multiple of p. These subdomains are then assigned to different MPI processes evenly. Since each row of the matrix corresponds to a vertex in the graph, when the vertex-based partition is used, rows corresponding to a subdomain assigned to the ith MPI process are assigned to that MPI process to form A_i. On the other hand, when the edge-based partition is used, rows corresponding to interior vertices are assigned to the corresponding MPI process. The equations corresponding to the vertices in the vertex separator are also assigned to different MPI processes to form A_i. Each block A_i is further partitioned into p block columns similarly based on the column. Corresponding to this partition, we can write the original matrix in the following form:

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,p} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p,1} & A_{p,2} & \cdots & A_{p,p} \end{pmatrix}. \tag{2.21}$$

A common practice, which is also used in this dissertation, is to store the diagonal block $A_{i,i}$ and the remaining blocks separately in two different matrices. In this dissertation, we refer to the diagonal block of A_i as A_i^{diag}, and the matrix obtained by replacing the diagonal block of A_i with a zero matrix as A_i^{offd}.

We mainly use two formats for storing local sparse matrices. The simplest data structure among them is the coordinate (COO) format. A sparse matrix is stored using three arrays. Those three arrays have lengths equal to the number of nonzeros of the sparse matrix. After giving the nonzero elements of the sparse matrix an arbitrary order, the ith elements of these three arrays store the row index, the column index, and the value of the ith nonzero elements of the sparse matrix. We typically call those arrays IA, JA, and AA, respectively. The indices in IA and JA are shifted by one so that they start from zero, conforming to a convention known as 0-based indexing. As an example, to store the sparse matrix

$$\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \tag{2.22}$$

in COO format, we can set IA, JA, and AA as

$$\begin{array}{rlllllll} \text{IA:} & 0, & 0, & 1, & 1, & 1, & 2, & 2; \\ \text{JA:} & 0, & 1, & 0, & 1, & 2, & 1, & 2; \\ \text{AA:} & 4, & -1, & -1, & 4, & -1, & -1, & 4. \end{array}$$

The COO format and its variants are widely used. For example, the matrix market format [23] and the sparse matrix in the software `MATLAB` [24] are both based on the COO format. It is relatively easier to add entries to it, so we store many of the intermediate results using the COO format.

If the nonzero elements are ordered so that the array IA holding the row indices is in ascending order, we can further compress this array to obtain the CSR format. For an $m \times n$ matrix, we can use an array of length $m + 1$ to hold the row indices information. Instead of storing the row index explicitly, IA now stores the pointers. That is, the column indices and values of the entries within the ith row of the matrix are stored in JA and AA within the interval specified by the ith entry and the $i + 1$th entry of IA. If we compress JA instead, we will obtain the Compressed Sparse Column (CSC) format. The above sparse matrix can be written in CSR format as

$$\begin{array}{rlllllll} \text{IA:} & 0, & 2, & 5, & 7; & & & \\ \text{JA:} & 0, & 1, & 0, & 1, & 2, & 1, & 2; \\ \text{AA:} & 4, & -1, & -1, & 4, & -1, & -1, & 4. \end{array}$$

Compared to the COO format, the CSR format provides more compact storage and is more efficient for operations such as matrix-vector multiplication and triangular solve. In this dissertation, most of the sparse matrices are stored using the CSR format.

Regarding the global sparse matrices, the local block A_i is stored as is in either COO or CSR format. However, the off-diagonal block is stored in a compressed format. As mentioned earlier, modern applications sometimes require the solution of large-scale problems. As the size of the matrix may necessitate the use of long integers, a common practice is to map the columns in A_i^{offd} to contiguous integers starting from zero and create a mapping array that maps these new column numbers back to their original indices. This means that if there are l_i nonzero columns in A_i^{offd}, we renumber its columns to $0, 2, \ldots, l_i - 1$, and the mapping from new column indices to old column indices is stored in a long integer array of length l_i. In many applications, l_i is not significantly large, which allows memory to be saved, as there is no need to designate JA as a long integer array. We again show a simple example of storing the following matrix with two MPI processes:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \left(\begin{array}{cc|cc} 4 & -1 & & \\ -1 & 4 & -1 & \\ \hline & -1 & 4 & -1 \\ & & -1 & 4 \end{array}\right). \tag{2.23}$$

A_1^{offd} in this case is $[0, -1]^T$ and A_2^{offd} is $[-1, 0]^T$. The matrix A_1^{offd} is stored together with a mapping array with a single element 2, since the only nonzero column of A_1^{offd} is the third column. Similarly, the matrix A_2^{offd} is stored together with the mapping array with a single element 1.

In this chapter, we have introduced the fundamental concepts, terminologies, algorithms, and data structures associated with the main topic of this dissertation. In the subsequent two chapters, we will discuss parallel algorithms for linear systems and eigenvalue problems.

Chapter 3

Parallel ILU Preconditioner for Linear Systems

3.1 Introduction

The third chapter of this dissertation considers the problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}, \tag{3.1}$$

using preconditioned Krylov subspace methods, where $A \in \mathbb{R}^{n\times n}$ is large and sparse.

As discussed in the previous chapter, a good preconditioner M should lead to better clustering of the eigenvalues in the preconditioned matrix $M^{-1}A$ compared to the original matrix A. In addition, M^{-1} should be able to be applied efficiently, and the required information for applying the M^{-1} can be computed inexpensively. Using such a preconditioner can significantly speed up the solution of a problem compared to directly applying Krylov subspace methods to the original system. This dissertation focuses on ILU preconditioning strategies discussed in Section 2.1.5, as they have been demonstrated to be more reliable in solving indefinite and ill-conditioned problems compared with AMG [25, 26, 27, 6, 28]. Some variants of the classical ILU preconditioners, including modified ILU and shifted ILU, have exhibited increased reliability when solving challenging problems[29, 30, 31, 32, 33, 34].

While the classical ILU preconditioners offer various advantages, their sequential nature presents a significant limitation. This limitation makes it difficult to use them on distributed-memory systems or modern many-core processors [35, 28], and thus reduces their capability to solve large-scale applications on modern supercomputers. This dissertation focuses on parallel algorithms for distributed-memory systems. Many new ILU strategies have been developed to improve the parallel performance of the classical ILU preconditioners. Among these strategies, DD-based strategies have proven to be the most successful. As discussed in Section 2.3, the original problem is partitioned into several subdomains when using DD-based strategies. Each subdomain corresponds to several rows of the original matrix A. One of the simplest DD-based approaches is the block Jacobi approach, where the inter-domain couplings are ignored by setting all off-diagonal blocks to zero during the ILU factorization. That is, each MPI process builds the preconditioner using only the local diagonal matrix A_i^{diag} and ignores A_i^{offd}. While constructing and applying the block Jacobi ILU preconditioner can be efficiently done in parallel, its convergence performance tends to degrade as the number of subdomains increases. Methods that utilize information from inter-domain couplings can improve the convergence performance of the preconditioner. When A is obtained using FEM and the elements are known, BDDC [36], FETI-DP [37, 38], as well as the GenEO preconditioner [39] have been found to be effective. Under the situation where only the matrix A is accessible, a simple strategy is to expand the local diagonal blocks with information from the inter-domain couplings but only update local unknowns when applying the preconditioner. This strategy leads to the Restricted Additive Schwarz (RAS) method [40]. Other commonly used strategies include graph-based algorithms [41, 42], algebraic Schur complement approaches [43, 27], incomplete triangular solve [44], and low-rank approximate inverse [45]. The works presented in the following two sections utilize the algebraic Schur complement techniques, and the works presented in the remaining two sections utilize the low-rank approximate inverse techniques.

Many modern distributed-memory systems are also equipped with high-performance GPUs. The total computing power provided by the GPUs on each node is often comparable to, if not greater than, the computing power provided by the CPUs on the same node. Numerous research works have demonstrated the power of GPU-accelerated sparse linear solvers [46, 47, 48, 49, 50, 51, 52, 53, 35, 54]. Most of these works utilize sparse matrix computation kernels to speed up the computation. Packages including

PARALUTION [55] and HIFLOW [56] provide distributed-memory block-Jacobi ILU factorizations with GPU support. In this dissertation, we also discuss the GPU acceleration of our proposed algorithms.

The remainder of this chapter is organized as follows: Section 3.2 reviews several DD-based parallel ILU preconditioning algorithms. Section 3.3 discusses the use of modified ILU to improve the performance and presents the parallel implementation details of several algorithms in the package `hypre`. Section 3.4 presents a parallel low-rank approximate inverse preconditioning algorithm and discusses the implementation details of it in the package `parGeMSLR`. Section 3.5 presents a polynomial low-rank approximate inverse preconditioning algorithm.

3.2 Schur Complement Approach for Sparse Linear Systems

DD-based Parallel ILU factorization typically begins with a reordering step to obtain the reordered system with the following block structure

$$(P^TAP)(P^T\mathbf{x}) = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} = P^T\mathbf{b}, \tag{3.2}$$

where P is a permutation matrix, which is obtained by rearranging the rows or columns of an identity matrix. The above formula is for symmetric reordering, meaning the same reordering is applied to both the rows and columns of the matrix A. It is worth noting that nonsymmetric reordering, denoted as $P_1^TAP_2$, is also possible, but this dissertation focuses solely on symmetric reordering. The block B typically is chosen to allow for efficient parallel computation of B^{-1}. We denote by $A^{(0)}$, $\mathbf{x}^{(0)}$, and $\mathbf{b}^{(0)}$ the reordered coefficient matrix P^TAP, the reordered solution $P^T\mathbf{x}$, and the reordered right-hand side $P^T\mathbf{b}$, respectively.

To solve this system, we use the block LDU factorization of $A^{(0)}$ to write the original linear system in the form:

$$\begin{pmatrix} I & \\ EB^{-1} & I \end{pmatrix} \begin{pmatrix} B & \\ & S \end{pmatrix} \begin{pmatrix} I & B^{-1}F \\ & I \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix}, \tag{3.3}$$

where $S = C - EB^{-1}F$ is called the Schur complement matrix. It is easy to verify that the inverse of the block LDU factorization of $A^{(0)}$ is given by

$$(A^{(0)})^{-1} = \begin{pmatrix} I & -B^{-1}F \\ & I \end{pmatrix} \begin{pmatrix} B^{-1} & \\ & S^{-1} \end{pmatrix} \begin{pmatrix} I & \\ -EB^{-1} & I \end{pmatrix}, \tag{3.4}$$

and thus, the solution can be written as

$$\mathbf{x}^{(0)} = (A^{(0)})^{-1}\mathbf{b}^{(0)} = \begin{pmatrix} I & -B^{-1}F \\ & I \end{pmatrix} \begin{pmatrix} B^{-1} & \\ & S^{-1} \end{pmatrix} \begin{pmatrix} I & \\ -EB^{-1} & I \end{pmatrix} \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix}. \tag{3.5}$$

While the formula contains three occurrences of B^{-1}, we only need to apply B^{-1} to a vector twice each time we apply $(A^{(0)})^{-1}$. We first compute the right-hand side of the Schur complement system, then compute $\mathbf{v}$, and finally compute $\mathbf{u}$ using $\mathbf{v}$. To construct a preconditioner, we replace B^{-1} and S^{-1} with their approximation $\widetilde{B}^{-1}$ and $\widetilde{S}^{-1}$, as summarized in Algorithm 8. We will explain the term *additive* in the algorithm's name later. The key component of a two-level additive preconditioner is the selection of $\widetilde{B}^{-1}$ and $\widetilde{S}^{-1}$.

The subsequent subsection will discuss how the Schur complement approach is used under the DD framework. Specifically, we will discuss the common DD-based Schur complement approach for Edge-based partition and vertex-based partition.

3.2.1 Edge-based partition and vertex-based partition

As discussed in Section 2.3, we employ a simple domain decomposition (DD) strategy that partitions the adjacency graph of $|A| + |A^T|$. In the case of edge-based partitioning, the first block row $[B, F]$ is selected to be the equations associated with unknowns within each subdomain, while the second block row $[E, C]$ is selected as the equations associated with unknowns within vertex-separator. Since there are no connections between unknowns within different subdomains as illustrated in Figure 2.3, if we order the unknowns from the same subdomains together, B will have a block diagonal structure,

Algorithm 8 Two-level additive preconditioning

```
1: Compute $\hat{\mathbf{g}} = \mathbf{g} - E\widetilde{B}^{-1}\mathbf{f}$
2: Solve $\mathbf{v} = \widetilde{S}^{-1}\hat{\mathbf{g}}$ with $\widetilde{S}^{-1} \approx (C - E\widetilde{B}^{-1}F)^{-1}$
3: Solve $\mathbf{u} = \widetilde{B}^{-1}(\mathbf{f} - F\mathbf{v})$
```

and $A^{(0)}$ can be written in the block form

$$A^{(0)} = \left(\begin{array}{cccc|c} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_p & F_p \\ \hline E_1 & E_2 & \cdots & E_p & C \end{array}\right). \tag{3.6}$$

Here, the 2 by 2 structure in Equation 3.2 is visually delineated by horizontal and vertical lines that segment the matrix into distinct blocks. Thus, $\widetilde{B}^{-1}$ can be applied without communication if we use the distributed CSR format and assign all unknowns corresponding to each B_i to the same MPI process for all i. However, communication is required for the matrix-vector multiplications with E and F, as well as for solving the Schur complement system.

When the vertex-based partition is used, we can have more parallelism within the E, F, and C blocks. In this case, the first block row $[B, F]$ is selected as the equations associated with interior unknowns, and the second block row $[E, C]$ is selected as the equations associated with exterior unknowns. We can partition $A^{(0)}$ as

$$A^{(0)} = \left(\begin{array}{cccc|cccc} B_1 & & & & F_1 & & & \\ & B_2 & & & & F_2 & & \\ & & \ddots & & & & \ddots & \\ & & & B_p & & & & F_p \\ \hline E_1 & & & & C_{1,1} & C_{1,2} & \cdots & C_{1,p} \\ & E_2 & & & C_{2,1} & C_{2,2} & \cdots & C_{2,p} \\ & & \ddots & & \vdots & \vdots & \ddots & \vdots \\ & & & E_p & C_{p,1} & C_{2,2} & \cdots & C_{p,p} \end{array}\right). \tag{3.7}$$

Note that when using the vertex-based partition, E and F also have block diagonal structures similar to that of B. This is because elements in E and F correspond to edges that connect interior and interface vertices. The definition of vertex-based partition prohibits any connection between an interior vertex from one subdomain with an interface vertex from another subdomain as illustrated in Figure 2.3.

Compared with the edge-based partition, the vertex-based partition offers the advantage of avoiding communication when computing the matrix-vector multiplication involving E and F. Thus, the only step that requires communication is the solution of the Schur complement system. Moreover, the Schur complement, under the vertex-based partition, also exhibits a special block structure. We can observe directly from Equation 3.7 that the Schur complement can be written as

$$S = \begin{pmatrix} S_1 & C_{1,2} & \cdots & C_{1,p} \\ C_{2,1} & S_2 & \cdots & C_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{p,1} & C_{p,2} & \cdots & S_p \end{pmatrix}, \tag{3.8}$$

where $S_i = C_{i,i} - E_i B_i^{-1} F_i$ are called the local Schur complements. Recall that when the vertex-based partition is used, the matrix

$$\begin{pmatrix} B_i & E_i \\ F_i & C_{i,i} \end{pmatrix} \tag{3.9}$$

belongs to the diagonal block of one of the MPI processes for all i when stored using the distributed CSR format. This indicates that the Schur complement can be formed locally.

On the other hand, the Schur complement obtained via edge-based partition is typically smaller in size than that obtained via vertex-based partition for the same problem with the same number of subdomains. In general, solving the corresponding Schur complement system for the edge-based partition incurs lower computational costs if similar accuracy is desired.

Both partitioning methods have their advantages and disadvantages. However, the selection of the partitioning algorithm does not impact the general algorithms that will be discussed in the following sections. The implementation details and considerations related to the partitioning methods will be discussed further in the implementation part.

3.3 A Parallel Two-level Incomplete LU Preconditioner

In the previous section, we summarized the basic formula of the additive DD-based Schur complement methods. In this section, we show that the performance of the algorithm can be improved by using a Galerkin approach, and forming $\widetilde{B}^{-1}$ using the modified ILU can further speed up convergence for elliptic-type PDEs.

3.3.1 Galerkin product and coarse-grid correction

The Galerkin preconditioning approach start by viewing the Schur complement methods from the perspective of coarse grid correction, similar to the concepts employed in the AMG methods. We can rewrite Equation 3.4 as

$$(A^{(0)})^{-1} = \begin{pmatrix} G & P^{\star} \end{pmatrix} \begin{pmatrix} B^{-1} & \\ & S^{-1} \end{pmatrix} \begin{pmatrix} G^T \\ R^{\star} \end{pmatrix}, \tag{3.10}$$

where $G = (I, 0)^T$. We refer to $P^{\star}$ and $R^{\star}$ as the ideal interpolation and restriction operators, respectively. The formulas of $P^{\star}$ and $R^{\star}$ are

$$P^{\star} = \begin{pmatrix} -B^{-1}F \\ I \end{pmatrix} \quad \text{and} \quad R^{\star} = \begin{pmatrix} -EB^{-1} & I \end{pmatrix}. \tag{3.11}$$

We can easily verify that the coarse-grid operator $R^{\star}A^{(0)}P^{\star}$ is the Schur complement S, and the $(A^{(0)})^{-1}$ can be written as

$$(A^{(0)})^{-1} = GB^{-1}G^T + P^{\star}S^{-1}R^{\star}. \tag{3.12}$$

The ideal operators have an energy-norm minimization property [5].

Algorithm 9 Two-level multiplicative preconditioning

```
1: Compute $\hat{\mathbf{x}}^{(0)} = G\widetilde{B}^{-1}G^T\mathbf{b}^{(0)}$                ▷ F-relaxation
2: Compute $\mathbf{r} = R(\mathbf{b}^{(0)} - A^{(0)}\hat{\mathbf{x}}^{(0)})$             ▷ Restriction
3: Solve $\mathbf{v} = \widetilde{S}^{-1}\mathbf{r}$ with $S = RA^{(0)}P$                ▷ C-correction
4: Compute $\mathbf{x}^{(0)} = \hat{\mathbf{x}}^{(0)} + P\mathbf{v}$                   ▷ Interpolation
```

In the context of preconditioning, we again replace B^{-1} and S^{-1} with their approximations, and construct the approximate operator P and R as

$$P = \begin{pmatrix} -\widetilde{B}^{-1}F \\ I \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} -E\widetilde{B}^{-1} & I \end{pmatrix}. \tag{3.13}$$

Given P and R in Equation 3.13, we summarize the preconditioning approach using the idea of interpolation and restriction in Algorithm 9.

We can verify that the approximation of $\mathbf{x}_0$ given by Algorithm 8 can be written as

$$\mathbf{x}^{(0)} \approx (G\widetilde{B}^{-1}G^T + P\widetilde{S}^{-1}R)\mathbf{b}^{(0)}, \tag{3.14}$$

where the global relaxation term $G\widetilde{B}^{-1}G^T$ and the coarse-grid correction term $P\widetilde{S}^{-1}R$ are applied to $\mathbf{b}_0$ additively. On the other hand, the approximation of $\mathbf{x}^{(0)}$ given by Algorithm 9 can be written as

$$\mathbf{x}^{(0)} \approx \left[G\widetilde{B}^{-1}G^T + P\widetilde{S}^{-1}R(I - A^{(0)}G\widetilde{B}^{-1}G^T)\right]\mathbf{b}^{(0)}, \tag{3.15}$$

where the global relaxation term $G\widetilde{B}^{-1}G^T$ and the coarse-grid correction term $P\widetilde{S}^{-1}R$ are applied to $\mathbf{b}^{(0)}$ multiplicatively. The two algorithms are equivalent when $R = R^\star$ since $R^\star A^{(0)}G = 0$.

When $\widetilde{B}^{-1}$ and $\widetilde{S}^{-1}$ are sufficiently accurate, the performance of these two preconditioning strategies is generally similar. If a less accurate $\widetilde{B}^{-1}$ is used, the multiplicative approach often generates better approximations. On the other hand, the Schur complement in the form $RA^{(0)}P$ is generally denser than $C - E\widetilde{B}^{-1}F$ with the same $\widetilde{B}^{-1}$.

3.3.2 Modified ILU factorizations for building the interpolation

In the previous section, we presented the formula for multiplicative preconditioning and briefly compared the performance of the multiplicative preconditioning and the additive preconditioning using the same $\widetilde{B}^{-1}$. Using the multiplicative algorithm can generally improve convergence behavior when the accuracy of B^{-1} is compromised. For example, when using $ILU(0)$, the approach in Algorithm 9 generally leads to much better performance. In this section, we show that the performance of the multiplicative preconditioning can be further improved using the modified ILU approaches.

One notable advantage of algebraic multigrid (AMG) over ILU methods, particularly for simpler problems like elliptic-type PDEs, is its ability to maintain a nearly constant number of iterations as the problem size increases. This characteristic renders AMG highly efficient for such problems. Given that the multiplicative approach bears similarities to AMG, we can leverage similar concepts and strategies to enhance the performance of the multiplicative algorithm further. Since the P and R depend only on the selection of $\widetilde{B}^{-1}$, we focusing on the selection of $\widetilde{B}^{-1}$.

The intuition behind the use of the modified ILU approaches is that if we use Algorithm 9 as a basic iterative method, the error propagation of it can be written as

$$\hat{\mathbf{e}}_{i+1} := (I - P\widetilde{S}^{-1}RA^{(0)})(I - G\widetilde{B}^{-1}G^T A^{(0)})\hat{\mathbf{e}}_i, \tag{3.16}$$

where $\hat{\mathbf{e}}_i$ denotes the error in the solution after the ith iteration. We can view this as applying a smoothing operator $I - G\widetilde{B}^{-1}G^T A^{(0)}$ followed by a correction operator $I - P\widetilde{S}^{-1}RA^{(0)}$. Suppose $\widetilde{S}^{-1} = (RA^{(0)}P)^{-1}$ is exact, $I - P\widetilde{S}^{-1}RA^{(0)}$ is then an A-orthogonal projector with kernel $\mathrm{Ran}(P)$. This can be verified since for any vector $\mathbf{x}$ we have $P\mathbf{x} - P(RA^{(0)}P)^{-1}RA^{(0)}P\mathbf{x} = \mathbf{0}$. As a result, we can eliminate the error components we hope to eliminate by including them in $\mathrm{Ran}(P)$ so that it becomes near zero after the smoothing step. For elliptic-type PDEs, constant vectors represent the smoothest mode of $A^{(0)}$. Compared to other error modes, the smoothest modes typically converge relatively slowly, which is why interpolation operators used in standard AMG typically interpolate constant vectors exactly. We will show that we can also put constant vectors into the range of P when building it using the ILU factorization $\widetilde{L}_B\widetilde{U}_B \approx B$.

To have a given vector satisfy the following condition

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} \in \operatorname{Ran}(P) \quad \text{where} \quad P = \begin{pmatrix} -\widetilde{U}_B^{-1}\widetilde{L}_B^{-1}F \\ I \end{pmatrix}, \tag{3.17}$$

we need to let

$$-\widetilde{U}_B^{-1}\widetilde{L}_B^{-1}F\mathbf{z} = \mathbf{y} \quad \text{or} \quad \widetilde{L}_B\widetilde{U}_B\mathbf{y} = -F\mathbf{z}. \tag{3.18}$$

We can express B as the sum of the LU factors plus an error term H as

$$B = \widetilde{L}_B\widetilde{U}_B + H. \tag{3.19}$$

We can then write Equation 3.18 as

$$(B - H)\mathbf{y} = -F\mathbf{z} \quad \Leftrightarrow \quad H\mathbf{y} = B\mathbf{y} + F\mathbf{z} \equiv \mathbf{w}. \tag{3.20}$$

Considering the ith outer loop of a row-wise ILU factorization as described in Algorithm 3, we have

$$H_{i,:} = B_{i,:} - \sum_{j \le i} l_{B_{ij}} \widetilde{U}_{B_{j,:}}\ , \quad l_{B_{ii}} = 1, \tag{3.21}$$

where $H_{i,:}$ and $B_{i.:}$ denotes the ith row of matrix H and B, $\widetilde{U}_{B_{j,:}}$ denotes the jth row of the matrix $\widetilde{U}_B$, and $l_{B_{ij}}$ denotes the (i,j)th entry of the matrix $\widetilde{L}_B$. To make the condition $H_{i,:}\mathbf{y} = \mathbf{w}_i$ to be satisfied, we can add a perturbation term to the diagonal element $u_{B_{ii}}$ to get the modified diagonal entry $u_{B_{ii}} + \Delta_i$. By doing so, $H_{i,:}$ becomes $H_{i,:} - \Delta_i \mathbf{e}_i^T$, where $\mathbf{e}_i$ is the i-th column of the identity matrix. Using Equation 3.20, we know that when $\mathbf{y}_i \neq 0$, we can select Δ_i to be

$$\Delta_i = \frac{H_{i,:}\mathbf{y} - \mathbf{w}_i}{\mathbf{y}_i}. \tag{3.22}$$

When selecting $\mathbf{y}$ and $\mathbf{z}$ to be any nonzero constant vectors, we know that $\mathbf{y}_i \neq 0$ for all i. Besides, we have $B\mathbf{y} + F\mathbf{z} = 0$ for elliptic-type PDEs. Thus, Δ_i in Equation 3.22 is $\Delta_i = \sum_j H_{i,j}$, which is the sum of the dropped entries during the factorization of row i. In this case, the algorithm is exactly the standard modified ILU (MILU) algorithm[57].

3.3.3 Implementation details

After discussing the algorithms, in this section, we discuss our implementation details of the several ILU-based preconditioners in `hypre` on both CPUs and GPUs. The implemented preconditioning options include block-Jacobi ILU, restricted additive Schwartz ILU, two-level additive ILU, and two-level multiplicative ILU. While the algorithm itself is independent of the partitioning scheme, our implementation is based on the vertex-based partitioning strategy to align with the interface of `hypre`.

For the block-Jacobi ILU approach and the additive Schwartz ILU approach, the ILU factorization is computed either on the diagonal block or an expanded version of it. Specifically, the diagonal block $A_{i,i}$ in Equation 2.21 is factorized. The factorization is performed independently on each MPI process without the need for communication. In the case of the additive Schwartz ILU, communication is required to gather the extended matrix. Reordering strategies might be used on the local matrix to reduce fill-in and improve the quality of the preconditioner. We implemented the Reverse Cuthill-McKee (RCM) reordering [58]. In our CPU implementation, we do not explicitly form the reordered matrix. Instead, we keep tracking a permutation array which allows us to apply the factorization and triangular solve effectively. The CPU implementation is straightforward, and thus, we will omit its detailed discussion in this section.

For the GPU implementation, during the setup phase, we take advantage of the ILU(0) routine from the `NVIDIA cuSPARSE` library. This routine is based on the level-scheduling algorithm [35, 59]. In this algorithm, the outer loop of the ILU factorization is divided into multiple levels. Each level represents a set of rows that can be computed in parallel. This approach allows for efficient parallel execution of the factorization process. Our current implementation only provides the GPU setup of ILU(0). For other options, the setup is performed on the host, and the results are subsequently transferred to the GPU for further computations. During the solve phase, we use the triangular solve routine from the `cuSparse` library, which is also based on the level-scheduling algorithm.

Another significant difference between our CPU and GPU implementation is the handling of permutation matrices when ILU(0) is used. The `cuSPARSE` routines we utilize do not support the use of permutation arrays, and thus it is necessary to form the permutation matrix explicitly. However, since `cuSPARSE` performs the factorization in-place, no additional memory is required compared to the CPU version. When applying the triangular solve, we use the `gather` and `scatter` functions from `thrust` library to permute the input and output vectors. We summarize the setup phase and the solve phase of the GPU version of our block-Jacobi ILU implementation with ILU(0) in Algorithm 10 and Algorithm 11, respectively.

Algorithm 10 Block-Jacobi ILU setup on GPU

```
1: Compute the RCM ordering $\mathbf{p}_i$ of $A_{i,i}$
2: Reorder $A_{i,i}$ to $A_{i,i}^{\mathbf{p}_i}$ with $\mathbf{p}_i$
3: Call cuSPARSE to compute ILU(0) of $A_{i,i}^{\mathbf{p}_i}$
4: Setup triangular solve
```

Algorithm 11 Block-Jacobi ILU solve on GPU

```
1: Call thrust::gather on $\mathbf{b}_i$ with permutation $\mathbf{p}_i$
2: Call cuSPARSE to solve $L_{A_i}U_{A_i}\mathbf{x}_i = \mathbf{b}_i$
3: Call thrust::scatter $\mathbf{x}_i$ with permutation $\mathbf{p}_i$
```

These algorithms illustrate the steps involved in the GPU setup and solve phases for the block-Jacobi ILU preconditioner. The restricted additive Schwartz ILU has similar steps. We add extra steps to gather the extended matrix with the help of the `hypre_ParCSRMatrixExtractBExt` routine in `hypre`.

Two-level additive ILU

Next, we start to discuss our implementation for the two-level ILU preconditioners. We first present our discussion for the two-level additive ILU preconditioner. As discussed earlier, our implementations in `hypre` use edge-based partitioning. Recall that the local diagonal block $A_{i,i}$ could be written as

$$\begin{pmatrix} B_i & F_i \\ E_i & C_{i,i} \end{pmatrix}. \tag{3.23}$$

One straightforward approach for constructing the preconditioner is to compute the ILU factorization of B_i first. Then, we approximate the local Schur complement S_i

by forming $C_{i,i} - E_i \widetilde{U}_{B_i}^{-1} \widetilde{L}_{B_i}^{-1} F_i$ and apply a dropping strategy to it. Once we have the approximate Schur complement, we can proceed to build an approximation of its inverse.

In our CPU implementation, we compute the ILU factorization using the partial ILU strategy and solve the Schur complement system in the step 2 of Algorithm 8 using the preconditioned GMRES. Compared to the above strategy, using the partial ILU strategy is more efficient and flexible.

We can write the ILU factorization of the entire $A_{i,i}$ as

$$\begin{pmatrix} \widetilde{L}_{B_i} & \\ \widetilde{W}_i & \widetilde{L}_{S_i} \end{pmatrix} \begin{pmatrix} \widetilde{U}_{B_i} & \widetilde{Z}_i \\ & \widetilde{U}_{S_i} \end{pmatrix}, \tag{3.24}$$

where $\widetilde{W}_i \approx E_i \widetilde{U}_{B_i}^{-1}$, $\widetilde{Z}_i \approx \widetilde{L}_{B_i}^{-1} F_i$, and $\widetilde{L}_{S_i} \widetilde{U}_{S_i} \approx S_i$. In the case of partial ILU, only a portion of the ILU factorization is computed. The resulting factorization takes the form:

$$\begin{pmatrix} \widetilde{L}_{B_i} & \\ \widetilde{W}_i & I \end{pmatrix} \begin{pmatrix} \widetilde{U}_{B_i} & \widetilde{Z}_i \\ & \widetilde{S}_i \end{pmatrix}. \tag{3.25}$$

Compared to standard ILU, the factorization of the upper part is the same. However, the inner loop during the partial ILU factorization only applies to the first block column, as illustrated in Figure 3.1.

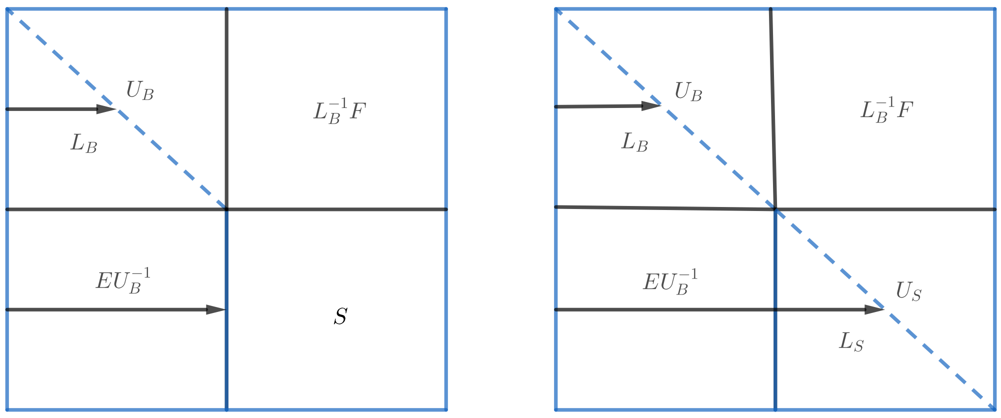


Figure 3.1: Illustration of the partial ILU factorization, which leaves an approximation to the Schur complement at the (2,2) block (left), and the normal ILU factorization, which computes an ILU factorization of the Schur complement as well (right).

Algorithm 12 Two-level additive preconditioning with partial ILU

```
1: Compute $\hat{\mathbf{f}} = \widetilde{L}_B^{-1}\mathbf{f}$
2: Compute $\hat{\mathbf{g}} = \mathbf{g} - W\hat{\mathbf{f}}$
3: Solve $\mathbf{v} = \widetilde{S}^{-1}\hat{\mathbf{g}}$ with GMRES
4: Solve $\mathbf{u} = \widetilde{U}_B^{-1}(\hat{\mathbf{f}} - Z\mathbf{v})$
```

This allows us to form an approximation to the local Schur complement efficiently. Besides, two triangular solve operations can be replaced with matrix-vector multiplication operations, leading to improved computational efficiency. Denote by W and Z the global matrix formed by W_i and Z_i, the two-level additive preconditioning is summarized in Algorithm 12. We provide several different options for preconditioning the Schur complement system. By default, block-Jacobi ILU is used as the preconditioner.

In our GPU implementation, if ILU(0) is not used, the setup phase is again performed on the host, with the results transferred to the GPU later. If ILU(0) is employed, our setup phase is also performed on the device. However, the partial ILU algorithm cannot be utilized due to the lack of support for this algorithm in the `cuSPARSE` library. Forming the Schur complement system can again be costly under this situation, as computing $E_i\widetilde{U}_{B_i}^{-1}\widetilde{L}_{B_i}^{-1}F_i$ is inefficient on GPU. Thus, our GPU implementation for ILU(0) computes the factorization of the entire $A_{i,i}$ as shown in Equation 3.24. After that, our Schur complement system on the device takes the following form:

$$\begin{pmatrix} I & \widetilde{S}_1^{-1}C_{1,2} & \cdots & \widetilde{S}_1^{-1}C_{1,p} \\ \widetilde{S}_2^{-1}C_{2,1} & I & \cdots & \widetilde{S}_2^{-1}C_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{S}_p^{-1}C_{p,1} & \widetilde{S}_p^{-1}C_{p,2} & \cdots & I \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{pmatrix} = \begin{pmatrix} \widetilde{S}_1^{-1}\hat{g}_1 \\ \widetilde{S}_2^{-1}\hat{g}_2 \\ \vdots \\ \widetilde{S}_p^{-1}\hat{g}_p \end{pmatrix}. \tag{3.26}$$

Note that we do not explicitly form the matrix $\widetilde{S}_i$. When applying $\widetilde{S}_i^{-1}$ to a vector, we use two triangular solves with $\widetilde{L}_{S_i}$ and $\widetilde{U}_{S_i}$ instead. We summarize the setup phase and the solve phase of the GPU version of our two-level additive ILU implementation with ILU(0) in Algorithm 13 and Algorithm 14, respectively.

Algorithm 13 Two-level additive ILU setup on GPU

```
1: Compute local permutation $\mathbf{p}_i$
2: Permute $A_{i,i}$ to $A_{i,i}^{\mathbf{p}_i}$ with $\mathbf{p}_i$
3: Call cuSPARSE to compute ILU(0) of $A_{i,i}^{\mathbf{p}_i}$
4: Extract factors $\widetilde{L}_{B_i}$, $\widetilde{U}_{B_i}$ $\widetilde{L}_{S_i}$, $\widetilde{U}_{S_i}$, $\widetilde{W}_i$, and $\widetilde{Z}_i$
5: Setup triangular solve
```

Algorithm 14 Two-level additive ILU solve on GPU

```
1: Call thrust::gather to permute $\mathbf{b}_i$ with $\mathbf{p}_i$
2: Call cuSPARSE to solve for $\mathbf{f}_i' = \widetilde{L}_{B_i}^{-1}\mathbf{f}_i$
3: Compute $\mathbf{g}_i' = \mathbf{g}_i - \widetilde{W}_i\mathbf{f}_i'$
4: Apply GMRES to (12) to solve for $\mathbf{v}$
5: Compute $\mathbf{f}_i'' = \mathbf{f}_i' - \widetilde{Z}_i\mathbf{v}_i$
6: Call cuSPARSE to solve $\mathbf{u}_i = \widetilde{U}_{B_i}^{-1}\mathbf{f}_i''$
7: Call thrust::scatter to permute $\mathbf{x}_i$ with $\mathbf{p}_i$
```

Two-level multiplicative ILU

In this section, we focus on the implementation of the two-level multiplicative approach, specifically the case where modified ILU is used. The implementation of the standard version without modified ILU is relatively simpler and will not be discussed in detail here.

In our CPU implementation, similar to the two-level additive preconditioning, we compute the matrix W and Z to reduce the computation cost. We use the modified ILU factorization of the entire $A_{i,i}$ as shown in Equation 3.24 to build P and R in the form:

$$P = \begin{pmatrix} -U_B^{-1}\widetilde{Z} \\ I \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} -\widetilde{W}L_B^{-1} & I \end{pmatrix} \tag{3.27}$$

We then demonstrate that MILU can also be used to include a given vector in Ran(P). We again include the error term to write the following equation

$$\begin{pmatrix} B & F \end{pmatrix} = \begin{pmatrix} L_BU_B + H_{11} & L_B\widetilde{Z} + H_{12} \end{pmatrix}. \tag{3.28}$$

Again, to have a given vector satisfy the following condition

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} \in \mathrm{Ran}(P) \quad \text{where} \quad P = \begin{pmatrix} -\widetilde{U}_B^{-1}\widetilde{Z} \\ I \end{pmatrix}, \tag{3.29}$$

we need to have

$$-\widetilde{U}_B^{-1}\widetilde{Z}\mathbf{z} = \mathbf{y}. \tag{3.30}$$

By left multiplying $\widetilde{L}_B\widetilde{U}_B$ to both sides of the equation, we obtain

$$\widetilde{L}_B\widetilde{U}_B\mathbf{y} = -\widetilde{L}_B\widetilde{Z}\mathbf{z} \Leftrightarrow (B - H_{11})\mathbf{y} = (H_{12} - F)\mathbf{z}. \tag{3.31}$$

This immediately leads to

$$H_{11}\mathbf{y} + H_{12}\mathbf{z} = B\mathbf{y} + F\mathbf{z} \equiv \mathbf{w}. \tag{3.32}$$

Thus, if we build P and R using $\widetilde{L}_B$, $\widetilde{U}_B$, and $\widetilde{Z}$ from the the MILU of $A_{i,i}$, we can again include constant vectors in $\mathrm{Ran}(P)$.

A significant difference between the additive and multiplicative preconditioning methods is that in the multiplicative approach, the Schur complement is not formed explicitly. As discussed earlier, forming the Schur complement matrix $RA^{(0)}P$ is not practical since the matrix $RA^{(0)}P$ is typically much denser. We keep $RA^{(0)}P$ in its factorized format since GMRES only requires the matrix-vector multiplication operator. The remaining task is to build a preconditioner for the Schur complement. In our case, we use the L_{S_i} and U_{S_i} to construct a block-Jacobi preconditioner for the Schur complement system.

In our GPU implementation, we leave the setup on the host in our current version since MILU(0) is unavailable in `cuSPARSE`. The default option computes two ILU factorizations. One is a standard ILU for the F-relaxation step in Algorithm 9, and the other is a modified ILU for forming R and P. This choice typically yields better performance compared to using MILU during the F-relaxation step. We provide a summary of the setup phase and solve phase in the GPU implementation of our two-level multiplicative ILU method with ILU(0) in Algorithm 15 and Algorithm 16, respectively.

Algorithm 15 Two-level multiplicative ILU solve on GPU

```
1: Compute local permutation $\mathbf{p}_i$
2: Permute $A_i$ to $A^{\mathbf{p}_i}$ with $\mathbf{p}_i$
3: Call cuSPARSE to compute ILU(0) $\widetilde{L}_{A_i}\widetilde{U}_{A_i} \approx A^{\mathbf{p}_i}$
4: Compute MILU(0) of $A^{\mathbf{p}_i}$
5: Extract factors $\widetilde{L}_{B_i}$, $\widetilde{U}_{B_i}$ $\widetilde{L}_{S_i}$, $\widetilde{U}_{S_i}$, $\widetilde{W}_i$, and $\widetilde{Z}_i$ from the MILU(0) of $A^{\mathbf{p}_i}$
```

Algorithm 16 Two-level multiplicative ILU solve on GPU

```
1: Call thrust::gather to permute $\mathbf{b}_i$ with $\mathbf{p}_i$
2: Call cuSPARSE to solve for $\widetilde{L}_{A_i}\widetilde{U}_{A_i}\hat{\mathbf{x}}_i = \mathbf{b}_i$
3: Call cuSPARSE to compute $\mathbf{r} = R(\mathbf{b} - \hat{A}\hat{\mathbf{x}})$
4: Apply GMRES to solve $S\mathbf{v} = \mathbf{r}$
5: Call cuSPARSE to compute $\mathbf{x}_i = \hat{\mathbf{x}}_i + P_i\mathbf{r}_i$
6: Call thrust::scatter to permute $\mathbf{x}_i$ with $\mathbf{p}_i$
```

3.3.4 Numerical experiments

In this section, we demonstrate the performance of our algorithm. Our algorithm is implemented as a part of the `hypre` linear solver library. We run our experiments on the HPC clusters Ray and Lassen of Lawrence Livermore National Laboratory. Each node of Ray has 256 GB memory and consists of 2 IBM POWER8 CPUs (dual-socket) with 20 cores in total and 4 NVIDIA P100 GPUs. Each node of Lassen has 256 GM memory and consists of 2 IBM POWER9 CPUs (dual-socket) with 44 cores in total and 4 NVIDIA V100 GPUs. The CUDA program was compiled using `nvcc` with the option `-gencode arch=compute_60,"code=sm_60"` for P100, and `compute_70, sm_70` for V100 respectively. We run all of the experiments with double-precision arithmetic. To bind MPI processors to physical cores, we use `mpibind` [60], which, on a single node, binds by socket first.

Throughout the rest of this section, unless otherwise specified, we choose Flexible GMRES (FGMRES) as the outer iterative solver. The reason for using FGMRES instead of GMRES is that the preconditioner is not a fixed operator since GMRES is used to solve the global Schur complement system. The restart dimension for FGMRES is set to 50, and the stopping tolerance for the relative residual norm in FGMRES is set equal to 1.0e-8. Unless mentioned otherwise, the solution of the linear system $Ax = b$ will be equal to the vector of all ones with an initial approximation equal to zero.

We denote the block-Jacobi ILU preconditioner by `BJILU`, the two-level additive ILU precondition by `SchurILU`, the two-level multiplicative ILU preconditioner by `RAPILU`, and our two-level multiplicative MILU preconditioner by `RAPMILU`.

Throughout the rest of this proposal, we adopt the following notation:

- $\mathbf{n_p} \in \mathbb{N}$: total number of MPI processes.
- **p-t** $\in \mathbb{R}$: preconditioner setup time.
- **i-t** $\in \mathbb{R}$: iteration time of FGMRES.
- **its** $\in \mathbb{N}$: total number of FGMRES iterations.

Convergence results

We begin our experiment by evaluating the performance of different ILU methods as the preconditioners for FGMRES(50). We use the (M)ILU(0) variants of the factorization.

We consider a Finite Difference discretization of the model problem

$$\begin{aligned} -\Delta u - cu &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{3.33}$$

where 3-pt stencil is used for 1D problem with $\Omega = (0,1)$, 5-pt stencil is used for 2D problem with $\Omega = (0,1)^2$, and 7-pt stencil is used for 3D problem with $\Omega = (0,1)^3$. We test with a 2D problem of size $n = 1024^2$ and a 3D problem of size $n = 512^3$.

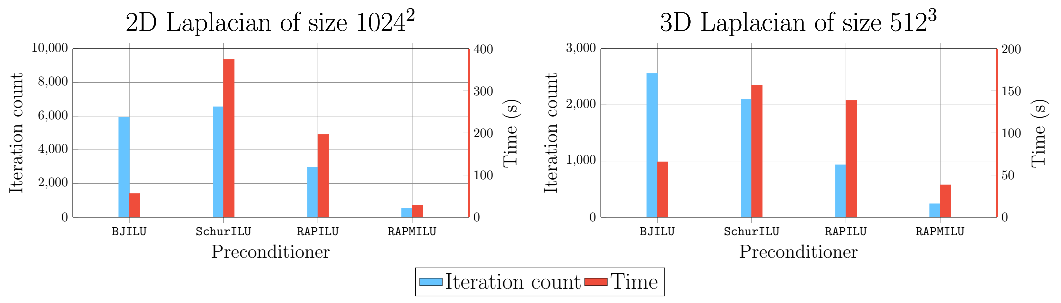


Figure 3.2: Iteration counts and timings of the block Jacobi preconditioner with ILU(0) and the two-level ILU(0)/MILU(0) preconditioners for 2D/3D Laplacian along with FGMRES(50). The runs used 64 processes on 16 nodes on Ray.

Figure 3.2 plots the results with $\mathbf{n_p} = 64$ on 16 nodes on Ray with 4 MPI processes per node. The results indicate that we can benefit by using the RAP strategy with MILU. The reason is that the `RAPMILU` constructs the interpolation and restriction operators that capture the smooth error components. The `BJILU` is the second fastest algorithm in both cases, while its convergence rate is generally slow due to the drop of all inter-domain connections. The `SchurILU` and `RAPILU` typically show intermediate rates of convergence, and the `RAPILU` is better in both tests, as expected. Due to the cost of solving the global Schur complement system, the total time-to-solution for those two approaches is much longer.

We then perform a weak scaling study of the `RAPMILU` strategy. The parallel efficiency is defined as $T_1/T_{\mathbf{n_p}}$, where T_1 and $T_{\mathbf{n_p}}$ denote the wall-clock time achieved by the sequential version and parallel version (with $\mathbf{n_p}$ MPI processes), respectively. We use different preconditioned FGMRES(50) to solve Equation 3.33 and assign 256^2 and 128^3 unknowns to each node for the 2D and 3D problems, respectively.

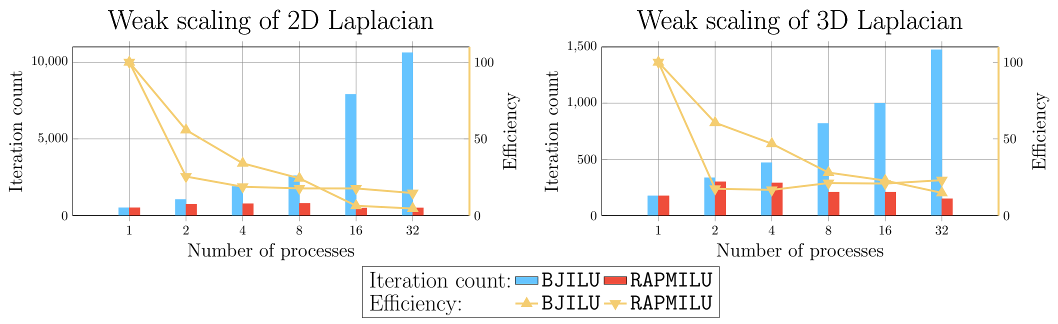
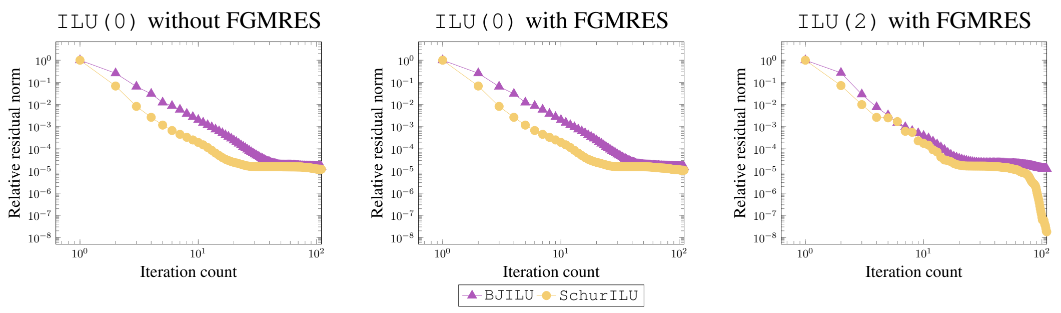


Figure 3.3: Weak scalability study of the `BJILU` and the `RAPMILU` preconditioners with ILU(0)/MILU(0) for 2D/3D Laplacian along with FGMRES(50) with up to 64 MPI processes on up to 16 nodes of Ray. Each MPI process holds 256^2 and 128^3 unknowns for the 2D and the 3D problem, respectively.

Figure 3.3 plots the weak scaling results on up to 8 nodes on Ray with up to 4 MPI processes per node. The results indicate that the `RAPMILU` has better weak scalability since its number of iterations is almost fixed as the number of MPI processes increases. In both cases, we can observe a significant decrease in the efficiency with `RAPMILU` when the number of MPI processes is increased from 1 to 2. This is because `RAPMILU` requires both a classical ILU factorization and a MILU factorization, which increases the setup time. Additionally, the solve time is also increased due to the extra matrix-vector multiplication with S. Despite this, as the number of MPI processes grows, the

RAPMILU method ultimately outperforms the alternative approaches.

In our next experiment, we use our preconditioner in an application in multiphase flow in porous media. We use the problem setup described in [61], and the problem size is $614,400$. Standard AMG is known to be ineffective for this problem. The most commonly used strategy for this type of problem is to solve a global system using an ILU-based approach and solve another reduced system using AMG.



n_p	SchurILU			BJILU		
	its	**p-t**	**i-t**	**its**	**p-t**	**i-t**
1	89	8.75	23.49	89	8.75	23.49
2	80	4.64	10.59	82	4.33	10.47
4	57	2.36	4.13	92	2.10	5.86
8	68	1.19	2.50	96	1.05	3.09
16	74	0.65	1.55	97	0.52	1.61
32	85	0.33	0.98	129	0.24	1.04
64	94	0.17	0.64	222	0.11	0.88

Figure 3.4: Relative residual norm, iteration counts, and timings of the block Jacobi preconditioner with ILU and the two-level additive ILU preconditioners for solving the compositional flow problem. The runs used up to 64 processes on 16 nodes on Ray.

Figure 3.4 provides the results for the multiphase flow problem. In this experiment, we also include the results using different ILU approaches as solvers since ILU is often used as a standalone solver in these applications. We refrain from using modified ILU in this test due to the global problem's lack of a strong elliptic property. As the results indicate, the two-level strategy has a better convergence rate in all tests. In the table, we show the strong scaling results for this problem with up to 64 MPI processes on Rayusing ILU(2). Here we set the convergence tolerance for FGMRES to 1.0e-5 as high accuracy is not a requisite for the global solve phase in this problem. The results show that the two-level ILU has better performance with a large number of MPI processes.

In the next set of experiments, we evaluate the performance of our ILU preconditioner as smoothers for AMG. We use the `BoomerAMG` in `hypre` to solve Equation 3.33 with $f = 1$ and $c = 0$ on a crooked pipe domain. We use `MFEM` to generate a finite element discretization [62, 63], a sample discretization is visualized using `GTLVis` in Figure 3.5 [64].

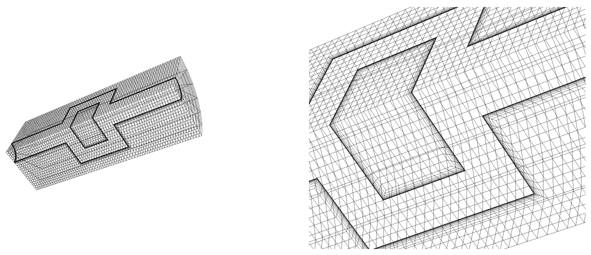


Figure 3.5: A crooked pipe mesh.

We compare the ILU smoothers with the l_1 Jacobi smoother in `hypre` [65]. Note that in our test, we only change the smoother for the finest level. We use l_1 Jacobi smoother for all other levels. It is worth pointing out that the standard `BJILU` fails when solving this problem. Thus, we use an l_1-`BJILU` variant, which adds all entries in the off-diagonal blocks to the diagonal of that row before factorization as shown in the following equation:

$$(A_{i,i})_{k,k} = \sum_{i \neq j} \sum_{l} \left| (A_{i,j})_{k,l} \right|. \tag{3.34}$$

Our experiments were conducted utilizing two commonly used coarsening algorithms in `hypre`: Parallel Modified Independent Set (`PMIS`) and Hybrid Modified Independent Set (`HMIS`). The `PMIS` algorithm provides similar coarse grids for different MPI processes at a lower cost, while the `HMIS` algorithm generally provides higher-quality coarse grids that can vary significantly for different numbers of MPI processes. Table 3.1 shows the results with up to 32 MPI processes on up to 2 nodes on Ray. First, we notice that the iteration count of `SchurILU` is the smallest among all tests for both coarsening algorithms. The wall-clock times of the solve phase of `SchurILU` are also the smallest, except when using $HMIS$ with $\mathbf{n_p} = 8$. The results also indicate that `SchurILU`'s iteration counts are similar for different numbers of MPI processes. However, the setup phase of the two ILU-based options takes longer wall-clock times. Overall, `SchurILU` demonstrates the best performance in terms of total wall-clock time and scalability.

Table 3.1: Iteration counts and timings of the l_1-block-Jacobi smoother with ILU(1) and the two-level additive ILU(1) smoother along with AMG for solving the crooked pipe problem. The number of unknowns is 966,609.

$\mathbf{n_p}$	l_1-Jacobi			l_1-BJILU			SchurILU		
	its	**p-t**	**i-t**	**its**	**p-t**	**i-t**	**its**	**p-t**	**i-t**
AMG with HMIS coarsening									
1	96	8.38	38.75	41	23.29	37.36	41	23.29	37.36
2	141	5.26	29.69	85	12.75	39.35	43	13.55	20.39
4	155	3.42	16.88	80	7.16	18.79	48	7.55	12.17
8	100	1.89	5.86	54	3.72	6.57	46	3.93	6.26
16	146	1.15	4.63	92	2.09	5.83	45	2.20	3.35
32	174	0.65	2.97	106	1.12	3.45	45	1.19	1.86
AMG with PMIS coarsening									
1	284	7.68	100.64	75	22.43	64.76	75	22.43	64.76
2	280	4.38	51.63	102	11.82	44.90	74	12.94	33.08
4	291	2.65	27.91	112	6.35	24.99	72	6.76	17.25
8	295	1.53	14.86	107	3.35	12.16	74	3.56	9.47
16	293	0.92	8.13	102	1.85	6.13	75	1.97	5.31
32	283	0.52	4.32	114	0.98	3.48	73	1.05	2.87

We then evaluate the performance of `SchurILU` smoother with different mesh sizes and orders for the finite element discretization on the same problem using `HMIS`. The results in Table 3.2 show that `SchurILU` has smaller iteration counts in all tests and has smaller total wall-clock time results except for the smallest problem. We could expect an even better overall performance in the situation where multiple right-hand sides are solved with the same linear system.

GPU Speedup

In this section, we study the GPU speedup of our implementation in `hypre`. Since the speedup of single routines of the `cuSPARSE` library has been studied in detail, our experiments focus on the GPU speedup of the two-level ILU options [66]. We again consider Equation 3.33 and solve a 3D problem of size 128^3. We run all tests on a single node on either Ray or Lassen and enable OpenMP threading for all the CPU runs to use all the CPU cores on a single node. To ensure consistent comparison, we use the same number of subdomains for both the CPU and GPU runs, as DD can influence convergence. For the GPU tests on Ray, we use 4 MPI processes, each bound to a GPU. For the corresponding CPU tests, we use 4 MPI processes, each with 5 OpenMP threads. The tests on Lassen have a similar setup, except that the number of OpenMP threads for each MPI process is 11 for the CPU runs.

Table 3.3 shows the performance of the CPU variant and GPU variant of BJILU and two two-level ILU methods and the speedup of all these options. Since the speedup of `RAPILU` and `RAPMILU` is similar, we only report one of them. As the results indicate, we can have a total speedup of a factor of 3.38 for `BJILU`, 2.52 for `SchurILU`, and 1.34 for `RAPMILU` on Ray with P100 GPUs. With the V100 GPUs on Lassen , these numbers are 3.0, 1.2, and 1.74, respectively.

Comparision with other ILU(k)

In our final set of experiments, we compare our new implementation with the `Euclid` library, which is also available in `hypre` [41]. The `Euclid` library implements a two-level graph-based parallel ILU(k) algorithm. We again consider Equation 3.33, and test 3D problems of size 128^3 and 256^3 with ILU(2). In our tests, we increase the dimension of

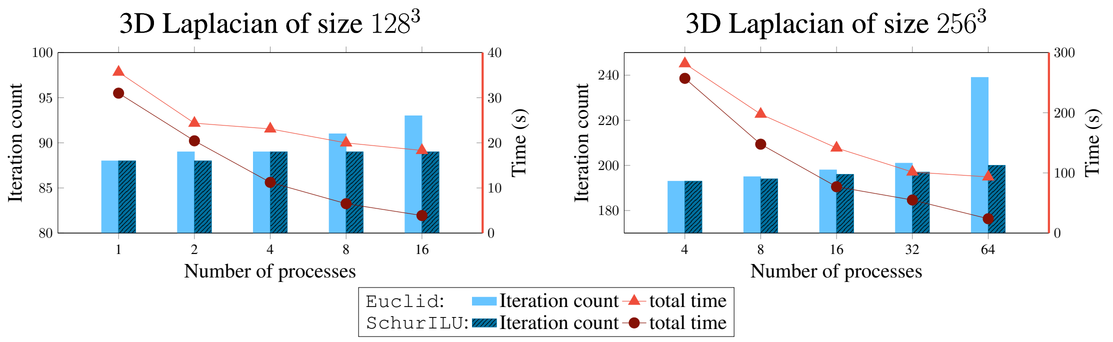
Table 3.2: Iteration counts and timings of the l_1-Jacobi smoother and the additive two-level ILU smoother with ILU(1) for solving the crooked pipe problem with different mesh sizes and different orders of finite elements. The runs used 32 MPI processes on 2 nodes.

Order	**#Unknowns**	**Smoother**	**its**	**p-t**	**i-t**
1	126,805	l_1-Jacobi	82	0.09	0.20
		`SchurILU`	48	0.15	0.30
1	966,609	l_1-Jacobi	174	0.64	2.97
		`SchurILU`	45	1.19	1.86
1	7,544,257	l_1-Jacobi	212	6.91	28.48
		`SchurILU`	58	11.37	18.09
2	126,805	l_1-Jacobi	212	0.10	0.80
		`SchurILU`	30	0.30	0.45
2	966,609	l_1-Jacobi	464	0.72	13.00
		`SchurILU`	47	2.28	4.58
3	414,472	l_1-Jacobi	285	0.53	6.08
		`SchurILU`	27	2.45	2.58
3	3,209,173	l_1-Jacobi	694	3.93	121.43
		`SchurILU`	34	18.57	19.50

Table 3.3: Iteration counts and timings (in seconds) of the CPU and the GPU implementations of the `BJILU`, the `SchurILU`, and the `RAPMILU` preconditioners with ILU(0)/MILU(0) for 3D Laplacian along with FGMRES(50). The size of the problem is 128^3.

Preconditioner	**Device**	**its**	**P100 GPU**		**V100 GPU**	
			p-t	**i-t**	**p-t**	**i-t**
`BJILU`	CPU	229	0.17	8.35	0.15	6.07
	GPU	229	0.64	1.88	0.86	2.04
`SchurILU`	CPU	175	0.20	9.35	0.18	6.02
	GPU	175	0.65	3.14	0.7	5.28
`RAPMILU`	CPU	154	0.20	9.11	0.27	17.96
	GPU	154	1.23	5.70	1.78	10.34

FGMRES to 100 to maintain a similar number of iterations for different preconditioning options.



n_p	Preconditioner	its	p-t	i-t
1	Euclid	88	8.42	27.25
	SchurILU	88	2.37	28.64
2	Euclid	89	10.14	14.22
	SchurILU	88	1.15	19.30
4	Euclid	89	15.59	7.50
	SchurILU	89	0.65	10.59
8	Euclid	91	15.60	4.39
	SchurILU	89	0.33	6.19
16	Euclid	93	15.72	2.60
	SchurILU	89	0.18	3.71

n_p	Preconditioner	its	p-t	i-t
4	Euclid	193	86.17	195.44
	SchurILU	193	5.18	252.02
8	Euclid	195	83.86	113.83
	SchurILU	194	2.83	144.9
16	Euclid	198	81.04	60.44
	SchurILU	196	1.41	75.64
32	Euclid	201	77.35	24.15
	SchurILU	197	0.79	54.21
64	Euclid	239	79.34	14.08
	SchurILU	200	0.38	23.50

Figure 3.6: Strong scalability study of the `Euclid` preconditioner and `SchurILU` preconditioner for 3D Laplacian along with FGMRES(100) with up to 64 MPI processes on 16 nodes on Ray.

The results are presented in Figure 3.6. It can be observed that both ILU strategies exhibit similar iteration counts under the same test settings. While the wall-clock time of the solve phase of `SchurILU` is slightly longer, its setup phase demonstrates significantly higher efficiency compared to `Euclid`. Moreover, the table within the figure highlights the superior strong scalability of `SchurILU`.

3.3.5 Conclusion

In this section, we show the benefits of using the multiplicative two-level ILU preconditioners with modified ILU. We also conduct a detailed analysis of our implementation of several two-level strategies that are readily available in `hypre` with GPU support. The performance of our implementation on distributed-memory systems and with GPU support was demonstrated on both model and real-world problems, verifying the efficiency of the approaches as preconditioners, as standalone solvers, and as smoothers for AMG.

Future work will consider implementing CUDA kernels that support permutation arrays for efficient ILU factorizations and triangular solve. We will also consider a multilevel ILU framework and its GPU version to improve the efficiency of the global Schur complement solve.

We end this section by offering guidance on selecting the parameters for ILU preconditioners impelemented in `hypre`. As a general approach, it is recommended to begin with more cost-effective strategies such as `BJILU` with ILU(0) and `RAPMILU` with ILU(0). If the convergence is not satisfied, increasing the level of fill and switching to ILUT with a small drop tolerance should typically improve the convergence. Combining a more accurate preconditioner with a higher Krylov Subspace Dimension is often beneficial for challenging problems. These adjustable options empower users to strike a balance between cost and accuracy, thereby influencing the overall performance of the preconditioner.

3.4 A Parallel Multilevel Schur Complement Low-Rank Preconditioner

In the previous section, we show that the Schur complement approach could be improved using the multiplicative ILU approach from the AMG viewpoint. However, one limitation of this approach is that the resulting Schur complement, whether in the form of $RA^{(0)}P$ or $C - EB^{-1}F$, is typically much denser than the original matrix C. Thus the same algorithm cannot be applied recursively on the Schur complement. This becomes particularly problematic when the original matrix $A^{(0)}$ is large, leading to a large and potentially challenging Schur complement. In such cases, a multilevel method with better parallel performance may be desired. Another limitation is that the incomplete factorization used in the approach is not easily updatable unless the iterative ParILUT algorithm is used [67, 68]. ParILUT has its own limitations and may fail for complex problems with indefinite matrices. Therefore, a more flexible algorithm that overcomes these limitations can be beneficial in various situations.

In this section, we explore an alternative approach to address the limitations of the previous methods by utilizing low-rank corrections in the Schur complement computation [69, 70, 45, 71, 72]. We present a parallel algorithm based on the GeMSLR multilevel preconditioner [45], which is a non-Hermitian extension of the methods proposed in [69, 70]. We refer to our algorithm parallel GeMSLR, which is implemented in the `parGeMSLR` package.

3.4.1 Schur complement approximate inverse preconditioners via low-rank corrections

The algorithm discussed in this section can be classified as an additive Schur complement method, which has been discussed in Section 3.2. Thus, we only focus on discussing the solution of the Schur complement system in this section. The algorithm starts by expressing the Schur complement $S \in \mathbb{R}^{n_C \times n_C}$ as

$$S = (I - EB^{-1}FC^{-1})C = (I - G)C, \tag{3.35}$$

where $G = EB^{-1}FC^{-1}$. By utilizing the complex Schur decomposition $G = WRW^H$, we can write the above equation as

$$S = (I - WRW^H)C = W(I - R)W^H C.$$

Applying the Sherman-Morrison-Woodbury (SMW) formula, we can express the inverse of the Schur complement as

$$S^{-1} = C^{-1}(I + W[(I - R)^{-1} - I]W^H). \tag{3.36}$$

Applying the inverse of the Schur complement to a vector in this formula requires the solution of a linear system of equations with C and $I - R$, as well as applying the matrix-vector multiplication with W and W^H.

If we compute the inverse of the Schur complement using the above formula, we need to solve linear systems of equations with C and $I - R$ and apply matrix-vector multiplication with W and W^H. In the context of preconditioning, instead of the exact C^{-1} and B^{-1}, we only have their approximation $\widetilde{C}^{-1}$ and $\widetilde{B}^{-1}$. Using those two approximations, we can write the approximate Schur complement as

$$\widetilde{S} = (C\widetilde{C}^{-1} - E\widetilde{B}^{-1}F\widetilde{C}^{-1})\widetilde{C} = (I - \widetilde{G})\widetilde{C}, \tag{3.37}$$

where $\widetilde{G} = E\widetilde{B}^{-1}F\widetilde{C}^{-1} + I - C\widetilde{C}^{-1}$. We can again utilize the complex Schur decomposition $\widetilde{G} = \widetilde{W}\widetilde{R}\widetilde{W}^H$ and use the SMW formula to write the inverse of the approximate Schur complement as

$$\widetilde{S}^{-1} = \widetilde{C}^{-1}(I + \widetilde{W}[(I - \widetilde{R})^{-1} - I]\widetilde{W}^H). \tag{3.38}$$

To build a practical preconditioner, we replace the exact Schur decomposition with a rank k approximation in the form of $\widetilde{G} \approx \widetilde{W}_k\widetilde{R}_k\widetilde{W}_k^H$ where k is a given parameter. Our final approximate inverse preconditioner takes the form

$$\widetilde{S}^{-1} \approx \widetilde{C}^{-1}(I + \widetilde{W}_k[(I - \widetilde{R}_k)^{-1} - I]\widetilde{W}_k^H). \tag{3.39}$$

In practice, we compute the low-rank approximation using the Arnoldi process. We first obtain an approximation in the form:

$$\widetilde{G}\widetilde{V}_m = \widetilde{V}_m\widetilde{H}_m + \beta_m\tilde{\mathbf{v}}_{m+1}\mathbf{e}_m^H, \tag{3.40}$$

where $[\widetilde{V}_m, \tilde{\mathbf{v}}_{m+1}]$ has orthonormal columns, $\mathbf{e}_m$ is the mth column of the identity matrix, and $\widetilde{H}_m$ is upper-Hessenberg. We can obtain our final approximation by performing the complex Schur decomposition $\widetilde{H}_m = QTQ^H$. We define $\widetilde{R}_k$ as the $k \times k$ leading principal submatrix of matrix T, and $\widetilde{W}_k = \widetilde{V}_mQ_k$ where Q_k is the $n_C \times k$ matrix holding the k leading Schur vectors of matrix $\widetilde{H}_m$.

Since $I - \widetilde{R}_k$ is a k by k triangular matrix, its inverse could be formed inexpensively, and thus applying the low-rank correction only involves matrix-vector multiplication which is easily parallelizable. The remaining part of our approximation solves a linear system with the matrix C instead of the matrix S, which addresses the first limitation we address at the beginning of this section since C is generally much sparser.

An extra parameter θ can be introduced to adjust the spectrum of the preconditioned matrix for better clustering results. We can define our final approximation of the Schur complement as

$$\widetilde{S}^{-1} \approx \widetilde{C}^{-1}[(1+\theta)I + \widetilde{W}_k[(I - \widetilde{R}_k)^{-1} - (1+\theta)I]\widetilde{W}_k^H]. \tag{3.41}$$

Note that when selecting $\theta = 0$ we obtain Equation 3.39.

3.4.2 Spectrum analysis

In this subsection, we analyze the relationship between the eigenvalues of G, R, and the preconditioned matrix. The results are very useful when building the low-rank correction terms. The spectrum of the preconditioned matrix of Schur complement low-rank approximate inverse preconditioners was studied in [69] and [71] for SPD problems where the eigenvalues of G are between $(0, 1)$. In this section, we use a more complex form similar to the one used in [69] for SPD problems.

In our analysis, we assume that the inverse of C and B are exact. The following lemma is an extension of the results in [69] to general matrices.

Lemma 3.4.1. *Let the eigenvalues of G be $\lambda_1, \lambda_2, \ldots, \lambda_{n_C}$, and we select the first k eigenvalues to form the low-rank correction, when using the exact inverse of C and B, the eigenvalues of the preconditioned matrix $S\widetilde{S}^{-1}$ are*

$$\begin{cases} \gamma_i = 1 & \text{if } i \le k \\ \gamma_i = (1-\lambda_i)(1+\theta) & \text{if } i > k \end{cases}. \tag{3.42}$$

Proof. Under this situation, we can rewrite $\widetilde{S}^{-1}$ as

$$C^{-1}[(1+\theta)I + W \begin{pmatrix} (I - R_k)^{-1} - (1+\theta)I & O \\ O & O \end{pmatrix} W^H]. \tag{3.43}$$

Simply left multiply it by S. Since R is a triangular matrix, we can easily obtain our results. □

From this lemma, we can see that $S\widetilde{S}^{-1}$ with $\theta = 0$ has k eigenvalues that are equal to 1, and other eigenvalues equal to $1 - \lambda_i$, where λ_i are the eigenvalues of G. To have the eigenvalues clustered at 1, the goal is to find an order to minimize $\max\{|\lambda_i|\}$ for

$i = k+1, \ldots, n_C$. The best option is to put the k eigenvalues of G with the largest magnitude in the leading part in the Schur decomposition $G = WRW^H$.

For SPD problems, if we choose $\theta = \frac{\lambda_{k+1}}{1-\lambda_{k+1}}$, the eigenvalues of the preconditioned matrix would all be greater than 1. The preconditioner typically works better [69]. This strategy could be generalized to general problems. We could compute θ based on the real part of λ_{k+1}. In future work, we will explore more strategies.

Figure 3.7 shows the spectrum of $S\widetilde{S}^{-1}$ for a 8000×8000 matrix obtained with equation 3.33. The matrix is highly indefinite, with 120 negative eigenvalues. The size of S is 780×780. We use the exact inverse for C and B.

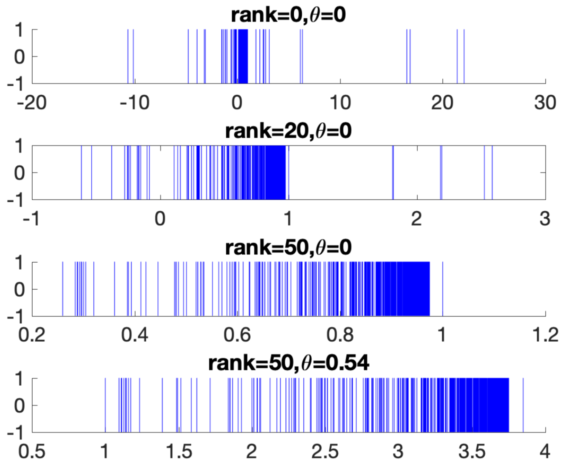


Figure 3.7: Spectrum of $S\widetilde{S}^{-1}$ of a problem of size 8000×8000 with different rank value and θ.

As we can see from the figure, as the rank number increases, the spectral radius of $S\widetilde{S}^{-1}$ decreases. The last plot in the figure shows the effect of a non-zero θ. After setting $\theta = \frac{\lambda_{k+1}}{1-\lambda_{k+1}}$, we shift the eigenvalues of the preconditioned matrix to be greater than 1. In practice, this would typically lead to better performance with a k that is large enough.

3.4.3 Multilevel extensions

For large-scale, high-dimensional problems where C is large, we can recursively apply this approach to C, which leads to a multilevel preconditioner. Let $l_{ev} \in \mathbb{N}$ denote the total number of levels and define the sequence of matrices

$$A^{(l)} = P^{(l-1)}C^{(l-1)}P^{(l-1)} = \begin{bmatrix} B^{(l)} & F^{(l)} \\ E^{(l)} & C^{(l)} \end{bmatrix}, \quad C^{(-1)} = A, \quad l = 0, 1, \ldots, l_{ev} - 2, \tag{3.44}$$

where the 2×2 block matrix partition of each matrix $A^{(l)}$ is obtained by applying DD methods on $C^{(l-1)}$. We can solve the corresponding reordered linear system on level l for $l = 0$ to $l_{ev} - 2$ as

$$\begin{bmatrix} \mathbf{u}^{(l)} \\ \mathbf{v}^{(l)} \end{bmatrix} = \begin{bmatrix} I & -(B^{(l)})^{-1}F^{(l)} \\ & I \end{bmatrix} \begin{bmatrix} (B^{(l)})^{-1} & \\ & (S^{(l)})^{-1} \end{bmatrix} \begin{bmatrix} I & \\ -E^{(l)}(B^{(l)})^{-1} & I \end{bmatrix} \begin{bmatrix} \mathbf{f}^{(l)} \\ \mathbf{g}^{(l)} \end{bmatrix},$$

where $S^{(l)} = C^{(l)} - E^{(l)}(B^{(l)})^{-1}F^{(l)}$ is the Schur complement matrix associated with the l-th level, which is again approximated using $(\widetilde{C}^{(l+1)})^{-1}$ plus a low-rank term as

$$(\widetilde{S}^{(l)})^{-1} = (\widetilde{C}^{(l)})^{-1}(I + \widetilde{W}_k^{(l)}[(I - \widetilde{R}_k^{(l)})^{-1} - I](\widetilde{W}_k^{(l)})^H. \tag{3.45}$$

It is worth noting that we use $\theta = 0$ when deriving the formula for simplicity. Nonzero θ can be used follow Equation 3.41. The approximation $(\widetilde{S}^{(l)})^{-1}$ is approximated recursively except for the last level, where a form of the ILU factorization of the matrix $C_{l_{ev}-1}$ is computed. We will discuss several strategies for building the preconditioner at the last level later. We summarize the construction of the above preconditioner in Algorithm 17.

Algorithm 17 Parallel GeMSLR Setup

```
1: Generate $l_{ev}$-level structure as in 3.44.
2: for $l$ from 0 to $l_{ev} - 2$ do
3:     Compute ILU factorization $L^{(l)}U^{(l)} \approx B^{(l)}$.
4:     Compute matrices $\widetilde{W}_k^{(l)}$ and $\widetilde{R}_k^{(l)}$.
5: end for
6: Compute an ILU factorization $L^{(l_{ev}-1)}U^{(l_{ev}-1)} \approx C^{(l_{ev}-1)}$.
```

After setting up the preconditioner, applying it to a vector is straightforward. The preconditioner can be applied using Algorithm 18. Note that on the first level $l = 0$, it is possible to enhance the preconditioner by using a few steps of GMRES on the Schur complement system.

Algorithm 18 Parallel GeMSLR Solve

```
 1: Apply reordering $\begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} = P^{(l-1)}\mathbf{b}$.
 2: Solve $\hat{\mathbf{f}} = (U^{(l)})^{-1}(L^{(l)})^{-1}\mathbf{f}$.
 3: Compute $\hat{\mathbf{g}} = \mathbf{g} - E^{(l)}\hat{\mathbf{f}}$.
 4: if $l = 0$ then
 5:     Solve $\widetilde{S}^{(l)}\mathbf{v} = \hat{\mathbf{g}}$ by right preconditioned GMRES.
 6: else
 7:     Compute $\hat{\hat{\mathbf{g}}} = \widetilde{W}_k^{(l)}[(I - \widetilde{R}_k^{(l)})^{-1} - I](\widetilde{W}_k^{(l)})^H\hat{\mathbf{g}}$
 8:     Call $\mathbf{v} = \text{pGeMSLRSolve}(\hat{\mathbf{g}} + \hat{\hat{\mathbf{g}}}, l + 1)$.
 9: end if
10: Compute $\mathbf{u} = \hat{\mathbf{f}} - (U^{(l)})^{-1}(L^{(l)})^{-1}F^{(l)}\mathbf{v}$.
11: Apply reordering $\mathbf{x} = P^{(l-1)}\begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}$.
```

Next, we discuss the multilevel reordering approach used in our parallel GeMSLR preconditioner. The MSLR and GeMSLR preconditioner uses a Hierarchical Interface Decomposition (HID) [73] to build the multilevel structure. HID can be obtained using many different approaches. A common approach to obtaining HID is the nested dissection algorithm, which recursively partitions the graph into two subdomains using the edge-based partition. The final adjacency graph is partitioned into $2^{l_{ev}}$ subdomains with a recursion depth of l_{ev}. In the final multilevel structure of A, $B^(l)$ will have $2^{l_{ev}-l}$ diagonal blocks.

HID is unsuitable for our parallel GeMSLR algorithm since most of the $B^{(l)}$ matrices only have limited numbers of diagonal blocks. Instead, we use a multilevel p-way partition algorithm. The graph is first partitioned into p parts using an edge-based partition. Then, the vertex separator is recursively partitioned into p parts until a multilevel structure with desired level is obtained or the size of the last level is sufficiently small. Using this partition, all the $B^{(l)}$ matrices have p diagonal blocks, leading to much better parallel performance than HID. We summarize the reordering algorithm in

Algorithm 19 Parallel GeMSLR Reordering

```
1: Set $C^{(-1)} \equiv A$.
2: for $l$ from 0 to $l_{ev} - 2$ do
3:     Apply $p$-way partitioning to the graph associated with the matrix $|C^{(l-1)}| + |(C^{(l-1)})^T|$.
4:     Set $A^{(l)} = P^{(l-1)} C^{(l-1)} P^{(l-1)} = \begin{bmatrix} B^{(l)} & F^{(l)} \\ E^{(l)} & C^{(l)} \end{bmatrix}$.
5: end for
```

Algorithm 19. It is worth mentioning that similar to the previous section, reordering algorithms, including RCM and approximate minimal degree algorithm (AMD), could be applied on diagonal blocks of $B^{(l)}$ to reduce the memory cost of the preconditioner [74, 75].

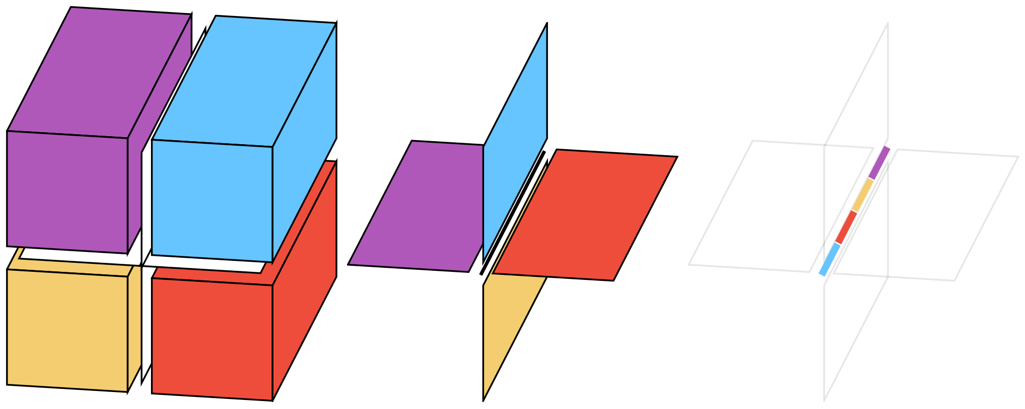
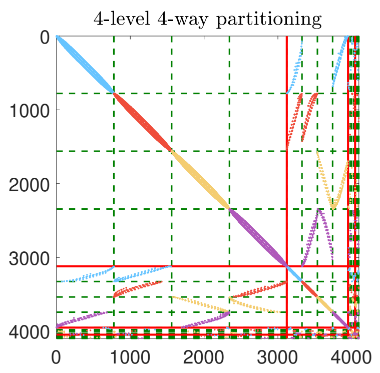


Figure 3.8: Four-way partition of a 3D cube domain (left), the top-level separator (middle), and the second-level separator (right). Those partitions form a four-level, four-way partition.

In Figure 3.8, we present a visualization of a four-level, four-way partition applied to a 3D cube domain. The left figure illustrates the four-way partition on the top level (l=0), with the vertex separator depicted as two white rectangles. Moving to the middle figure, we observe the four-way partition on the second level (l=1), where the vertex separator from the previous level is again partitioned using a four-way partition. The vertex separator on the second level is represented by the black line positioned in the center. Finally, the right figure displays the four-way partition on the third level (l=2), where the vertex separator from the second level undergoes further partitioning using a



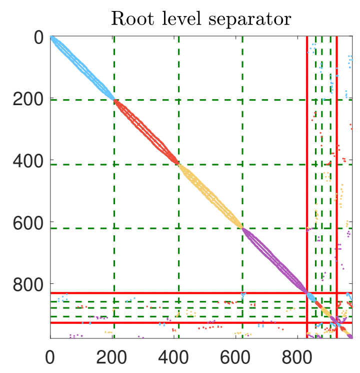


Figure 3.9: Global permutation of matrix A following a multilevel partitioning with $l_{ev} = 4$ and $p = 4$ (left) and zoom-in at the submatrix associated with the permutation of the vertex separators (right).

four-way partition. The vertex separator on the last level consists of only three vertices.

We also plot the reordered matrix corresponding to a 3D problem on a cube domain using Equation 3.33 using finite difference discretization. We use a four-level, four-way partition as illustrated earlier and plot the multilevel structure in Figure 3.9. The left figure shows the global matrix, while the right figure shows a zoom-in of the top-level separator. All the $B^{(l)}$ blocks are reordered using RCM in the figure. As we can see from the figure, the $B^{(l)}$ matrices on all levels have four diagonal blocks, which is more suitable for parallel computation than HID.

3.4.4 Implementation details

In this section, we discuss the implementation of our parallel GeMSLR algorithm in the library `parGeMSLR`. As discussed in the previous section, the implementation of our preconditioner consists of three major components: the reordering phase, the setup phase, and the solve phase.

Reordering phase

The first step for setting up the parallel GeMSLR preconditioner is to apply the recursive p-way partitioning and build the multilevel structure. Our current implementation takes advantage of the parallel graph partition routines in the package `ParMETIS` [76]. Since `parMETIS` does not directly provide the vertex separator, we implement a simple strategy for finding a vertex separator. Denote by $\mathcal{G}_1$ and $\mathcal{G}_2$ the induced subgraph of two vertices set obtained by partitioning an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{I})$ using the vertex-based partition. We can easily check that the set of vertices in $\mathcal{G}_1$ with at least one neighborhood in $\mathcal{G}_2$ forms a vertex separator for a two-way edge-based partition of $\mathcal{G}$. By recursively applying this strategy, we can build a p-way edge-based partition and find the vertex separator using a p-way vertex-based partition. This strategy typically does not lead to the smallest size. However, considering the computation and communication cost, this is currently the default option in our implementation.

Once we finish the multilevel partitioning, the ith row/column is given a new row/column number $\mathbf{p}[i]$, which requires a global redistribution of the original matrix across all MPI processes. In our implementation, we apply the row permutation first, followed by the column permutation. The row permutation begins with a global communication so that each MPI process knows where to receive the data. We currently use a single `MPI_Alltoall` communication to achieve this step since neighborhood information of the new linear system is unknown. After knowing the neighborhood information, point-to-point communications between neighboring MPI processes are used to redistribute rows. We then use a hashing-based implementation to reorder columns and avoid forming the global permutation array. We first search through all local entries and put columns into a hash table, then apply similar point-to-point communication to gather the new column indices, and finally apply the permutation.

We also provide an option that utilizes the edge-based partition on the top level and the vertex-based partition on the subsequent levels. This option slightly reduces the setup cost, but the size of the top-level Schur complement generally is much larger.

Setup phase

Next, we discuss our implementation of the setup phase of the parallel GeMSLR algorithm in `parGeMSLR`. The setup phase can be further divided into two parts, the computation of the ILU factorization and the construction of the low-rank correction.

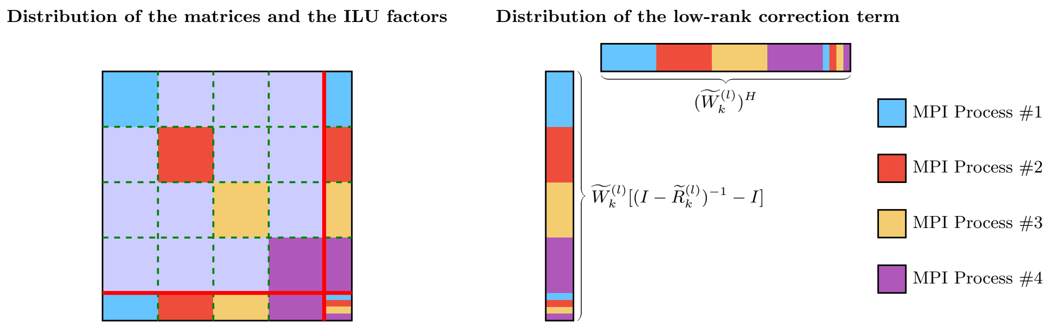
The ILU factorization of $B^{(l)}$ matrices can be easily computed in parallel. Thus, we only discuss the factorization of the last level $C_{l_{ev}-2}$ here. In this multilevel framework, the size of the last level can be very small. Thus, simply computing the ILU factorization redundantly on all MPI processes can be a useful approach. However, this approach will soon become impractical as the size of the last level grows. On the other hand, if we use distributed strategies that require communication, the communication overhead can be large on the last level. The default option implemented in `parGeMSLR` is a simple block-Jacobi ILU approach. The matrix $C_{l_{ev}-2}$ is reordered using RCM ordering and distributed evenly to all MPI processes, and ILU factorization is applied to diagonal blocks. We also provide an implementation of the power Schur complement (PSLR) method described in [77]. This approach uses a low-rank correction to improve the accuracy of the block-Jacobi ILU further.

To build the low-rank correction, we implemented the thick-restart Arnoldi algorithm in `parGeMSLR`. The Arnoldi algorithm is applied to $\widetilde{G}^{(l)}$ on each level to compute the low-rank correction. Recall that applying $\widetilde{G}^{(l)}$ requires $(\widetilde{C}^{(l)})^{-1}$. Thus, building low-rank correction is done after the ILU factorization step, and the low-rank correction terms are built in bottom-up order. Besides the communication cost spent in applying $\widetilde{G}^{(l)}$, the Arnoldi algorithm requires the orthogonalization of the Krylov basis. `MPI_Allreduce` is required during the computation. We choose the classical Gram-Schmidt process with reorthogonalization in our implementation to reduce communication costs [78]. The Srhur decomposition and eigendecomposition are done using `LAPACK` redundantly on all MPI processes. The final $\widetilde{W}_k^{(l)}$ is stored in a row-distributed format using the same distribution as that of the global matrix.

Solve Phase

Finally, we describe our implementation of the solve phase. The solve phase can be further divided into three parts, applying the triangular solve, applying matrix-vector multiplication with $E^{(l)}$ and $F^{(l)}$, and applying the low-rank correction.

The parallelization of the triangular solves is trivial. The MPI process that is responsible for a certain ILU factorization is responsible for the corresponding triangular solve operation. Applying the triangular solve can be done without communication in parallel if block-Jacobi ILU is used on the last level. Applying the matrix-vector multiplication is also straightforward. We simply use standard distributed matrix-vector multiplication for row-distributed CSR matrix. Point-to-point communication is required between neighborhood MPI processes. Finally, applying the low-rank correction requires a `MPI_Allreduce` operation after the matrix-vector multiplication with $(\widetilde{W}_k^{(l)})^H$.



Distribution of the matrices and the ILU factors
Distribution of the low-rank correction term
$(\widetilde{W}_k^{(l)})^H$
$\widetilde{W}_k^{(l)}[(I-\widetilde{R}_k^{(l)})^{-1}-I]$
MPI Process #1
MPI Process #2
MPI Process #3
MPI Process #4

Figure 3.10: Layout of the matrics and ILU factors (left) and the low-rank correction terms (right) across four MPI processes on the last two levels with a four-way partition.

We compute and store $\widetilde{W}_k^{(l)}[(I-\widetilde{R}_k^{(l)})^{-1}-I]$ in a matrix so that applying it can be done in parallel.

In Figure 3.10, we visualize the distribution of the matrices, ILU factors, and the low-rank correction terms on the last two levels using a four-way partition.

Communication analysis

Finally, we summarize the total communication cost in our implementation. We conduct our analysis under the assumption that the total number of MPI processes $\mathbf{n_p}$ is equal to the p used in the p-way partition and a constant rank k is used on all levels.

We first consider the cost of applying the operator $(\widetilde{C}^{(l)})^{-1}$. When applying this operator, according to Algorithm 18, we need to apply matrix-vector multiplication with $E^{(l+1)}$ and $F^{(l+1)}$, triangular solve with the ILU factors of $B^{(l+1)}$, apply the low-rank correction with $(\widetilde{W}_k^{(l)+1})^H$ and $\widetilde{W}_k^{(l+1)}[(I-\widetilde{R}_k^{(l+1)})^{-1}-I]$, as well as a recursive solve with $(\widetilde{C}^{(l+1)})^{-1}$, until the last level is reached. The triangular solve operations can be done in parallel, and the matrix-vector multiplication requires only point-to-point communications. Thus, the major cost comes from the $l_{ev}-(l+2)$ `MPI_Allreduce` operations from the low-rank correction on each level. Thus m steps of Arnoldi iterations using classical Gram-Schmidt with reorthogonalization require $(l_{ev}-l)m$ `MPI_Allreduce` operations. Here, most of the operations comes from the $(\widetilde{C}^{(l)})^{-1}$ operation when computing matrix-vector multiplication with $\widetilde{G}^{(l)}$. Each iteration also requires two extra `MPI_Allreduce` operations during the Gram-Schmidt procedure. In our implementation, we set $m=2k$ by default. Thus, the total communication overhead is bounded by

$2k\delta(k)\sum_{l=0}^{l=l_{ev}-2}(l_{ev}-l)$ `MPI_Allreduce` operations, where $\delta(k)-1$ equals the number of restarts performed during the Arnoldi iterations. Similarly, we can see that applying the preconditioner requires $l_{ev}-1$ `MPI_Allreduce` operations.

The results indicate that the communication cost is associated with the total number of levels l_{ev} and the rank k. This indicates that the total number of levels should not be too large in practice when p is large. In applications, l_{ev} typically will not be not large since our recursive p-way partition generally could reduce the vertex separator to a very small size with three to four levels, as indicated in Figure 3.8.

3.4.5 Numerical experiments

In this section, we demonstrate the parallel performance of our parallel implementation in the `parGeMSLR` linear solver library. We run our pure CPU experiments on the HPC cluster QUARTZ cluster of Lawrence Livermore National Laboratory. Each node of QUARTZ has 128 GB memory and consists of 2 Intel Xeon E5-2695 CPUs with 36 cores in total. We use MVAPICH2 2.2.3 to compile `parGeMSLR`. Again, all of the experiments presented below are executed with double-precision arithmetic. However, it is worth mentioning that `parGeMSLR` supports both real and complex arithmetic, as well as both single and double precision. The current version of `parGeMSLR` uses `LAPACK` for sequential matrix decompositions and `ParMETIS` for distributed graph partitioning [76]. On top of distributed-memory parallelism, `parGeMSLR` can take advantage of shared memory parallelism using either OpenMP or CUDA. The GPU speedup is obtained on the HPC cluster RAY and LASSEN discussed in the previous section.

Similar to the previous section, we choose FGMRES(50) as the outer iterative solver. The reason is that the preconditioner is not fixed if we enable the inner solver in step 9 of Algorithm 18. The stopping tolerance for the relative residual norm in FGMRES is set equal to 1.0e-6. Unless mentioned otherwise, the solution of the linear system $Ax=b$ will be equal to the vector of all ones with an initial approximation equal to zero. We compute the low-rank correction to two digits of accuracy using thick-restart Arnoldi with a restart cycle of $2k$.

We compare our preconditioner with a) the BoomerAMG parallel implementation of the algebraic multigrid method in `hypre`, and b) the two-level additive `SchurILU` approach in `hypre` discussed in the previous section.

Throughout the rest of this section, we adopt the following notation in addition to the notation used in the previous section:

- **k** $\in \mathbb{N}$: number of low-rank correction terms at each level.
- **fill** $\in \mathbb{R}$: the ratio between the number of non-zero entries of the preconditioner and that of matrix A.
- **r-t** $\in \mathbb{R}$: reordering time. This includes the time required to compute the ILUT factorizations and low-rank correction terms in `parGeMSLR`.
- **F**: flag signaling that FGMRES failed to converge within 1000 iterations.

Model Problem

In the first set of experiments, we again consider the model problem in Equation 3.33

We begin our experiment by evaluating the weak scalability of our implementation. Our first experiment studies the weak scaling efficiency of the reordering phase, setup phase, and solve phase. Regarding the solve phase, we study the per-iteration weak scalability of our algorithm. We pick $c = 0$ and fix the problem size on each MPI process at 50^3. We select $p = 8\mathbf{n_p}$ and run our tests with $l_{ev} \in \{2, 3\}$ and $k \in \{0, 100, 200\}$.

Figure 3.11 plots the results of our first experiment on up to $\mathbf{n_p} = 1,024$ MPI processes on QUARTZ. We can observe from the figure that we can always achieve higher efficiency with $l_{ev} = 3$. However, the overall efficiency is similar across most of the tests. Additionally, the preconditioner demonstrates higher weak scaling efficiency with $k = 0$, as this option avoids communication from the low-rank correction terms. Nevertheless, a drop in efficiency is noticeable as $\mathbf{n_p}$ increases, primarily due to the load imbalance introduced by the DD. Another significant factor contributing to the efficiency drop is the increase in the size of the local Schur complement caused by the existence of boundaries. This is illustrated in Figure 3.12 with a simple example. Consequently, a relatively higher drop in efficiency is observed when the number of MPI processes is small. The results also highlight limited efficiency in the reordering phase due to the weak scalability limitations of graph partitioning and global reordering.

Next, we show the timing results using preconditioned FGMRES(50) with `parGeMSLR` and `SchurILU`. Figure 3.13 plots the weak scalability of `parGeMSLR` and `SchurILU`. This time we allow enough iterations for FGMRES to converge. As previously, we use eight

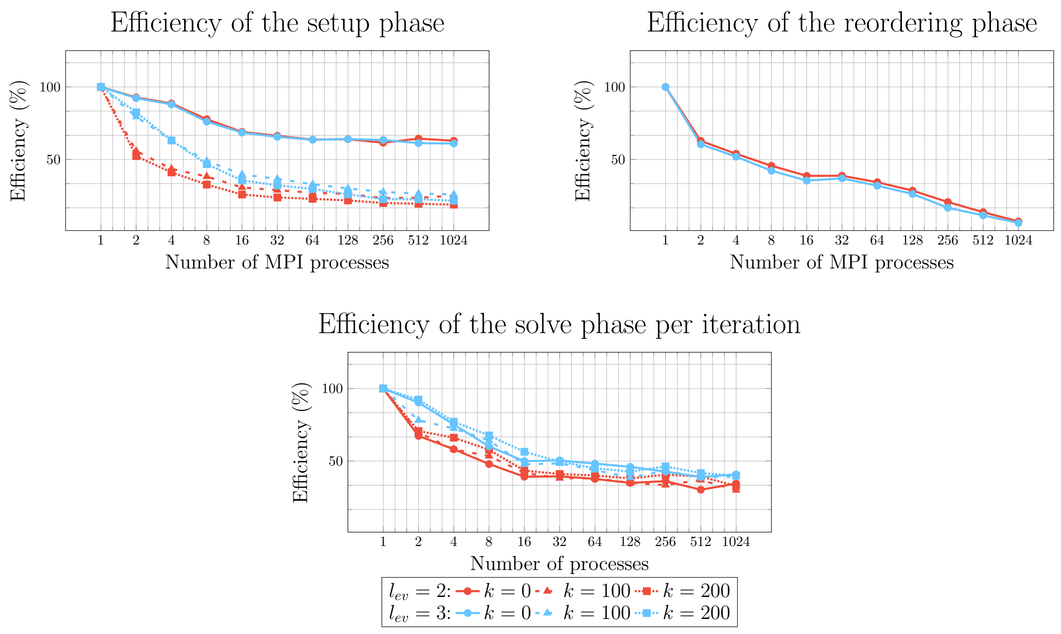


Figure 3.11: Weak scaling of `parGeMSLR` for the Laplacian when the number of iterations performed by FGMRES is fixed to thirty, and the number of levels is set to $l_{ev} = 2$ and $l_{ev} = 3$. The number of unknowns on each MPI process is $125,000$, for a maximum problem size $n = 800 \times 400 \times 400$.

subdomains per MPI process, but this time we fix $l_{ev} = 3$ and $k = 10$. In summary, `parGeMSLR` is faster and more scalable than `SchurILU` during the solve phase. Moreover, `parGeMSLR` also converges much faster than `SchurILU`, and the number of iteration counts increases only marginally with the problem size. On the other hand, the weak scaling of the preconditioner setup phase of `parGeMSLR` is impacted negatively as the problem size increases due to the need to perform more Arnoldi iterations to compute the low-rank correction terms.

We now present strong scaling results. We study the strong scaling results by fixing the size of the problem and varying the number of MPI processes. We fix the problem size to be 320^3 and select $p = 2,048$. We run our tests with $l_{ev} \in \{2, 3\}$ and $k \in \{0, 50, 100\}$.

Figure 3.14 plots the strong scaling results on up to $\mathbf{n_p} = 1,024$ MPI processes on Quartz. We can observe from the figure that we can always achieve higher efficiency with $l_{ev} = 2$. Besides, the strong scaling efficiency is more sensitive to the number of

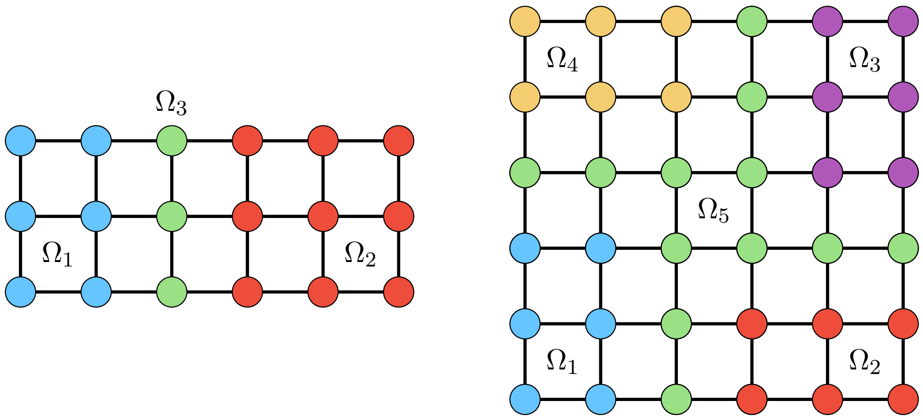


Figure 3.12: A 2-way partitioning of a 3×6 discretized domain obtained (left) and a 4-way partitioning of a 6×6 discretized domain (right). In both figures, green vertices correspond to the vertex separators. The size of the vertex separator increases by a factor of four when the size of the problem increases by a factor of two.

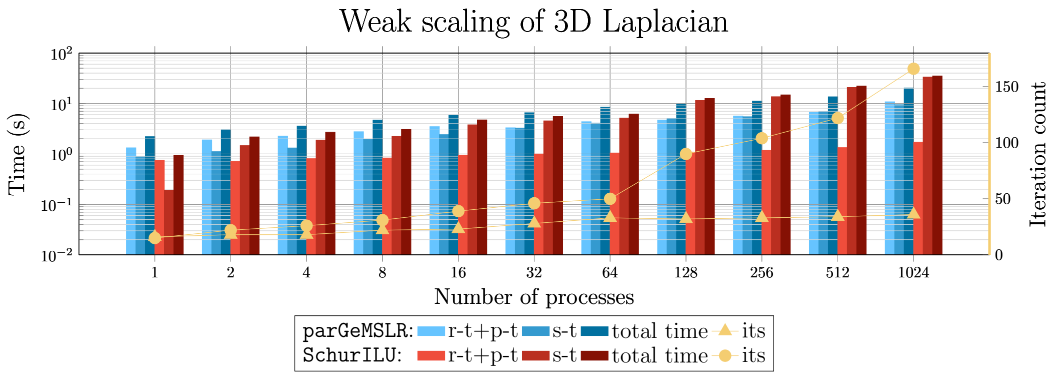


Figure 3.13: Weak scaling of **parGeMSLR** and SchurILU on Laplacian. The number of unknowns on each MPI process is $125,000$, for a maximum problem size $n = 800 \times 400 \times 400$.

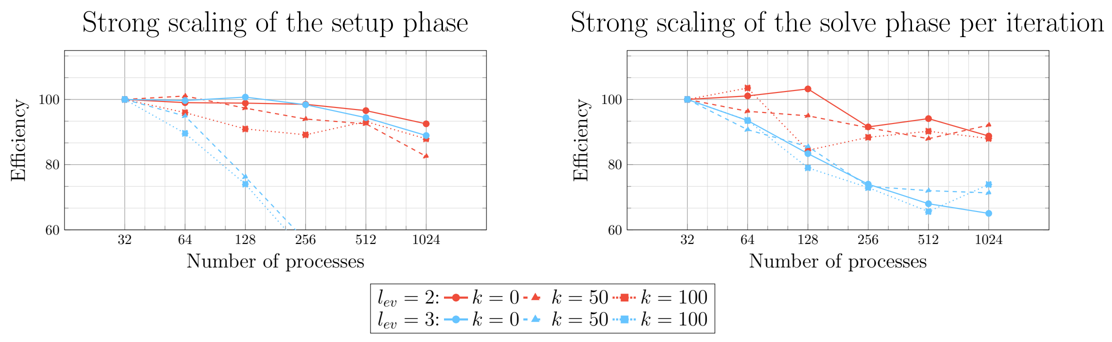


Figure 3.14: Strong scaling results for Laplacian of size $n = 320^3$. The number of subdomains is set equal to $2,048$ in all levels.

levels. We again observe a better efficiency with $k = 0$. Nonetheless, all the options exhibit good parallel efficiency.

In the final test in this section, we evaluate the performance of our implementation using the crooked pipe domain discussed in the previous section. The visualization of the mesh can be found in Figure 3.5. We use different levels of mesh refinement to generate problems of three different sizes. We vary the number of inner iterations in step 5 of Algorithm 18 between three to five for different problem sizes. Our implementation is compared against `SchurILU` and `BoomerAMG` with HMIS coarsening. We test both the Gauss-Seidel smoother and the l_1-Jacobi smoother.

Table 3.4 reports our results on up to $\mathbf{n_p} = 64$ on QUARTZ. As we can see from the table, `parGeMSLR` has the smallest iteration counts and solve phase wall-clock times in most of the tests, except for `BoomerAMG` with the Gauss-Seidel smoother. The performance of the `parGeMSLR` is much better than `SchurILU` and `BoomerAMG` with the l_1-Jacobi smoother.

Linear elasticity equation

The second problem we consider in our experiments is the following linear elasticity equation:

$$\mu \Delta u + (\lambda + \mu)\nabla(\nabla \cdot u) \quad = \quad f \quad \text{in } \Omega, \tag{3.46}$$

Table 3.4: Comparison between the iteration counts and timings of `SchurILU`, `BoomerAMG`, and the `parGeMSLR` on up to 64 MPI processes for solving Equation 3.33 on a crooked pipe mesh domain.

Preconditioner	**#Unknowns**	$\mathbf{n_p}$	**k**	**fill**	**p-t**	**i-t**	**its**
`BoomerAMG` GS	126,805	16	-	1.71	0.17	0.69	106
	966,609	32	-	1.79	0.79	5.7	198
	7,544,257	64	-	1.81	3.36	45.12	250
`BoomerAMG` Jacobi	126,805	16	-	1.71	0.18	1.29	226
	966,609	32	-	1.79	0.8	10.95	431
	7,544,257	64	-	1.81	3.39	72.1	568
`SchurILU`	126,805	16	-	1.53	0.22	0.51	65
	966,609	32	-	1.86	1.2	12.46	383
	7,544,257	64	-	1.94	5.51	-	**F**
`parGeMSLR`	126,805	16	10	1.05	0.54	0.46	25
	966,609	32	10	1.18	3.59	4.70	53
	7,544,257	64	10	1.32	11.76	48.35	128

where u is the displacement, f is the force, while λ and μ are the material's Lamė constants. We discretize the problem using first-order FEM. We select Ω as a 3D cantilever beam, as shown in Figure 3.15. The left end of the beam is fixed, while a constant force pulls down the beam from the right end. We set $\mu = 1$ and test $\lambda = 80$ and $\lambda = 80$. The Poisson ratio λ/μ are approximately 0.455 and 0.494, respectively, which makes the problems challenging. We again use different levels of mesh refinement to generate problems of three different sizes and vary the number of inner iterations in step 5 of Algorithm 18 between three to five for different problem sizes. We also increase the rank k as the size of the problem increases. Our implementation is compared against `SchurILU` and `BoomerAMG` with HMIS coarsening. We test both the Gauss-Seidel smoother and the l_1-Jacobi smoother. However, `BoomerAMG` options fail to solve all linear elasticity problems, so we exclude them from the table.

Table 3.5 presents the results of this set of experiments. As the ratio $\frac{\lambda}{\mu}$ approaches 0.5, the problem becomes more ill-conditioned, leading to better results for $\lambda = 10$ compared to $\lambda = 80$. From the results, it can be observed that `parGeMSLR` outperforms `SchurILU` in terms of total time for all tests, except for two small tests with $\lambda = 10$. The advantage of `parGeMSLR` becomes even more significant when multiple right-hand sides are solved using the same coefficient matrix.

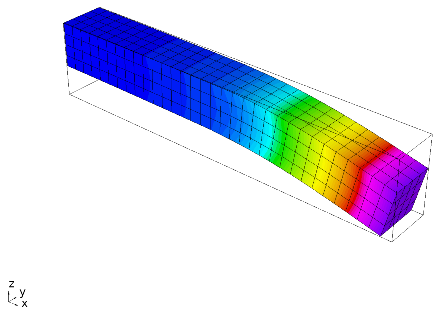


Figure 3.15: Linear elasticity problem on a 3D beam.

Table 3.5: Comparison between the iteration counts and timings of `SchurILU` and the `parGeMSLR` on up to 64 MPI processes for solving Equation 3.46 on a beam mesh. `BoomerAMG` options fail to solve all linear elasticity problems, so we exclude them from the table.

Preconditioner	**#Unknowns**	$\mathbf{n_p}$	**k**	**fill**	**p-t**	**i-t**	**its**
			$\mu = 1,\ \lambda = 10$				
`SchurILU`	2,475	4	-	2.62	0.03	0.06	49
	15,795	8	-	3.78	0.32	0.60	238
	111,843	16	-	7.81	4.80	19.05	751
	839,619	64	-	11.82	19.67	-	**F**
`parGeMSLR`	2,475	4	20	1.94	0.12	0.01	18
	15,795	8	40	3.58	0.92	0.04	23
	111,843	16	40	7.86	10.06	0.64	41
	839,619	64	80	10.05	63.25	3.13	65
			$\mu = 1,\ \lambda = 80$				
`SchurILU`	2,475	4	-	2.21	0.03	0.26	336
	15,795	8	-	4.03	0.35	1.48	549
	111,843	16	-	8.94	6.45	-	**F**
	839,619	64	-	14.75	32.17	-	**F**
`parGeMSLR`	2,475	4	20	1.91	0.15	0.01	41
	15,795	8	40	3.58	1.09	0.15	75
	111,843	16	80	6.48	16.16	1.49	93
	839,619	64	120	10.31	133.2	6.15	128

Helmholtz equation

In the next experiment, we consider the Helmholtz problem

$$-(\Delta + \omega^2)u = f \quad \text{in } \Omega = [0,1]^3, \tag{3.47}$$

where we use the Perfectly Matched Layer (PML) boundary condition [34] and set the number of points per wavelength is equal to eight. We used random initial guesses in our tests. Note that since the problem is complex, we use the complex version of `parGeMSLR`

In our first experiment, we fix the problem and vary the rank k. The size of the problem is set to 50^3, and the number of levels is fixed at three. We conduct tests with different rank values ranging from 10 to 100, and the fill-in of the low-rank correction term is approximately three when $k = 100$.

Figure 3.16 shows the test results with 16 MPI processes on Quartz. The figure presents the total wall-clock time and the wall-clock time specifically for the solve phase. It can be observed that the total time consistently decreases as the rank increases. Although the wall-clock time for the setup phase increases as the rank increases, a large rank leads to a smaller iteration count. The utilization of low-rank correction significantly improves the performance of this particular problem.

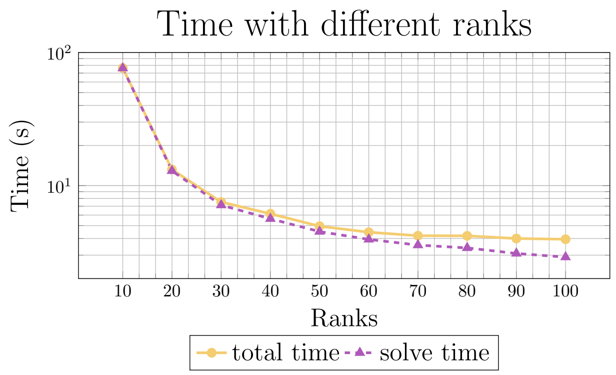


Figure 3.16: Total and iteration wall-clock times of the 3-level `parGeMSLR` to solve the Helmholtz equation of size $n = 50^3$ using 16 MPI processes.

Next, we use different ω values to generate problems of three different sizes. Since the problems with large sizes are challenging, we add a complex shift $0.05\mathrm{i} * \sum_j |A_{jj}|/n_A$ during the ILU factorization. Here, we introduce the symbol i in the upright style to

Table 3.6: `parGeMSLR` with/without complex shifts for Helmholtz problem. The problem size is equal to $n = (4\omega/\pi)^3$.

	with shift							without shift		
ω	$\mathbf{n_p}$	**k**	**fill**	**r-t**	**p-t**	**i-t**	**its**	**fill**	**total time**	**its**
5π	1	0	3.40	0.04	0.02	0.05	9	3.80	0.12	9
7.5π	1	0	3.81	0.17	0.10	0.40	20	4.76	6.47	241
10π	2	5	3.52	0.43	0.41	1.03	36	4.11	15.48	449
12.5π	4	5	3.79	0.70	0.58	1.50	42	4.79	-	**F**
15π	8	10	4.16	1.25	1.20	2.33	55	4.63	-	**F**
20π	16	10	4.40	1.51	1.29	3.51	57	4.77	-	**F**
40π	64	20	5.49	4.87	7.84	14.43	92	5.73	-	**F**

represent the imaginary unit. This strategy is known to be useful for Helmholtz equation [79, 45, 80].

We present the results in Table 3.6. As we can see from the table, our implemented preconditioner can efficiently solve this problem. Besides, adding a complex shift helps create a more stable ILU for this problem.

GPU Speedup

Finally, we consider a $n = 128^3$ discretization of the model problem (3.33) and focus on the speedup achieved during the solution phase if GPUs are enabled. We set the number of levels equal to $l_{ev} = 2$ and $l_{ev} = 3$, and vary the low-rank correction terms as $k \in \{0, 100, 200, 300, 400, 500\}$. At each level, we apply a 4-way partition and assign each partition to a separate MPI process bound to a V100 NVIDIA GPU. Figure 3.17 plots the speedups achieved by the hybrid CPU+GPU version of `parGeMSLR` during its solve phase. As expected, the peak speedup is obtained for the case $k = 500$, since the cost to apply the low-rank correction term increases linearly with the value of k.

3.4.6 Conclusion

In this section, we show a multilevel parallel Schur complement low-rank preconditioner and study its convergence as well as parallel performance. The algorithm is implemented in `parGeMSLR`, a C++ parallel software library for the iterative solution of general sparse systems distributed among several processor groups communicating via MPI.

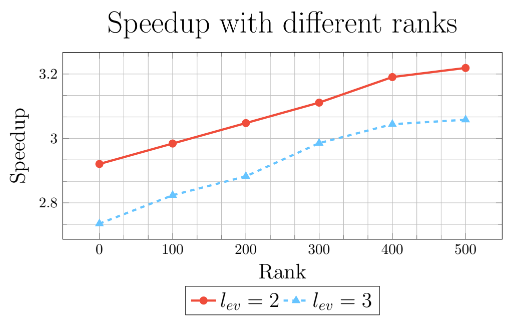


Figure 3.17: Speedup of the solution phase of `parGeMSLR` if GPU acceleration is enabled when $l_{ev} = \{2, 3\}$, and $k \in \{0, 100, 200, 300, 400, 500\}$. The problem size is equal to $n = 128^3$.

In future work, we plan to replace standard Arnoldi with either its block variant or randomized subspace iteration. This should improve performance by reducing latency during the preconditioner setup phase. Moreover, the cost of the setup phase can be amortized over the solution of linear systems with multiple right-hand sides, e.g., see [81, 82, 83, 84], and we plan to apply `parGeMSLR` to this type of problems.

3.5 Polynomial Schur Complement Low-rank Approximate Inverse

The Schur complement low-rank preconditioner discussed in the previous section is an appealing approach. However, in some situations, the rank required to achieve good convergence might be high. For example, the problem might be challenging, or we want an approximate inverse with high accuracy. In the last section of this chapter, we proposed an algorithm that uses the polynomial Schur complement approach that can achieve better accuracy as well as speed up the construction of the preconditioner for SPD matrices.

3.5.1 Schur complement approaches with low-rank correction

Under the assumption that A is SPD, we use symmetric factorization instead of non-symmetric factorization. In this section, we deduce a general formula of the Schur complement low-rank method under the assumption that the exact factorized inverse

$L_B^T L_B = B^{-1}$ of B and the approximate factorized inverse $\widetilde{L}_C^T \widetilde{L}_C \approx C^{-1}$ of C is used. This assumption is due to the fact that B can be factorized in parallel, and thus, obtaining a high-accuracy approximation is not too expensive. Note that, unlike the previous sections, we use inverse factorization, and $L^T L$ approximates the inverse of the target matrix.

Assume we have obtained L_B and $\widetilde{L}_C$. According to 3.4, we then only need to have S^{-1} in order to compute A^{-1}. When C is large, forming S^{-1} directly can be expensive and impractical. However, since the matrix-vector multiplication with S is relatively cheap, we can use polynomial approximation to compute S^{-1}. A straightforward approach is to select a polynomial $f(x) \approx 1/x$, and use $f(S)$ as an approximation for S^{-1}. Equivalently we can write the approximation using $f(x) \approx (1-x)^{-1}$ as

$$S^{-1} \approx f(I - S). \tag{3.48}$$

The spectrum radius of S can be large, thus limiting the above approximation's accuracy. We can have a more accurate polynomial approximation using a slightly more expensive formula. Writing the Schur complement as

$$S = \widetilde{L}_C^{-1}(\widetilde{L}_C S \widetilde{L}_C^T)\widetilde{L}_C^{-T} = \widetilde{L}_C^{-1}[I - \underbrace{(I - \widetilde{L}_C S \widetilde{L}_C^T)}_{H}]\widetilde{L}_C^{-T} = \widetilde{L}_C^{-1}(I - H)\widetilde{L}_C^{-T}. \tag{3.49}$$

Applying H to a vector requires two extra matrix-vector multiplications than applying $I - S$ to a vector. Nonetheless, it is still worth doing so since the eigenvalues of H can be much better clustered according to the following lemma [69].

Lemma 3.5.1. *When using the exact factorized inverse $L_C^T L_C = C^{-1}$ of C, all of the eigenvalues of $H = I - L_C S L_C^T$ are inside the interval $[0, 1)$.*

Proof. Since A is SPD, we know that S is SPD. Thus, $I - H$ is SPD.

On the other hand, since B is also SPD, $H = I - L_C S L_C^T = I - L_C(C - E^T B^{-1} E)L_C^T = L_C E^T B^{-1} E L_C^T$ is PSD. $\square$

According to the proof of the above lemma, if we replace the exact L_C with its approximation $\widetilde{L}_C$, $I - H$ is still SPD. Thus, the eigenvalues of H are still less than one. As a result, $I - H$ is always invertible, and the exact inverse of the Schur complement can be written as

$$S^{-1} = \widetilde{L}_C^T (I - H)^{-1} \widetilde{L}_C. \tag{3.50}$$

We can again use polynomial $f(x) \approx (1 - x)^{-1}$ to build the approximation

$$S^{-1} \approx \widetilde{L}_C^T f(H) \widetilde{L}_C. \tag{3.51}$$

Using the same polynomial $f(x)$, applying the above formula to a vector is slightly more expensive compared with 3.48. Nonetheless, according to the proof of 3.5.1, the matrix H is a PSD matrix plus the term $I - \widetilde{L}_C C \widetilde{L}_C^T$. As long as $\widetilde{L}_C^T \widetilde{L}_C$ is an approximation to C^{-1} with reasonable accuracy, most of the eigenvalues of $I - \widetilde{L}_C C \widetilde{L}_C^T$ are close to zero. And the eigenvalues of $I - H$ are still well clustered. Hence, it is easier to select a polynomial for H than for $I - S$, and the approximation using $f(H)$ can be more accurate if we select $f(x)$ carefully. In figure 3.18, we reorder a 3D Laplacian matrix and plot the spectrum of both H and $I - S$. All of the eigenvalues of H are between $[0, 1)$, and most of them are small. On the other hand, the eigenvalues of S are distributed more evenly within a much larger interval.

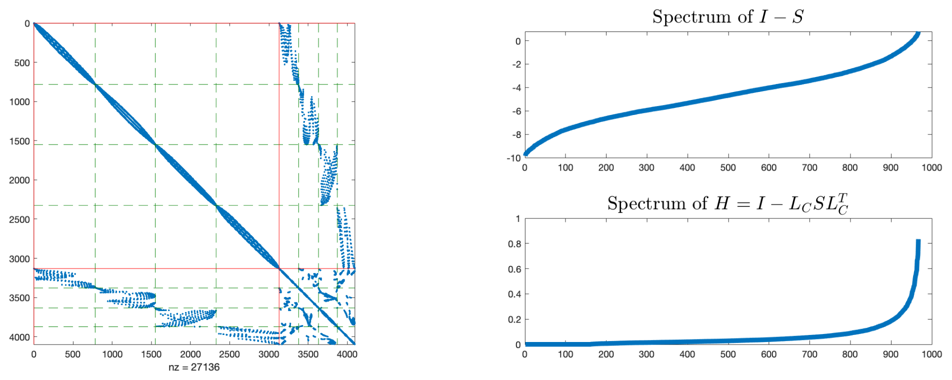


Figure 3.18: An example of a 3D Laplacian matrix. Left: The matrix is reordered into 2 by 2 block form with a block diagonal $(1, 1)$ block. Right: The eigenvalues of H are all inside the interval $[0, 1)$, and are better clustered than those of $I - S$.

We can further improve the accuracy of the polynomial approximation using a low-rank correction term thanks to the decay property of $(I - H)^{-1}$ [69, 71]. Denote by $g(H)$ the error term $(I - H)^{-1} - f(H)$. We can select a $f(x)$ such that $g(H)$ can be well approximated by a low-rank matrix. Using a partial eigendecomposition $g(H) \approx U_k g(\Lambda_k) U_k^T$ of rank k, we can improve the polynomial approximation in 3.51 to

$$S^{-1} = \widetilde{L}_C^T [f(H) + g(H)] \widetilde{L}_C \approx \widetilde{L}_C^T [f(H) + U_k g(\Lambda_k) U_k^T] \widetilde{L}_C. \tag{3.52}$$

We usually compute the partial eigendecomposition $H = U_k \Lambda_k U_k^T$ to get $g(\Lambda_k)$. If $f(x)$ is known beforehand, we can instead compute the eigenvalue decomposition of $f(H)(I - H)$ if $f(H)$ does not have zero eigenvalues. This is because

$$f(H) U_k \left([f(\Lambda_k)(I - \Lambda_k)]^{-1} - I \right) U_k^T = U_k g(\Lambda_k) U_k^T. \tag{3.53}$$

As a side note, it is generally not difficult to select a $f(x)$ such that the eigenvalues of $g(H)$ decay rapidly. Simple choices like the truncated Neumann series $f(x) = \prod_{j=0}^{d} x^j$ work reasonably well in many situations [77]. We will describe strategies for selecting $f(x)$ later.

Finally, we can use a scalar α to adjust the entire spectrum of the approximation, which makes our formula more general. We set α equal to one in our algorithm, as many other algorithms do. However, α can be other values in some cases.

We define the following *Schur complement low-rank approximate inverse matrix* as our final approximation to S^{-1}

$$\widetilde{S}^{-1}(k, \alpha, f(x)) = \alpha \left[\widetilde{L}_C^T f(H) \widetilde{L}_C + \widetilde{L}_C^T U_k g(\Lambda_k) U_k^T \widetilde{L}_C \right]. \tag{3.54}$$

For simplicity we use $\widetilde{S}^{-1}(k, \alpha, f)$ or $\widetilde{S}^{-1}$ in this paper when there is no ambiguity. We denote by $\widetilde{A}^{-1}$ the corresponding approximate inverse of the original matrix by replacing the S^{-1} in 3.4 with $\widetilde{S}^{-1}$. The following proposition discusses the relationship between the parameters and the residual's eigenvalues.

Proposition 3.5.1. *Let the eigenvalues of H be $\lambda_1, \lambda_2, \cdots, \lambda_{n_C}$, and we select the first k eigenvalues to form the partial eigendecomposition $U_k \Lambda_k U_k^T$. For real function $f(x)$ and scalar α, the eigenvalues of the residual matrix*

$$X_R = I - \widetilde{S}^{-1}(k, \alpha, f(x)) S \tag{3.55}$$

are

$$\xi_{R,i} = \begin{cases} 1-\alpha, & i \le k \\ 1-\alpha(1-\lambda_i)f(\lambda_i), & i > k \end{cases}. \tag{3.56}$$

Proof. By the definition of $\widetilde{S}^{-1}$, we can write X_R as

$$X_R = I - \widetilde{S}^{-1}S = \widetilde{L}_C^T \left(I - \alpha(f(H) + U_k g(\Lambda_k)U_k^T)(I-H)\right)\widetilde{L}_C^{-T}. \tag{3.57}$$

Thus, matrix X_R is similar to $I - \alpha(f(H) + U_k g(\Lambda_k)U_k^T)(I-H)$, which directly leads to the result. □

3.5.2 General formula for the approximate Schur complement

Next, we show that the approximation generated using many existing Schur complement low-rank methods can be written as $\hat{S}^{-1}(k,\alpha,f)$ with proper k, α and $f(x)$. It is easy to see that the standard Schur complement low-rank (SLR) method [69, 71] and its multilevel variant multilevel Schur complement low-rank (MSLR) method [70] is $\hat{S}^{-1}(k,1,1)$ or $\hat{S}^{-1}[k,1,(\lambda_{k+1}-1)^{-1}]$. The Power Schur complement low-rank (PSLR) [77] method improves SLR by using the truncated Neumann series $f(x) = \prod_{j=0}^{d} x^j$ as the polynomial. A more complicated approach is to use $\hat{S}^{-1}$-preconditioned Krylov subspace methods to solve the system $Sx = b$, which is originally used in the Generalized Multilevel Schur complement low-rank (GeMSLR) method [45, 6]. We refer to this approach as the inner iteration method. We are going to show that the resulting approximation with the inner iteration method is still a Schur complement low-rank approximate inverse matrix with the help of the following lemma.

Lemma 3.5.2. *The matrices $\hat{S}_1^{-1} + \hat{S}_2^{-1}$ and $\hat{S}_1^{-1}S\hat{S}_2^{-1}$ are Schur complement low-rank approximate inverse matrices if $\hat{S}_1^{-1}$ and $\hat{S}_2^{-1}$ are Schur complement approximate inverse matrices with same rank.*

Proof. The essential observation is that we have $f_1(H)U_k f_2(\Lambda_k)U_k^T = U_k f_1 \cdot f_2(\Lambda_k)U_k^T$ and $U_k f_1(\Lambda_k)U_k^T f_2(H) = U_k f_1 \cdot f_2(\Lambda_k)U_k^T$. With this property, it is straightforward to show that

$$\hat{S}^{-1}(k,\alpha_1,f_1)S\hat{S}^{-1}(k,\alpha_2,f_2) = \hat{S}^{-1}(k,\alpha_1\alpha_2,f_1\cdot f_2\cdot(1-x)), \tag{3.58}$$

$$\hat{S}^{-1}(k,\alpha_1,f_1)+\hat{S}^{-1}(k,\alpha_2,f_2) = \hat{S}^{-1}[k,\alpha_1+\alpha_2,(\alpha_1 f_1+\alpha_2 f_2)/(\alpha_1+\alpha_2)]. \tag{3.59}$$

□

With the above lemma, we can show the following result for inner iteration methods.

Corollary 3.5.1. *Solving $S\mathbf{x} = \mathbf{b}$ with left $\hat{S}^{-1}$-preconditioned Krylov subspace methods is equivalent to multiplying a Schur complement low-rank approximate inverse matrix with* $\mathbf{b}$.

Proof. The left preconditioned Krylov subspace methods search for the solution of $\hat{S}^{-1}S\mathbf{x} = \hat{S}^{-1}\mathbf{b}$ in the subspace spanned by $\{(\hat{S}^{-1}S)^j\hat{S}^{-1}b\}$ from $j = 0$ to some integer d.

From Lemma 3.5.2 we know that we can write $(\hat{S}^{-1}S)^j$ as $\hat{S}_j^{-1}S$ with some $\hat{S}_j^{-1}$ for any integer $j > 0$. As a result, the approximation can be written as

$$x \approx \sum_{j=0}^{d}\alpha_j(\hat{S}^{-1}S)^j(\hat{S}^{-1}\mathbf{b}) = \sum_{j=0}^{d}\hat{S}_j^{-1}\mathbf{b} = \hat{S}_{d,\mathbf{b}}^{-1}\mathbf{b} \tag{3.60}$$

with a Schur complement low-rank approximate inverse matrix $\hat{S}_{d,\mathbf{b}}^{-1}$. □

It is worth mentioning that if we choose the simplest SLR-preconditioned inner iteration with $\hat{S}^{-1}(k,1,1)$ as the preconditioner, we can see from Equation 3.58 that the polynomial of $(\hat{S}^{-1}S)^jS^{-1}$ is a polynomial of degree j for $j > 0$. Then from Equation 3.59 we know that the polynomial of $\hat{S}_{d,b}^{-1}$ is a polynomial of degree up to the dimension of the Krylov subspace.

3.5.3 Minimal residual approximation

In the previous section, we deduced a general formula of the Schur complement low-rank approximate inverse matrix. We showed that the approximation of the inverse of the Schur complement in SLR, MSLR, PSLR, and SLR-preconditioned inner iteration can all be written in the form $\hat{S}^{-1}(k, \alpha, f(x))$ with $f(x) \in \mathcal{P}_d$ for some d, where $\mathcal{P}_d$ is the set of polynomials of degree up to d.

When given a vector $\mathbf{b}$, the $\hat{S}^{-1}$ of the inner iteration method provides an approximate inverse with minimal residual in the Krylov subspace in either 2-norm or S-norm depending on the solver used. However, since the approximation varies for different $\mathbf{b}$, this approach is unsuitable for many applications. For instance, when using the corresponding $\hat{A}^{-1}$ as a preconditioner, one must use the flexible version of GMRES (FGMRES) instead of PCG or GMRES. Another issue is that we can not know $f(x)$ in advance. So, we can not accelerate the computation of partial eigendecomposition using $f(H)(1-H)$. Thus, we would like to find a fixed polynomial $f(x)$ that is near optimal.

Let the eigenvalues of H be in descending order $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{n_C}$, and we select the first k eigenvalues with the largest magnitude to form the partial eigendecomposition. In order to select a good polynomial $f(x)$ and scalar α given a degree d, we can use the result in Theorem 3.5.1.

In this section, we discuss the polynomial that minimizes the 2-norm of X_R, which is equivalent to minimizing $\max_i |\xi_{R,i}|$. According to the proposition, the selection of α only influences the first k eigenvalues. To minimize the 2-norm we can simply fix $\alpha = 1$ and select the polynomial $f(x)$ via

$$arg\min_{f \in \mathcal{P}_d} \left[\max |(1-(1-\lambda_i)f(\lambda_i))| \, \middle| i \in \{k+1, k+2, \dots, n\} \right]. \tag{3.61}$$

However, computing all eigenvalues of H is impractical, and instead, we solve the continuous version of it. Denote by $\mathcal{G}_d$ the subspace of polynomials that can be written as $(1-x)f(x)$ with $f(x) \in \mathcal{P}_d$, the continuous minimax problem is defined as

$$arg\min_{\hat{f} \in \mathcal{G}_d} \left|\left| 1 - \hat{f}(x) \right|\right|_\infty, \quad x \in [\lambda_{n_C}, \lambda_{k+1}]. \tag{3.62}$$

After obtaining $\hat{f}(x)$, we can immediately have $f(x) = (1-x)^{-1}\hat{f}(x)$. This continuous problem is easier to solve since we only need to estimate λ_{k+1} and λ_{n_C}.

If $\lambda_{n_C} = \lambda_{k+1}$, the solution is the zero degree polynomial $f(x) = (1-\lambda_{k+1})^{-1}$. We assume $\lambda_{k+1} > \lambda_{n_C}$ in the following discussion. To find the best uniform approximation of $h(x) = 1$ in $\mathcal{G}_d$ within $[\lambda_{n_C}, \lambda_{k+1}]$, we search for the approximation using the Remez exchange algorithm [85, 86, 87]. The idea behind the exchange algorithm is that the error function of the best uniform approximation takes extrema exactly $d+2$ times with alternate signs in the target region if we search within a subspace of dimension d that does not contain the target function. The subspace should satisfy Haar condition [88, 89], which we will discuss later. The use of exchange algorithm requires $d+2$ initial points $x_1, x_2, \ldots, x_{d+2}$ in $[\lambda_{n_C}, \lambda_{k+1}]$. During each iteration, the algorithm searches for a polynomial in $\mathcal{G}_d$ such that the error function takes the same absolute value at those $d+2$ points with alternate signs. Since the error function intersects with the x axis exactly $d+1$ times in $(\lambda_{n_C}, \lambda_{k+1})$, the interval $[\lambda_{n_C}, \lambda_{k+1}]$ is partitioned into $d+2$ segments. The $d+2$ local extrema points of the current error function within each segment are then used as new points to start the next iteration. This procedure is repeated until the error is satisfied or a maximum number is reached.

We choose to use $\{g_j(x) := (1-x)x^j | i = 0, 1, \ldots, d\}$ as the basis functions for $\mathcal{G}_d$, and solve $\sum_{s=0}^{d} a_s(1-x_j)x_j^s + (-1)^j E = 1$ for $j = 1, \ldots, d+2$ to get the approximation during each iteration. We select Chebyshev nodes as our initial points. The existence and uniqueness of the solution, as well as the convergence of the exchange algorithm, are discussed in the following proposition.

Proposition 3.5.2. *There exists an unique solution to Equation 3.62 in $\mathcal{G}_d$. Besides, exchange algorithm converges to the solution with any initial points $\lambda_{n_C} \le x_1 < x_2 < \cdots < x_{d+2} \le \lambda_{k+1}$.*

Proof. Since $\mathcal{G}_d$ is a subset of $\mathcal{P}_{d+1}$, zero polynomial is the only function in $\mathcal{G}_d$ that has more than $d+1$ roots in $[\lambda_{n_C}, \lambda_{k+1}]$ given the fact that 1 is always a root of functions in $\mathcal{G}_d$ and $\lambda_{k+1} < 1$. This implies that $\mathcal{G}_d$ satisfied the Haar condition.

According to the Theorem 9.3 of [89], there is a unique solution to the problem with those two bases, and the exchange algorithm converges with any valid initial points. $\square$

Finally, we estimate the error bound of the residual and error of the Remez-based approximation and compare the results with the PSLR algorithm. We first compute an error bound for the PSLR algorithm.

Lemma 3.5.3. *Assume that the approximation of B^{-1} and C^{-1} are exact. When using the approximation $\hat{S}(k, 1, \sum_{j=0}^{d} x^j)$, the 2-norm of the residual matrix X_R defined in*

Equation 3.55 is

$$||X_R||_2 = \lambda_{k+1}^{d+1}, \tag{3.63}$$

Proof. We have showed that the eigenvalues λ_j of H obeys $0 \le \lambda_j < 1$. Thus, $\lambda_{n_C} \ge 0$. Since $1-(1-x)f(x) = x^{d+1}$, we know that the maximal is λ_{k+1}^{d+1} when $x = \lambda_{k+1}$. □

Next, we compare the bound of PSLR with our Remez-based approximation.

Theorem 3.5.4. *Assume that the approximation of B^{-1} and C^{-1} are exact and $\lambda_{n_C} = 0$. When using the approximation $\hat{S}(k, 1, f(x))$ where $(1-x)f(x)$ is the best approximation to 1 in $\mathcal{G}_d$, the 2-norm of the residual matrix X_R defined in Equation 3.55 is bounded by*

$$||X_R||_2 \le \min\left(\frac{\beta_1}{\beta_2^d}, 1\right)\lambda_{k+1}^{d+1}, \tag{3.64}$$

where β_1 and $\beta_2 \ge 1$ are constants depends only on λ_{k+1}.

Proof. Since the selected polynomial is the best approximation to 1 in $\mathcal{G}_d$, we immediately have $||X_R||_2 \le \lambda_{k+1}^{d+1}$ according to Theorem 3.5.3.

On the other hand, the best approximating of $\frac{1}{1-x}$ in $\mathcal{P}_d$ also gives an approximation to 1 in $\mathcal{G}_d$. Although this polynomial is not optimal, it still gives a good bound when λ_{k+1} is small. By changing of variables we can write $1/(1-x)$ in $[0, \lambda_{k+1}]$ as

$$\frac{2}{\lambda_{k+1}} \frac{1}{y - \left(\frac{2}{\lambda_{k+1}} - 1\right)}, \quad y \in [-1, 1], \tag{3.65}$$

and $\frac{2}{\lambda_{k+1}} - 1 > 1$. According to the bound in Theorem 2.1 of [90] we have

$$||X_R||_2 \le \min_{f\in\mathcal{P}_d} ||(1-(1-x)f(x))||_\infty \le \left|\left|\frac{1}{1-x} - f(x)\right|\right|_\infty \le \frac{1}{2t^2(1+t)^{2d}}\lambda_{k+1}^{d+1}, \tag{3.66}$$

where $t = (1-\lambda_{k+1})^{1/2} \le 1$. We can choose $\beta_1 = t^{-2}/2$ and $\beta_2 = (1+t)^{-2}$ to get the result. □

It is worth mentioning that in practice, β_1/β_2^d is usually strictly less than one. The above theorem implies that the residual 2-norm of the Remez-based algorithm is smaller than the residual 2-norm of the PSLR algorithm with an exponential decay with respect to d. In Figure 3.19, we compute the improvement of the Remez-based algorithm over PSLR numerically with different λ_{k+1} and polynomial degree. We can indeed see exponential growth with respect to d, even when λ_{k+1} is close to one.

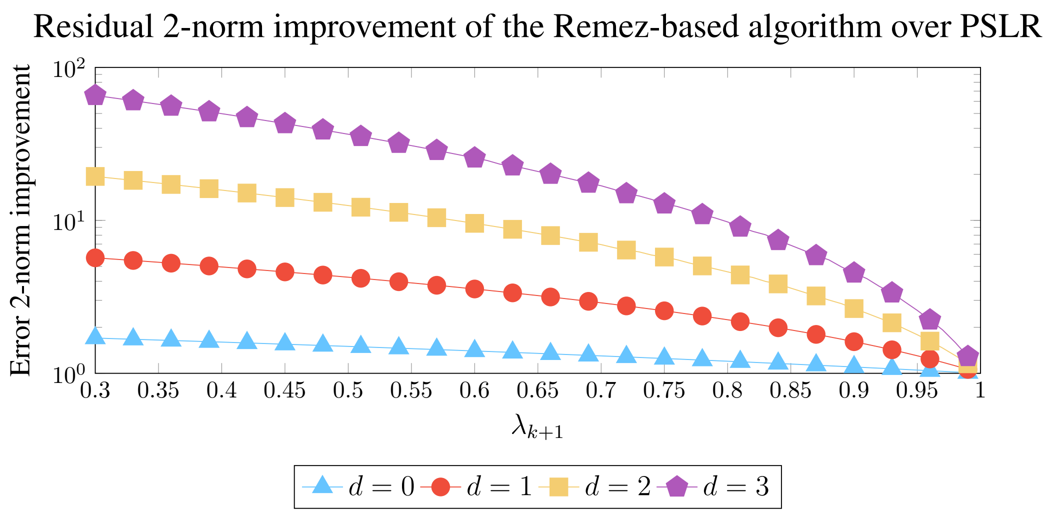


Figure 3.19: Improvement of the residual 2-norm $||X_R||_2$ of the Remez-based algorithm over that of the PSLR with different polynomial degrees with λ_{k+1} from 0.3 to 0.99. When fixing λ_{k+1}, the improvement increase exponentially.

3.5.4 Hermite-Remez approximation

In the previous section, we select the best uniform approximation of $h(x) = 1$ from $\mathcal{G}_d$ to minimize the 2-norm of the residual. The Remez-based approximation does not take the decay property of $(I - H)^{-1}$ into consideration. In practice, many eigenvalues of H tend to be close to zero. However, the error function $1 - \hat{f}(x)$ might have extrema near zero. For instance, if we use the exact L_C, λ_{n_C} is very close to zero if $(I - H)^{-1}$ decays rapidly. The error function $|1 - \hat{f}(x)|$ then would likely have extrema near zero if we obtain $\hat{f}(x)$ using the exchange algorithm. In practice, λ_{n_C} is usually not close to zero when using the approximation $\hat{L}_C$ instead of L_C. However, the error function might still reach its extrema near the origin. The consequence is that many eigenvalues of the residual matrix are relatively large, although the 2-norm of the residual matrix is minimized.

In general, if the eigenvalues of H have several clusters, we might prefer to use polynomials that lead to smaller errors near those clusters. One simple way to achieve this goal is to select a polynomial such that the error function and up to certain first derivatives are zero at some points within those clusters. In this section, we generalize our Remez-based algorithm by combining it with Hermite interpolation. The proposed Hermite-Remez approximation allows us to control the value of the approximation error $|1-\hat{f}(x)|$ near the origin at the cost of slightly increasing the 2-norm of the residual.

Ideally, we should estimate the Density of States (DOS) or Spectral Density of H to find several clusters of eigenvalues. If we find l clusters $y_0, y_1, \ldots, y_{l-1}$, we then select l corresponding integers $d_0, d_1, \ldots, d_{l-1}$ at each point, and build the interpolation of $h(x) = 1$ in the form $\hat{f}_H(x)$ with up to $d_j - 1$ first derivatives at those l interpolation points in $\mathcal{G}$. If $d_j \leq 0$ we ignore the point y_j in the interpolation. The values d_j can be selected based on the number of points within each cluster. However, it is typically enough to simply assume that zero is the only cluster of the eigenvalues of H due to the decay property of the eigenvalues of $(I-H)^{-1}$. We set $l = 1$ and pick $y_0 = 0$ as the only point with order d_0.

Assume $d_0 > 0$. Denote by $\mathcal{G}_{d/d_0}$ the subspace spanned by $\{(1-x)x^{j+d_0}\}$ with j from 0 to $d - d_0$ for $d_0 > 0$. We can verify that the functions in $\mathcal{G}_d$ such that the value and up to $d_0 - 1$ first derivative matches those of $h(x) = 1$ at $y_0 = 0$ take the form $(1-x)\sum_{j=0}^{d_0-1} x^j + \hat{f}(x)$ with $\hat{f}(x) \in \mathcal{G}_{d/d_0}$. We can write our minimax problem for the Hermite-Remez approach as

$$arg\min_{\hat{f}\in\mathcal{G}_{d/d_0}} \left|\left|x^{d_0} - \hat{f}(x)\right|\right|_\infty, \quad x \in [\lambda_{n_C}, \lambda_{k+1}], \tag{3.67}$$

with final approximation in the form $\sum_{j=0}^{d_0-1} x^j + (1-x)^{-1}\hat{f}(x)$.

However, the solution to the above problem might not be unique since $\mathcal{G}_{d/d_0}$ does not satisfy the Haar condition when $\lambda_{n_C} \leq 0$ given that 0 is always a root of polynomials in $\mathcal{G}_{d/d_0}$. We can find a nonzero polynomial from $\mathcal{G}_{d/d_0}$ with $d - d_0 + 1$ roots in $[\lambda_{n_C}, \lambda_{k+1}]$. The uniqueness of the solution is guaranteed if we slightly loosen the constraints. We can choose a function $w(x) \in C[\lambda_{n_C}, \lambda_{k+1}]$ such that $w(x) \neq 0$ for $x \in [\lambda_{n_C}, \lambda_{k+1}]$. Since $x^{d_0} - \hat{f}(x) = x^{d_0}w^{-1}(x)[w(x) - w(x)x^{-d_0}\hat{f}(x)]$, we can then solve

$$arg\min_{f\in\mathcal{P}_{d-d_0}} ||w(x) - (1-x)w(x)f(x)||_\infty, \quad x \in [\lambda_{n_C}, \lambda_{k+1}]. \tag{3.68}$$

The final approximation is then given by $\sum_{j=0}^{d_0-1} x^j + x^{d_0}f(x)$. The error function is then in the form

$$\frac{x^{d_0}}{w(x)}[w(x)-(1-x)w(x)f(x)]. \tag{3.69}$$

The solution to the above problem is unique, and the convergence is guaranteed with any proper initial points.

Proposition 3.5.3. *There exists an unique solution to Equation 3.68 in $\mathcal{P}_{d-d_0}$. Besides, exchange algorithm converges to the solution with any initial points $\lambda_{n_C} \le x_{d_0} < x_{d_0+1} < \cdots < x_{d+2} \le \lambda_{k+1}$.*

Proof. Solving Equation 3.68 is equivalent to searching for the best approximation of $w(x)$ in the subspace spanned by $\{(1-x)w(x)x^j\}$ with j from 0 to $d-d_0$. Since $w(x) \neq 0$ for $x \in [\lambda_{n_C}, \lambda_{k+1}]$, we know that this subspace satisfies the Haar condition, and thus we have the conclusion. $\square$

The remaining task is to select a good $w(x)$. We use $w_\epsilon(x) := |x^{d_0}| + \epsilon$ where ϵ is a small positive scalar. The intuition behind this selection is that $|w_\epsilon(x)|$ is a good approximation to $|x^{d_0}|$ and thus $|x^{d_0}/w_\epsilon(x)| \approx 1$. Next, we show that if we select ϵ properly, Equation 3.68 is indeed a good approximation to Equation 3.67.

Proposition 3.5.4. *If we select $w(x) = w_\epsilon(x) = |x^{d_0}| + \epsilon$ with $\epsilon > 0$, we have*

$$\min_{f\in\mathcal{P}_{d-d_0}} \left|\left|w(x)-(1-x)w(x)f(x)\right|\right|_\infty \le (1+\epsilon\beta) \min_{\hat{f}\in\mathcal{G}_{d/d_0}} \left|\left|x^{d_0}-\hat{f}(x)\right|\right|_\infty, \tag{3.70}$$

where β is a constant depends only on d_0, λ_{n_C}, and λ_{k+1}. Besides, if $f^\star(x)$ is the solution of Equation 3.68, then

$$\left|\left|x^{d_0}-(1-x)x^{d_0}f^\star(x)\right|\right|_\infty \le (1+\epsilon\beta)\left|\left|w_\epsilon-(1-x)w_\epsilon f^\star(x)\right|\right|_\infty. \tag{3.71}$$

Proof. We first prove the first inequality. We can see that that $\hat{f}(x) \mapsto x^{-d_0}(1-x)^{-1}\hat{f}(x)$

is a bijection between $\mathcal{G}_{d/d_0}$ and $\mathcal{P}_{d-d_0}$. For any function $\hat{f}(x) \in \mathcal{G}_{d/d_0}$ we have

$$\begin{aligned}
&(1 + \epsilon \left|\left|x^{d_0}\right|\right|_{\infty}^{-1}) \left|\left|x^{d_0} - \hat{f}(x)\right|\right|_{\infty} \\
&= \left|\left|x^{d_0} - (1-x)x^{d_0} f(x)\right|\right|_{\infty} + \epsilon \left|\left|x^{d_0}\right|\right|_{\infty}^{-1} \left|\left|x^{d_0} - (1-x)x^{d_0} f(x)\right|\right|_{\infty} \\
&\geq \left|\left||x^{d_0}| - (1-x)|x^{d_0}| f(x)\right|\right|_{\infty} + \epsilon \left|\left|1 - (1-x)f(x)\right|\right|_{\infty} \\
&\geq \left|\left|w(x) - (1-x)w(x)f(x)\right|\right|_{\infty}.
\end{aligned}$$

We can then show the first inequality by setting $\beta = \left|\left|x^{d_0}\right|\right|_{\infty}^{-1}$.

The proof of the second inequality is similar. We can write

$$\begin{aligned}
&\left|\left|x^{d_0} - (1-x)x^{d_0} f(x)\right|\right|_{\infty} = \left|\left||x^{d_0}| - (1-x)|x^{d_0}| f(x)\right|\right|_{\infty} \\
&= \left|\left||x^{d_0}| + \epsilon - (1-x)(|x^{d_0}| + \epsilon) f(x) - \epsilon[1 - (1-x)f(x)]\right|\right|_{\infty} \\
&\leq \left|\left|w_\epsilon - (1-x)w_\epsilon f(x)\right|\right|_{\infty} + \epsilon \left|\left|1 - (1-x)f(x)\right|\right|_{\infty} \\
&\leq \left|\left|w_\epsilon - (1-x)w_\epsilon f(x)\right|\right|_{\infty} + \epsilon \left|\left|w_\epsilon\right|\right|_{\infty}^{-1} \left|\left|w_\epsilon - (1-x)w_\epsilon f(x)\right|\right|_{\infty} \\
&\leq (1 + \epsilon \left|\left|w_\epsilon\right|\right|_{\infty}^{-1}) \left|\left|w_\epsilon - (1-x)w_\epsilon f(x)\right|\right|_{\infty}.
\end{aligned}$$

Since $\left|\left|w_\epsilon\right|\right|_{\infty}^{-1} = (\left|\left|x^{d_0}\right|\right|_{\infty} + \epsilon)^{-1} < \left|\left|x^{d_0}\right|\right|_{\infty}^{-1} = \beta$, we immediately have the result. □

The above proposition indicates that we can use the optimal solution of Equation 3.68 as approximate optimal solutions of Equation 3.67 by setting $w(x) = w_\epsilon(x)$. In fact, if we solve a sequence of problems with $\epsilon \to 0$, the solution will converge to one of the optimal solutions to Equation 3.67. In practice, if the ϵ is too small, the linear systems involved in the exchange algorithm might become very ill-conditioned with some point sets. Thus, we choose ϵ to be a moderate value 10^{-5} as the default value. In Figure 3.20 we compare the error functions of different weight functions with $\lambda_{n_C} = -0.5$, $\lambda_{k+1} = 0.7$, and $d_0 = 2$ using $f(x) \in \mathcal{P}_3$. The Remez-based algorithm with $d_0 = 0$ has the smallest maximum absolute error. As we can see, simply using $w(x) = 1$ leads to a large error at one end of the interval. Using weight function $w_\epsilon(x)$ can resolve this issue as we expected. The ϵ need not be too small for the weight function to be effective.

In Figure 3.21, we compare the performance on the same problem using different

d_0. The Remez-based algorithm with $d_0 = 0$ has the smallest maximum absolute error while having a large error at around zero. The PSLR algorithm with $d_0 = 4$ has a small absolute value over a large region centered at the origin while having larger errors in some other regions. As we increase d_0 from 0 to 4, the largest error increases, but the value around zero gets smaller. The selection of $d_0 = 1$ and $d_0 = 2$ fits our needs well. The largest errors are only slightly increased, but the error at the origin is reduced significantly.

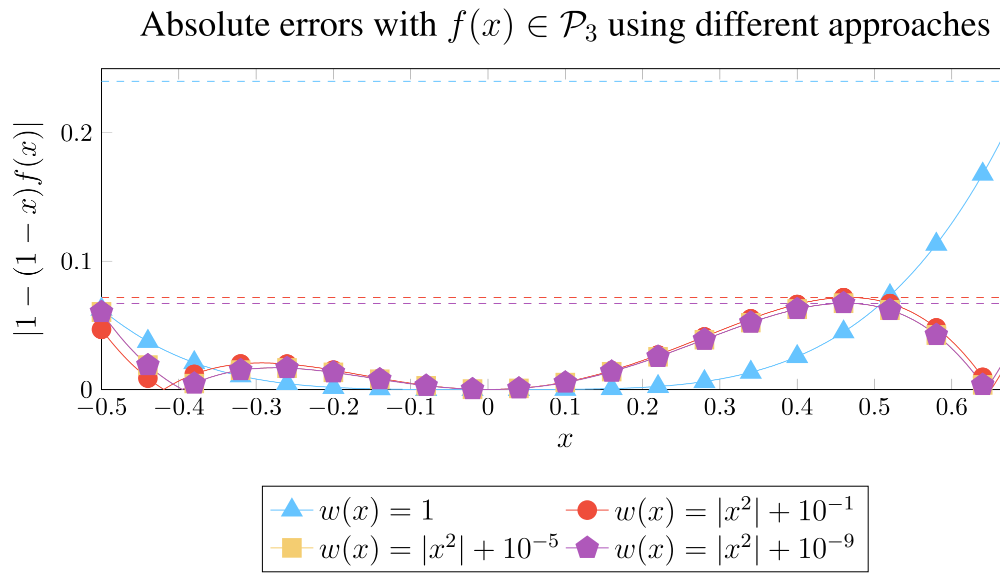


Figure 3.20: Absolute approximation error $|1-(1-x)f(x)|$ in $[-0.5, 0.7]$ with $f(x) \in \mathcal{P}_3$ using different methods. The maximum absolute error of different methods is marked with thin dashed lines. The Hermite-Remez approximation with $d_0 = 2$ and $w_\epsilon(x) = x^2$ have a small maximum error. The three ϵ values we tested lead to a similar error function, and the error value near zero is small.

The Hermite-Remez algorithm is summarized in Algorithm 20, and the final approximation takes the form $f(x) = \sum_{j=0}^{d} a_j x^j$. Steps 15-17 of the algorithm are used to guarantee that the final approximation $\hat{S}^{-1}$ is SPD. Note that the PSLR algorithm and the Remez-based algorithm can be seen as special cases of the Hermite-Remez algorithm. When $d_0 = 0$, the Hermite-Remez algorithm is equivalent to the Remez-based algorithm discussed in the previous section. On the other hand, when $d_0 = d + 1$, we obtain the PSLR algorithm.

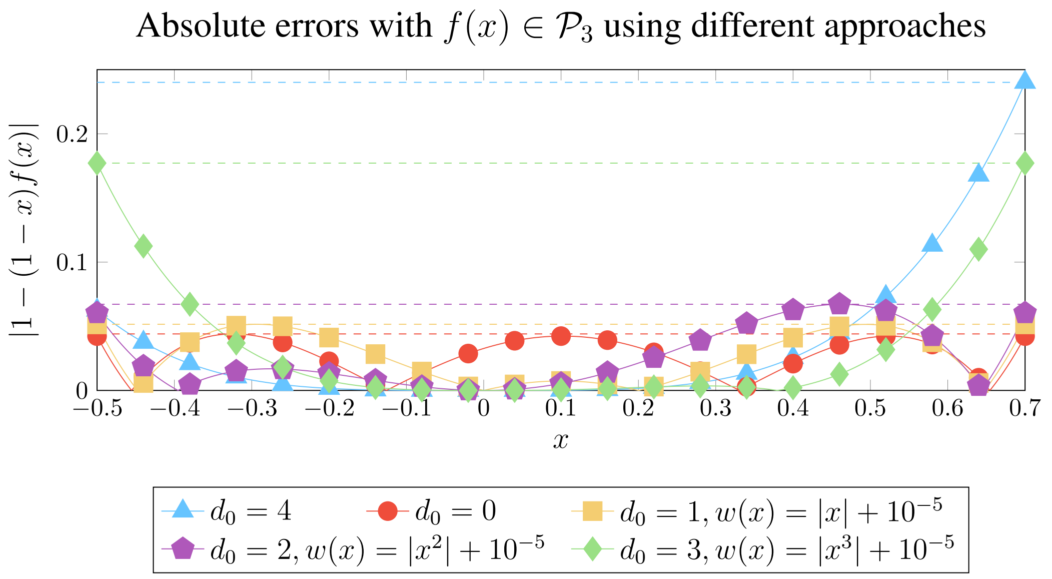


Figure 3.21: Absolute approximation error $|1-(1-x)f(x)|$ in $[-0.5, 0.7]$ with $f(x) \in \mathcal{P}_3$ using different methods. The maximum absolute error of different methods is marked with thin dashed lines. The Hermite-Remez approximation with $d_0 = 2$ and $w(x) = x^2 + 10^{-5}$ has a small maximum error, and the error value near zero is small.

Algorithm 20 Hermite-Remez Approximation

```
 1: if $\lambda_{n_C} = \lambda_{k+1}$ then return $\mathbf{a} = [1/(1-\lambda_{n_C}), 0, \ldots, 0]$.
 2: end if
 3: $d_1 \leftarrow d - d_0$
 4: Pick $d_1 + 2$ initial points $x_1 < x_2 < \ldots < x_{d_1+2}$ within $[\lambda_{n_C}, \lambda_{k+1}]$.
 5: while Not converge do
 6:     Solve $\sum_{j=0}^{d-d0} a_i(1-x)w(x)x^j + (-1)^j\gamma = 1$.
 7:     Find $d_1 + 2$ new local optimal.
 8:     Find new local extremas.
 9:     if Converged then
10:         Break.
11:     else
12:         Use local extremas as new $x_i$.
13:     end if
14: end while
15: if $\min_{x\in[\lambda_{n_C},\lambda_{k+1}]} f(x) \le 0$ then
16:     Increase $d$ and restart algorithm.
17: end if
18: return $\mathbf{a} = [1, \ldots, 1, a_{d_0}, \ldots, a_d]$.
```

3.5.5 Multilevel approximate inverse algorithm

Next, we discuss the construction of the partial eigendecomposition of H, or $f(H)(I-H)$ if $f(x)$ is selected beforehand. Without loss of generality, we only discuss the decomposition of H. We need to obtain several eigenvalues of H with the largest magnitude by solving the following eigenvalue problem

$$H\mathbf{v} = (I - \hat{L}_C S \hat{L}_C^T)\mathbf{v} = \lambda \mathbf{v}. \tag{3.72}$$

Lanczos algorithm or random Nyström approximation are good options when matrix-vector multiplication with $\hat{L}_C$ and $\hat{L}_C^T$ are explicitly available. However, the solution to the above problem is not trivial if we want to extend our algorithm into a multilevel approach.

Recall that the approximate inverse of the Schur complement in our approximation is in the form Equation 3.52. If we use our algorithm recursively on C, the approximation to C^{-1} might not be written in the form $\hat{L}_C^T\hat{L}_C$. We can bypass this issue by slightly modifying Equation 3.72.

Denote by $\hat{C}^{-1}$ the approximation $\hat{L}_C^T\hat{L}_C$. Multiplying $\hat{C}\hat{L}_C^T$ from left on both sides, we can transfer the above eigenvalue problem into the following symmetric generalized eigenvalue problem

$$(\hat{C} - S)\mathbf{w} = \hat{C}\lambda\mathbf{w}. \tag{3.73}$$

The eigenvalues of Equation 3.72 and Equation 3.73 are the same. However, applying the Lanczos algorithm to Equation 3.73 computes the eigenvectors in the form $w = \hat{L}_C^T v$, where v has a norm equal to one. Thus, solving Equation 3.73 gives Λ_k and $W_k = \hat{L}_C^T U_k$ where $U_k^T \Lambda_k U_k^T$ is a partial eigendecomposition of H. We can then rewrite the approximation of S^{-1} in Equation 3.54 as

$$\hat{S}^{-1}(k, \alpha, f(x)) = \alpha \left[f(I - \hat{C}^{-1}S)\hat{C}^{-1} + W_k g(\Lambda_k) W_k^T \right]. \tag{3.74}$$

This is because for any integer $i \geq 0$, $\hat{L}_C^T H^i \hat{L}_C = \hat{L}_C^T(I - \hat{L}_C S \hat{L}_C^T)^i \hat{L}_C = \hat{L}_C^T(\hat{L}_C(\hat{C} - S)\hat{L}_C^T)^i \hat{L}_C = (I - \hat{C}^{-1}S)^i \hat{C}^{-1}$.

Although we do not have $\hat{C}$ available, solving Equation 3.73 does not require applying $\hat{C}$ if we use Lanczos algorithm with random initial vector [22]. We use the thick-restart

Lanczos algorithm to compute the partial eigendecomposition in our implementation. We also use the thick-restart Lanczos algorithm to estimate λ_{n_C} and λ_{k+1} by restarting only with the left-most and the right-most eigenvalues. This approach is known to be very robust in practice [91].

With all the components available, we can easily go multilevel by recursively applying this algorithm to the C block and replacing L_B with an approximation $\hat{L}_B$. The Schur complement is now the approximation $\hat{S} = C - E^T\hat{L}_B^T\hat{L}_B E$. We summarize the setup phase of the approximate inverse in Algorithm 21. The approximate inverse can be applied to a vector following Algorithm 22.

Algorithm 21 Hermite-Remez Approximate Inverse Setup

```
1: Factorize $B^{-1} \approx \hat{L}_B^T\hat{L}_B$.
2: Construct $\hat{C}^{-1} \approx C^{-1}$.
3: Solve eigenvalue problem $(\hat{C} - \hat{S})W_k = \hat{C}W_k\Lambda_k$.
4: Estimate $\lambda_{n_C}$ and $\lambda_{k+1}$.
5: Compute $\mathbf{a}$=HermiteRemez$(d, d_0, w(x), \lambda_{k+1}, \lambda_{n_C})$.
6: **return** $L_B, \hat{C}^{-1}, W_k, \Lambda_k, \mathbf{a}$
```

Algorithm 22 Hermite-Remez Approximate Inverse Solve

```
1: Split $x = [x_u^T, x_l^T]^T$
2: Compute $z_u = \hat{L}_B^T\hat{L}_B x_u$
3: Compute $z_l = x_l - E^T z_u$
4: Compute $y_l = f(I - \hat{C}^{-1}\hat{S})\hat{C}^{-1}z_l + W_k g(\Lambda_k)W_k^T z_l$ where $f(x) = \sum_{j=0}^{d} a_j x^j$.
5: Compute $y_u = z_u - \hat{L}_B^T\hat{L}_B E y_u$
6: **return** $y = [y_u^T, y_l^T]^T$
```

As an alternative to Algorithm 21, we can estimate the DOS of H, and choose an interval $[a, b]$ without knowing the exact k value. We can then compute the partial eigen-decomposition of $f(H)$ with $f(x)$ computed using $[a, b]$ as $[\lambda_{n_C}, \lambda_{k+1}]$. This approach is similar to applying polynomial filtering, which can potentially accelerate convergence.

3.5.6 Factorized Remez approximation

In certain applications, the factorized approximated inverse in the form $\hat{G}^T\hat{G} \approx A^{-1}$ is wanted. According to Equation 3.4, this can be done by building the approximation of S^{-1} in factorized form. We can build the approximation to $(I-H)^{-1/2}$ instead of $(I-H)^{-1}$ in the form

$$f(H) + [(I-H)^{-1/2} - f(H)] \approx \alpha \left(f(H) + U_k[(I_k - \Lambda_k)^{1/2} - f(\Lambda_k)]U_k^T \right). \tag{3.75}$$

The approximation of the inverse of Schur complement can then be written in a factorized formula as

$$\hat{S}^{-1} = \alpha^2 \left(f(H)\hat{L}_C + U_k h(\Lambda_k) U_k^T \hat{L}_C \right)^T \left(f(H)\hat{L}_C + U_k h(\Lambda_k) U_k^T \hat{L}_C \right). \tag{3.76}$$

This variant can be easily generalized into a multilevel version since $\hat{L}_C$ is always available. We can again use the Hermite-Remez algorithm to compute the approximation.

We end this section with a comparison between different approximation methods with a polynomial of degree 4. We choose an even number so that the Factorized-Remez approach works.

We reorder a 3D Laplacian matrix of size $n = 1000$ into the above 2 by 2 block form and use the inverse of block Jacobi incomplete Cholesky factorization as the approximation to C^{-1}. Figure 3.22 shows the eigenvalues of the residual matrix X_R with different approaches. Using the same polynomial degree, the Factorized formula provides a similar residual 2-norm compared with PSLR. The eigenvalues of the residual matrix from the Hermite-Remez algorithm with $d_0 = 2$ have a much smaller spectrum radius. Decreasing d_0 from 2 to 0 slightly reduce the 2-norm of X_R, but the eigenvalues are no longer clustered at zero.

3.5.7 Improving FSAI with low-rank correction

According to the previous section, we can identify several key components that influence the accuracy of the output of Algorithm 22 most including the rank k, the degree of polynomial d, and the accuracy of $\hat{L}_B$ and $\hat{L}_C$. In this section, we describe strategies for obtaining an accurate approximation to L_B. An important observation is that an accurate approximation $\hat{L}_B$ to L_B should generally be dense. Similar to the multilevel approaches used earlier, we again combine sparse approximation with dense approximation to form the local approximation to $\hat{L}_B$ that is dense. The low-rank correction can

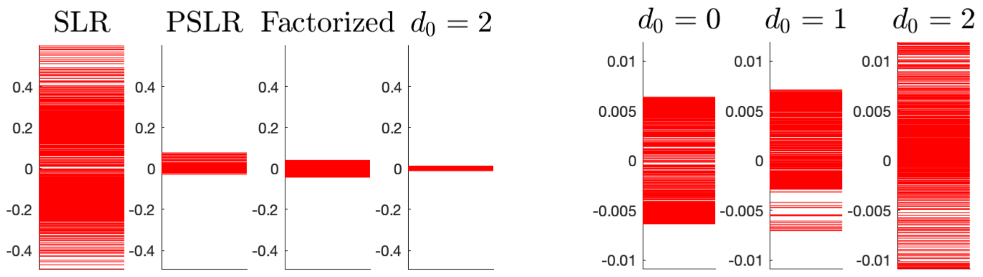


Figure 3.22: Spectrum of the residual matrix X_R from a 3D Laplacian of size $n = 1000$, $d = 4$, and $k = 10$. The size of C is $n_C = 502$, and we use the block Jacobi approach to obtain $\hat{L}_C$. We compare SLR, PSLR, and Hermite-Remez with $d_0 = 2$, and factorized Remez in the left figure. We compare Hermite-Remez with different d_0 in the right figure. Note that the two groups of figures scale differently.

be either added before or after the FSAI. In the paper, we call the former approach the *anterior correction* and the later approach the *posterior correction.*

Low-rank approximation of square-root inverse

We first describe the formula of our low-rank correction. In order to make the approximation symmetric, our low-rank correction takes the following form

$$(I - ZZ^T)A(I - ZZ^T) \approx I. \tag{3.77}$$

Ideally, the largest eigenvalues of $(I - ZZ^T)^2$ should match those of A^{-1}. If $AQ_1 = Q_1D$ where Q_1 has k orthonormal columns consisting of eigenvectors of A associated with the smallest k eigenvalues $\lambda_1, \ldots, \lambda_k$ and $D = diag(\lambda_i)$. We seek Z in the form $Z = Q_1S$ where S is diagonal and then

$$I - (I - ZZ^T)A(I - ZZ^T) = I - AQ_1SS^2A^T + Q_1S^2Q_1^TA - Q_1S^4Q_1^T \tag{3.78}$$

$$= I - A + Q_1[DS^2 + S^2D - S^2DS^2]Q_1^T, \tag{3.79}$$

which is minimized in the 2-norm sense when $I - D + 2DS^2 - DS^4 = 0$. Thus, when $S = diag(s_i)$ and $t_i \equiv s_i^2$ then $t_i^2 - 2t_i - (1 - \lambda_i)/\lambda_i = 0$ must be satisfied, which leads to

$$s_i = \sqrt{1 - \frac{1}{\sqrt{\lambda_i}}}. \tag{3.80}$$

Algorithm 23 Low-rank approximation of square-root inverse

```
1: Compute $AQ_k = Q_k D_k$
2: Compute $S$
3: **return** $Z = Q_k S$
```

Anterior correction

The reason for using the anterior correction is that after adding the low-rank correction terms, there are fewer large entries in A^{-1}. In this section, we discuss the anterior correction where the final approximation takes the form $(I - Z_k Z_k^T)\hat{U}\hat{U}^T(I - Z_k Z_k^T) \approx A^{-1}$. We focus on showing that adding low-rank correction could reduce the number of large entries in A^{-1}. However, it is generally difficult to apply the spectral analysis of A^{-1} for general matrices. Here, we only analyze A generated from the discretized Laplacian problem on regular grids. Considering the discretized Laplacian problem on regular grids of size $n_x \times n_y \times n_z$ with Dirichlet boundary condition and centered differences. We can write the eigenvalues of A as

$$\lambda_{\mathbf{i}} = 4\sin\left(\frac{\pi j_x}{2(n_x+1)}\right)^2 + 4\sin\left(\frac{\pi j_y}{2(n_y+1)}\right)^2 + 4\sin\left(\frac{\pi j_z}{2(n_z+1)}\right)^2, \tag{3.81}$$

and the eigenvectors of A as

$$v_{\mathbf{i},\mathbf{j}} = \sqrt{\frac{2}{n_x+1}}\sin\left(\frac{i_x j_x \pi}{n_x+1}\right)\sqrt{\frac{2}{n_y+1}}\sin\left(\frac{i_y j_y \pi}{n_y+1}\right)\sqrt{\frac{2}{n_z+1}}\sin\left(\frac{i_z j_z \pi}{n_z+1}\right), \tag{3.82}$$

where $\mathbf{i} = (i_x, i_y, i_z)$ and $\mathbf{j} = (j_x, j_y, j_z)$. Denote the eigendecomposition of A by $A = V\Lambda V^{-1}$.

For 1D problems, we can write the (i, j) entry of the inverse of the corrected matrix

$A_k = (I - Z_k Z_k^T)A(I - Z_k Z_k^T)$ as

$$(A_k^{-1})_{i,j} = \frac{1}{2(n+1)} \sum_{l=1}^{k} \sin\left(\frac{il\pi}{n+1}\right) \sin\left(\frac{lj\pi}{n+1}\right) + \sum_{l=k+1}^{n} \frac{\sin\left(\frac{il\pi}{n+1}\right) \sin\left(\frac{lj\pi}{n+1}\right)}{\sin\left(\frac{\pi l}{2(n+1)}\right)^2}. \quad (3.83)$$

Instead of directly studying the series above, we consider the continuous version of Equation 3.83. We map $\frac{i\pi}{n+i}$, $i = 1, \ldots, n$ to $\theta \in (0, \pi)$ and denote by x the θ value corresponding to k:

$$f_{i,j}(x) = C_{i,j} + \frac{1}{2\pi} \int_{\theta=x}^{\pi} \frac{cos^2 \frac{\theta}{2}}{sin^2 \frac{\theta}{2}} sin(i\theta) sin(j\theta) d\theta, \quad (3.84)$$

with

$$C_{i,j} = \frac{1}{2\pi} \int_{\theta=0}^{\pi} sin(i\theta) sin(j\theta) d\theta. \quad (3.85)$$

For the function $f_{i,j}(x)$ we have

$$\begin{cases} \lim_{x\to\pi} f_{i,j}(x) = 1, & i = j \\ \lim_{x\to\pi} f_{i,j}(x) = 0, & i \neq j \end{cases} \quad (3.86)$$

This can be easily verified by computing $C_{i,j}$ explicitly.

The derivative of it is

$$f'_{i,j}(x) = -\left(\frac{cos^2 \frac{x}{2}}{sin^2 \frac{x}{2}}\right) sin(ix) sin(jx). \quad (3.87)$$

The value of $|\frac{cos^2 \frac{x}{2}}{sin^2 \frac{x}{2}}|$ is large when x is small since the denominator is close to 0. This indicates that the first several low-rank terms are the most effective ones in changing the magnitude of the entries in the inverse matrix.

We can show similar results for 2D and 3D cases. If we use low-rank terms k_x, k_y, and k_z along each direction, the low-rank terms are more effective when both k_x, k_y, and k_z are small, which corresponding to the smallest eigenvalues of A.

Posterior correction

Next, we discuss the posterior correction where the final approximation takes the form $\hat{U}(I - Z_k Z_k^T)^2 \hat{U}^T \approx A^{-1}$. The use of posterior correction is based on the fact that most FSAI algorithms fail to approximate the smallest eigenvalues of A in many situations, especially when using sparse patterns. The idea of using low-rank correction to increase the accuracy of $\hat{U}$ has been discussed in [92]. We refer the reader to that paper for further discussion.

Iterative improvement

Finally, it is worth mentioning that we can repeat the above formula in an iterative way to improve the accuracy of the approach. That is, after having the first approximation $(I - Z_{1,k} Z_{1,k}^T)\hat{U}_1 \hat{U}_1^T (I - Z_{1,k} Z_{1,k}^T) \approx A^{-1}$, we can then define $A_1 = \hat{U}_1^T (I - Z_{1,k} Z_{1,k}^T) A (I - Z_{1,k} Z_{1,k}^T)\hat{U}_1$, and build another FSAI for A_1. The final approximation takes the form

$$A^{-1} \approx \prod_i [(I - Z_{i,k} Z_{i,k}^T)\hat{U}_i] \prod_i [\hat{U}_i^T (I - Z_{i,k} Z_{i,k}^T)] \tag{3.88}$$

3.5.8 Numerical experiments

In this section, we present a series of experiments to evaluate the performance of our algorithm. The experiments were conducted using our implementation in `MATLAB`, and we utilized `METIS` for implementing our DD algorithm.

To begin, we demonstrate the effectiveness of low-rank correction terms in constructing FSAI. The first experiment focuses on improving the decay property of the Cholesky factorization of the inverse of a matrix by incorporating a low-rank correction term. Specifically, we consider Equation 3.33 and use the coefficient matrix A of a 1D problem with a size of $1,000$.

Figure 3.23 presents the results of this experiment. The left panel illustrates the Cholesky factorization of A^{-1}, where it can be observed that the factorization contains numerous large entries. This indicates that the matrix cannot be easily approximated using FSAI alone. To address this, we construct a square-root inverse low-rank approximation matrix Z and compute the Cholesky factorization of the inverse of $(I - ZZ^T)A(I - ZZ^T)$. We set the rank of Z to 10 for this test. The right panel of

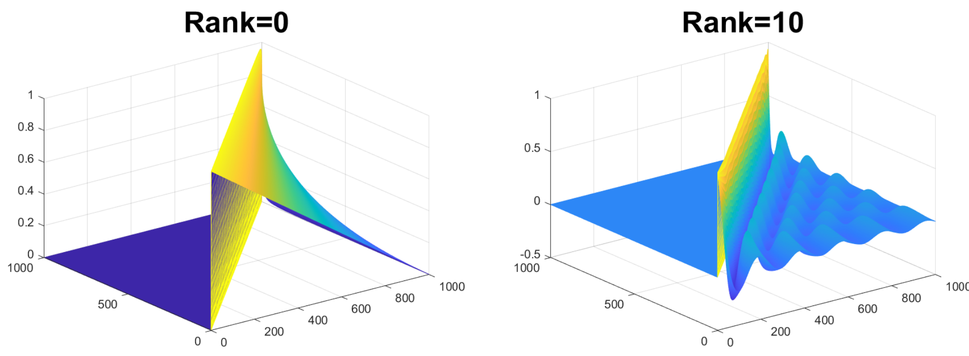


Figure 3.23: Cholesky factors of A^{-1} and $[(I - ZZ^TA(I - ZZ^T)]^{-1}$. A represents a 1D Laplacian matrix of size $1,000$, and the rank of matrix Z is set to 10.

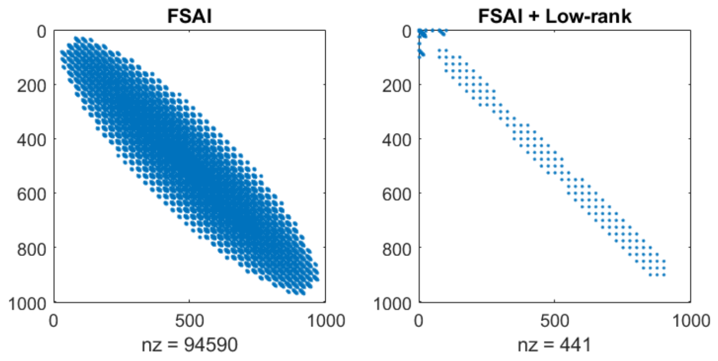


Figure 3.24: Sparsity pattern of large errors $|A^{-1} - \hat{U}\hat{U}^T| > 0.1$ (left) and $|A^{-1} - \hat{U}(I - ZZ^T)^2\hat{U}^T| > 0.1$. Here, A represents a 2D Laplacian matrix of size 25×40, $\hat{U}$ denotes the FSAI of A with a maximum of 10 nonzeros per column, and the rank of matrix Z is set to 30.

the figure shows the Cholesky factorization results for this modified matrix. As evident from the figure, the factorization exhibits significantly sparser patterns compared to the previous case. By introducing this low-rank correction term, we enable the use of FSAI to construct an approximate inverse with improved efficiency and accuracy.

In the subsequent experiments, we focus on evaluating the effectiveness of the posterior correction technique. To do so, we utilize a 2D model problem with dimensions of 25×40. Initially, we compute the FSAI of the coefficient matrix $\hat{U}\hat{U}^T \approx A$ using an iterative FSAI approach, allowing for a maximum of 10 nonzeros per column.

Following this, we construct the square-root inverse low-rank approximation matrix Z of $\hat{U}^T A \hat{U}$, with the rank set to 30. To assess the quality of the two approximate inverses, we compare the FSAI approximation $\hat{U}\hat{U}^T$ with the form $\hat{U}(I - ZZ^T)^2\hat{U}^T$. Both of these approximations serve as candidates for the inverse of the original matrix.

In Figure 3.24, we plot the sparsity pattern of large errors $|A^{-1} - \hat{U}\hat{U}^T| > 0.1$ (left) and $|A^{-1} - \hat{U}(I - ZZ^T)^2\hat{U}^T| > 0.1$. As we can see from the figure, adding a low-rank correction term can significantly reduce the number of large entries in the error.

In the next experiment, we turn our attention to evaluating the approximate inverse constructed using the polynomial low-rank correction technique. We continue with the 2D model problem of size 25×40. To generate a two-level structure, we employ a two-level 16-way partitioning scheme using a vertex-based partition. We apply the block-Jacobi FSAI approach on the C block. Taking advantage of the small size of each diagonal block in matrices B and C, although we actually utilize FSAI with low-rank correction, the approximations can be considered exact. To estimate the range of eigenvalues and compute the low-rank approximation, we utilize the thick-restart Lanczos algorithm. Additionally, the Hermitz-Remez approximation is employed to construct the polynomial. In this experiment, we compare three different approximations. The first approximation does not incorporate a low-rank correction term. The second and third approximations incorporate a low-rank correction term with a rank of 10, utilizing polynomial degrees of one and two, respectively.

In Figure 3.24, we present the sparsity pattern of errors larger than 0.01 obtained from our experiment. We reduced the tolerance to 0.01 for this experiment as the polynomial-based approaches generally yield more accurate results compared to the algorithms used in the previous experiment. As observed from the figure, incorporating

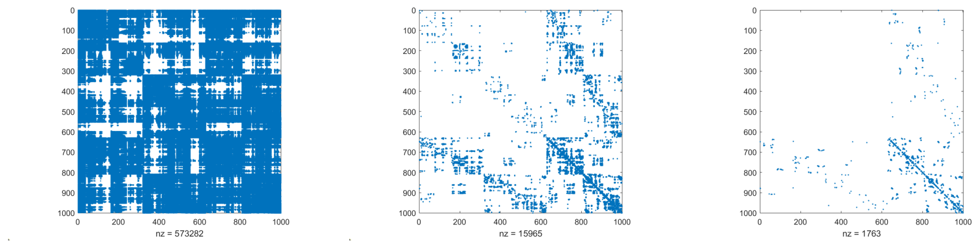


Figure 3.25: Sparsity pattern of errors exceeding 0.01 without low-rank correction (left), using a degree-1 polynomial with rank 10 (middle), and using a degree-2 polynomial with rank 10. Here, matrix A represents a 2D Laplacian matrix with dimensions of 25×40.

a low-rank correction term effectively reduces the number of large entries in the error. Furthermore, increasing the polynomial degree enhances the approximation, even with the same low-rank correction term. This demonstrates the improved accuracy achieved by utilizing higher-degree polynomials in the low-rank correction process.

In the final set of experiments, we utilize our polynomial-based approximate inverse as preconditioners for the GMRES(40) iterative solver to solve linear systems with random right-hand sides. We set the stopping tolerance for the solver to 1e-04. We continue to work with the 2D Laplacian matrix with dimensions of 25×40. In addition to this problem, we introduce a second group of tests called the shifted Laplacian problem. In this case, we add a shift of 0.1 to the diagonal of the coefficient matrix, resulting in an indefinite matrix. Due to the indefiniteness of the shifted Laplacian problem, we employ GMRES instead of the Conjugate Gradient (CG) method. For comparison purposes, we evaluate four different options. The first three options are the same as those used in the previous example, utilizing polynomial-based preconditioning with different low-rank correction terms. The fourth option involves using unpreconditioned GMRES. From the results in Figure 3.26, we can see that our polynomial low-rank approximate inverse can be used as an efficient preconditioner.

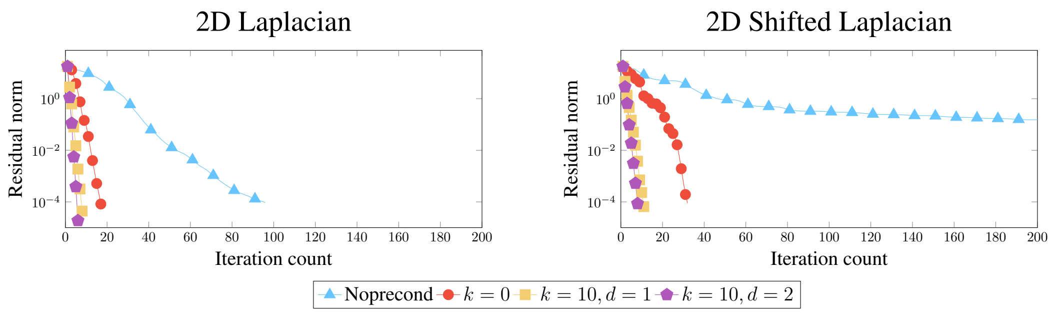


Figure 3.26: Convergence curves of different preconditioners for GMRES(40). Matrix A represents a 2D Laplacian matrix with dimensions of 25×40. The coefficient matrix for the shifted Laplacian problem includes a shift value of 0.1.

3.5.9 Conclusion

In conclusion, we introduced a novel Schur complement low-rank approximate inverse algorithm utilizing polynomial approximation. Our approach, based on the Hermite-Remez approximation, enhances the accuracy of the low-rank correction term. Moreover, we explored the use of local low-rank correction to further enhance the performance of FSAI.

In future work, we plan to evaluate the effectiveness of our proposed algorithm in real-world applications, such as traffic volume estimation [93], multivariate sampling [94], and Katz centrality with dynamic networks [95]. These applications will provide valuable insights into the practical utility of our algorithm and its potential impact in various domains.

Chapter 4

Parallel Algorithms for Eigenvalue Problems

4.1 Introduction

The fourth chapter of this dissertation considers the eigenvalue problem

$$A\mathbf{u} = \lambda M\mathbf{u}, \tag{4.1}$$

where the $n \times n$ matrices A and M are real symmetric and M is positive-definite. This dissertation focuses on the task of computing a handful of the algebraically smallest eigenvalues and associated eigenvectors that lie within $[\alpha, \beta]$. Problems of this sort arise, for instance, in spectral clustering [96], and low-frequency response analysis [97, 98].

Due to the size of modern matrix problems, parallel computing has become an integral part of software libraries targeting large-scale eigenvalue computations. In many packages (e.g., `PARPACK` [99, 100], `PRIMME` [101], `BLOPEX` [102]), linear algebra kernels are the main source of parallelism, with operations such as matrix-vector and dot products performed in parallel by distributing the data across multiple processors. Several recent packages improve scalability by exploiting additional levels of parallelism via techniques such as spectrum slicing (`pEVSL` [91]), rational filtering (`FEAST`/`PFEAST` [103, 104, 105], and `z-Pares` [106]), and parallel shift-and-invert methods [107, 108]. In addition, the `SLEPc` collection of distributed memory eigenvalue algorithms [109] contains implementations of several of these methods.

Another class of distributed memory eigenvalue solvers is based on algebraic domain decomposition, also known as algebraic substructuring; see the references in [110] for details. In domain decomposition, the adjacency graph associated with the pencil (A, M) is partitioned into several non-overlapping subgraphs. The eigenvalue problem then decouples into two separate tasks: first, one determines the eigenvector components associated with the interface variables of the partitioned graph; then, one finds the components associated with the interior variables. The second task parallelizes naturally over the subgraphs. For more information on this type of solver, see [97, 111, 110, 112, 113]. The work presented in the next section combines the domain decomposition approach with Chebyshev function approximation to design a new distributed memory parallel eigensolver. The work presented in the last section of this chapter is a new parallel version of the QR decomposition, which can be used in the work presented in the next section.

The remainder of this chapter is organized as follows: Section 4.2 presents a parallel algorithm for computing partial spectral factorizations of matrix pencils via Chebyshev approximation, which is implemented in `SchurCheb`. Section 4.3 presents a block Givens rotation-based QR decomposition algorithm.

4.2 A Parallel Schur Complement Eigenvalue Solver

The algorithm proposed in this section is a two-level Schur complement-based method using the vertex-based partition of the adjacency graph of $|A| + |M|$. The partition is similar to 3.7, and we can reorder the matrices with a permutation matrix P as shown in the following equations:

$$
P^TAP = \left(\begin{array}{cccc|cccc}
B_1 & & & & E_1 & & & \\
& B_2 & & & & E_2 & & \\
& & \ddots & & & & \ddots & \\
& & & B_p & & & & E_p \\
\hline
E_1^T & & & & C_{1,1} & C_{1,2} & \cdots & C_{1,p} \\
& E_2^T & & & C_{2,1} & C_{2,2} & \cdots & C_{2,p} \\
& & \ddots & & \vdots & \vdots & \ddots & \vdots \\
& & & E_p^T & C_{p,1} & C_{p,2} & \cdots & C_{p,p}
\end{array}\right)
$$
$$
P^TMP = \left(\begin{array}{cccc|cccc}
M_{B_1} & & & & M_{E_1} & & & \\
& M_{B_2} & & & & M_{E_2} & & \\
& & \ddots & & & & \ddots & \\
& & & M_{B_p} & & & & M_{E_p} \\
\hline
M_{E_1}^T & & & & M_{C_{1,1}} & M_{C_{1,2}} & \cdots & M_{C_{1,p}} \\
& M_{E_2}^T & & & M_{C_{2,1}} & M_{C_{2,2}} & \cdots & M_{C_{2,p}} \\
& & \ddots & & \vdots & \vdots & \ddots & \vdots \\
& & & M_{E_p}^T & M_{C_{p,1}} & M_{C_{p,2}} & \cdots & M_{C_{p,p}}
\end{array}\right). \tag{4.2}
$$

In this section, we make the assumption that matrices A and M have already been reordered accordingly. Consequently, we will omit the permutation matrix P. They can also be written in 2×2 block form as

$$
A = \begin{pmatrix} B & E \\ E^T & C \end{pmatrix}, \qquad M = \begin{pmatrix} M_B & M_E \\ M_E^T & M_C \end{pmatrix}, \tag{4.3}
$$

where the blocks are defined following the vertical and horizontal lines in Equation 4.2. In this section, we define d_i and s_i as the size of B_i and $C_{i,i}$, respectively, and define $d = d_1 + \cdots + d_p$ and $s = s_1 + \ldots + s_p$, the size of B and C, respectively.

4.2.1 A parallel algorithm based on Chebyshev approximation

The algorithm discussed in this section is based on the fact that the eigenvalues and eigenvectors of the matrix $A-\zeta M$ are analytic functions. It is easy to see that if $\zeta=\lambda_i$ is an eigenvalue of the pencil (A,M), 0 is an eigenvalue of $A-\zeta M$, and its null vectors correspond to the eigenvectors of (A,M) associated with λ_i. Continuity allows us to infer that when ζ is close to λ_i, the eigenvectors of $A-\zeta M$ corresponding to its small eigenvalues are suitable approximations to the null vectors of $A-\lambda_i M$.

Based on this concept, our algorithm calculates the eigenvectors related to the smallest eigenvalues of $A-\zeta_i M$ at multiple points ζ_i within the search interval $[\alpha,\beta]$. We show that by carefully selecting ζ_i, we can ensure that the subspace spanned by these "near-null" vectors contains reliable approximations to the eigenvectors of (A,M). Rayleigh-Ritz projection is then used to compute the final approximation.

In order to enhance the efficiency and parallelizability of this process, we reorder A and M using strategies discussed in the previous section. By doing so, we can leverage the block structure inherent in the reordered A and M. We proceed to partition the eigenvector $\mathbf{u}^{(i)}$ corresponding to the eigenvalue λ_i of (A,M) as:

$$\mathbf{u}^{(i)}=\begin{bmatrix}\mathbf{x}^{(i)}\\ \mathbf{y}^{(i)}\end{bmatrix},$$

where $\mathbf{x}^{(i)}\in\mathbb{R}^d$ and $\mathbf{y}^{(i)}\in\mathbb{R}^s$, aligning with the partitioning of A and M outlined in Equation 4.3. Additionally, we introduce the notations

$$B(\zeta)=B-\zeta M_B,\qquad E(\zeta)=E-\zeta M_E,\qquad C(\zeta)=C-\zeta M_C, \tag{4.4}$$

where $\zeta\in\mathbb{C}$. Using those notations, we can rewrite the eigenvector equation $(A-\lambda_i M)\mathbf{u}^{(i)}=0$ as

$$\begin{bmatrix}B(\lambda_i) & E(\lambda_i)\\ E^T(\lambda_i) & C(\lambda_i)\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(i)}\\ \mathbf{y}^{(i)}\end{bmatrix}=0. \tag{4.5}$$

We first assuming that $\lambda_i\notin\Lambda(B,M_B)$, i.e., $B(\lambda_i)$ is invertible, we can then substitute

$$\mathbf{x}^{(i)}=-B(\lambda_i)^{-1}E(\lambda_i)\mathbf{y}^{(i)}, \tag{4.6}$$

into the second row to get

$$[C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{-1}E(\lambda_i)]\mathbf{y}^{(i)} = 0. \tag{4.7}$$

The equation indicates that $\mathbf{y}^{(i)}$ is a null vector of the Schur complement $C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{-1}E(\lambda_i)$. That is, we can compute the $s \times 1$ bottom part of the eigenvector $\mathbf{u}^{(i)}$ using the Schur complement. Once we obtain $\mathbf{y}^{(i)}$, the $d \times 1$ upper part can be computed using Equation 4.6. During this step, solving a block diagonal linear system of size $d \times d$ is required.

Next, we discuss the situation where $\lambda_i \in \Lambda(B, M_B)$, although this scenario is infrequent in practical applications. In this case, we can divide the eigenvector into $\mathbf{x}^{(i)} = \mathbf{x}_P^{(i)} + \mathbf{x}_N^{(i)}$, where $\mathbf{x}_P^{(i)} \in \operatorname{Ran}(B(\lambda_i))$ and $\mathbf{x}_N^{(i)} \in \operatorname{Ker}(B(\lambda_i))$. Denote by $B^+(\lambda_i)$ the (Moore–Penrose) pseudoinverse of $B(\lambda_i)$, instead of Equation 4.6, we have

$$\mathbf{x}_P^{(i)} = -B(\lambda_i)^+E(\lambda_i)\mathbf{y}^{(i)}. \tag{4.8}$$

Again subsitute it into the second row of Equation 4.5, instead of Equarion 4.7, we now have

$$E(\lambda_i)^T\mathbf{x}_N^{(i)} + [C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^+E(\lambda_i)]\mathbf{y}^{(i)} = 0. \tag{4.9}$$

The first term $E(\lambda_i)^T\mathbf{x}_N^{(i)}$ is zero when $\operatorname{Ran}(E(\lambda_i)) \perp \operatorname{Ker}(B(\lambda_i))$. In this situation, the eigenvectors can be found in a manner analogous to the case when $\lambda_i \notin \Lambda(B, M_B)$ but with $B(\lambda_i)^{-1}$ replaced by $B(\lambda_i)^+$. However, it is better to simply avoid the case $\lambda_i \in \Lambda(B, M_B)$. A simple approach is to adjust the DD until the smallest eigenvalue of (B, M_B) is greater than β, which can be done by increasing the number of subdomains. This is unlikely to be necessary because none of the numerical experiments we reported in the experiment section required it. Therefore, for simplicity, we will not discuss a comprehensive strategy for this, leaving it as a potential topic for future research, and assume that $\beta < \min(\Lambda(B, M_B))$, and thus the $B(\zeta)$ we use are all invertible.

Return to Equation 4.6, define matrix value function

$$S(\zeta) = C(\zeta) - E^T(\zeta)B(\zeta)^{-1}E(\zeta), \tag{4.10}$$

our goal becomes finding all $\zeta \in [\alpha, \beta]$ such that $S(\zeta)$ is singular. $S(\zeta)$ is commonly referred to as the spectral Schur complement [114, 115]. Denote by $\mu_1(\zeta), \dots, \mu_s(\zeta)$ and $\mathbf{y}_1(\zeta), \dots, \mathbf{y}_s(\zeta)$ the eigenvalues and the corresponding eigenvectors of $S(\zeta)$:

$$S(\zeta)\mathbf{y}_i(\zeta) = \mu_i(\zeta)\mathbf{y}_i(\zeta), \qquad i = 1, \dots, s.$$

It is shown that μ_i and $\mathbf{y}_i$ are analytic functions of $\zeta \in \mathbb{C}$ away from $\Lambda(B, M_B)$, and have at most a pole singularity at each point of $\Lambda(B, M_B)$ [116, 117, 118, 119]. We refer to $\mu_i(\zeta)$ as the *eigencurves* of S. We also define the upper part as:

$$\mathbf{x}_i(\zeta) = -B(\zeta)^{-1}E(\zeta)\mathbf{y}_i(\zeta), \qquad i = 1, \dots, s,$$

which is also analytic in regions where $\mathbf{y}_i(\zeta)$ is analytic.

The spectral Schur complement is singular if and only if at least one of its eigenvalues $\mu_i(\zeta) = 0$ is zero. Assume that A has $n_{\text{ev}} \leq s$ eigenvalues in $[\alpha, \beta]$ counted according to multiplicity. In the following proposition, we show the connection between the eigenpairs of A and the spectral Schur complement. The assumption that $\beta < \min(\Lambda(B, M_B))$ in the proposition ensures that all the eigencurves are analytic in $[\alpha, \beta]$, while the assumption $n_{\text{ev}} \leq s$ ensures that there are no less than n_{ev} eigencurves, and thus the space we plan to search is large enough to contain all the eigenvectors we seek for.

Proposition 4.2.1. *Assume $\beta < \min(\Lambda(B, M_B))$, and $n_{\text{ev}} \leq s$. Then, there exist n_{ev} distinct integers $\kappa_1, \dots, \kappa_{n_{\text{ev}}} \in \{1, 2, \dots, s\}$ such that*

$$\mu_{\kappa_i}(\lambda_i) = 0, \qquad \mathbf{y}^{(i)} = \mathbf{y}_{\kappa_i}(\lambda_i), \qquad \mathbf{x}^{(i)} = \mathbf{x}_{\kappa_i}(\lambda_i). \tag{4.11}$$

Proof. First, consider the case in which the λ_i are all simple eigenvalues. Following (4.7), we have $S(\lambda_i)\mathbf{y}^{(i)} = 0$ for some where $\mathbf{y}^{(i)} \neq 0$. The matrix $S(\lambda_i)$ is singular and has exactly one zero eigenvalue, denoted by $\mu_{\kappa_i}(\lambda_i)$, for some $1 \leq \kappa_i \leq s$. The expressions in (4.11) follow directly. It remains to show that $\kappa_i \neq \kappa_j$ when $i \neq j$. This follows from the fact that the μ_{κ_i} are free of poles and strictly decreasing on $[\alpha, \beta]$ [116], which implies that λ_i is the only root of μ_{κ_i} in $[\alpha, \beta]$.

That the result also holds in the case where one or more of the λ_i have non-unit multiplicity can be seen by considering arbitrarily small perturbations of (A, M) that have all simple eigenvalues and appealing to continuity. $\square$

Proposition 4.2.1 tells us that the n_{ev} smallest eigenvalues of A are roots of n_{ev} distinct eigencurves, and the corresponding eigenvectors can be formed using the corresponding $\mathbf{y}_i(\zeta)$ and $\mathbf{x}_i(\zeta)$ at the root of $\mu_i(\zeta)$. For eigenvalues of non-unit multiplicity, there are multiple eigencurves with the same root. Since both $\mathbf{y}_i(\zeta)$ and $u_i(\zeta)$ are analytic on $[\alpha, \beta]$, they can be accurately approximated by interpolation at Chebyshev nodes in $[\alpha, \beta]$. We use the Chebyshev nodes of the second kind, which are defined as

$$\chi_j = \frac{\alpha+\beta}{2} + \cos\left(\frac{j\pi}{N-1}\right)\frac{\beta-\alpha}{2}, \; j = 0, \ldots, N-1, \tag{4.12}$$

for $N > 1$, and $\chi_0 = (\alpha+\beta)/2$ for $N = 1$.

In the following proposition, we show the accuracy of the approximation. Without loss of generality, we assume that $\kappa_i = i$ in the remainder of this section, which is equivalent to saying that $\mu_i(\lambda_i) = 0$ for all i. Denote by ℓ_j the jth Lagrange basis function for polynomial interpolation in these nodes, which is a degree $N-1$ polynomial takes the value one at χ_j and zero at χ_k for $k \neq j$. We also need the concept of *Bernstein ellipse* in the proposition. We denote by $\mathcal{E}_\rho$ the Bernstein ellipse centered on $[\alpha, \beta]$ with the parameter ρ, which is an open subset of $\mathbb{C}$. $\mathcal{E}_\rho$ is bounded by the ellipse with foci at α and β, and the sum of the lengths of its semi-major and semi-minor axes is equal to ρ. Since $\mathbf{y}_i(\zeta)$ and $\mathbf{x}_i(\zeta)$ are analytic on $[\alpha, \beta]$, they can be analytically continued to $\mathcal{E}_\rho$ for some $\rho > 0$. We now give the following proposition:

Proposition 4.2.2. *Assume that $\beta < \min(\Lambda(B, M_B))$, that $n_{\text{ev}} \leq s$, and that $\mathbf{x}_i$ and $\mathbf{y}_i$ are analytic in $\mathcal{E}_\rho$ for all $i = 1, \ldots, n_{\text{ev}}$ and some $\rho > 0$. For each i, there exists $w^{(i)} \in \mathbb{R}^N$ such that*

$$\mathbf{u}^{(i)} = \begin{bmatrix} \mathbf{x}^{(i)} \\ \mathbf{y}^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_i(\chi_0) & \cdots & \mathbf{x}_i(\chi_{N-1}) \\ \mathbf{y}_i(\chi_0) & \cdots & \mathbf{y}_i(\chi_{N-1}) \end{bmatrix} w^{(i)} + O(\rho^{-N}).$$

Proof. Let $w_j^{(i)} = \ell_j(\lambda_i)$ for $j = 0, \ldots, N-1$. Then, the top d (respectively, bottom s) components of the matrix-vector product give the value at λ_i of the polynomial interpolant to $\mathbf{x}(i)$ (respectively, $\mathbf{y}^{(i)}$) in the Chebyshev nodes χ_j. The result now follows from a standard theorem on the convergence of Chebyshev interpolants to analytic functions [120, Theorem 8.2]. $\square$

Proposition 4.2.2 indicates that we can use $\mathbf{x}_i(\zeta)$ and $\mathbf{y}_i(\zeta)$ at the Chebyshev nodes to generate a subspace close to $\mathbf{u}_i$, and ensures that the subspace contains a good approximation with sufficiently large N. Thus, the subspace cR, which is defined as

$$\mathcal{R} = \operatorname{span}\left\{ \begin{bmatrix} \mathbf{x}_1(\chi_0) \\ \mathbf{y}_1(\chi_0) \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{x}_1(\chi_{N-1}) \\ \mathbf{y}_1(\chi_{N-1}) \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{x}_{n_{\text{ev}}}(\chi_0) \\ \mathbf{y}_{n_{\text{ev}}}(\chi_0) \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{x}_{n_{\text{ev}}}(\chi_{N-1}) \\ \mathbf{y}_{n_{\text{ev}}}(\chi_{N-1}) \end{bmatrix} \right\}$$

contains approximations to all of the desired eigenvectors. Note that this approach does not require tracking each individual eigencurve and thus avoid the potential difficulty in the situation where eigencurves cross with each other. For example, we might have $\mu_2(\chi_{j_1}) < \mu_1(\chi_{j_1})$ and $\mu_2(\chi_{j_2}) > \mu_1(\chi_{j_2})$, making it difficult to distinguish $\mathbf{y}_1(\zeta)$ from $\mathbf{y}_2(\zeta)$. However, the order of these vectors does not influence the final subspace $\mathcal{R}$. In the next corollary, we provide a statement about the angle between this subspace and the sought eigenspace:

Corollary 4.2.1. *Let* $\mathcal{X} = \operatorname{span}\{\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(n_{\text{ev}})}\}$. *Then we have*

$$\sin\theta(\mathcal{X}, \mathcal{R}) = O(\rho^{-N}),$$

where $\theta(\mathcal{X}, \mathcal{R})$ *is the largest principal angle between* $\mathcal{X}$ *and the closest subspace of* $\mathcal{R}$ *to* $\mathcal{X}$ *with the same dimension as* $\mathcal{X}$.

Proof. The quantity $\sin\theta(\mathcal{X}, \mathcal{R})$ is known as the *gap* between $\mathcal{X}$ and $\mathcal{R}$ and can be expressed as [121] [117, sect. IV.2.1] [122, sect. II.4]

$$\sin\theta(\mathcal{X}, \mathcal{R}) = \max_{\mathbf{x}\in\mathcal{X}} \min_{\mathbf{r}\in\mathcal{R}} \frac{\|\mathbf{x} - \mathbf{r}\|}{\|\mathbf{x}\|}.$$

The result follows immediately from this formula and Proposition 4.2.2. □

Our algorithm solves an eigenvalue problem with the spectral Schur complement at each Chebyshev node to build the subspace $\mathcal{R}$. After that, Rayleigh-Ritz projection is used to build approximations to the n_{ev} desired eigenpairs of A. We summarize the entire procedure in Algorithm 24.

The proposed algorithm computes the eigenvectors associated with the n_{ev} algebraically smallest eigenvalues of $S(\chi_j)$ for each j. We then combine the eigenvectors

Algorithm 24 Schur Complement Chebyshev Eigenvalue Solver

```
1: Call a $p$-way edge separator to partition the graph associated with $|A| + |M|$.
2: If $\beta < \min(\Lambda(B, M_B))$ continue, else set $p := 2p$ and repeat step 1.
3: for $j = 0, \ldots, N-1$ do
4:     Set $\chi_j = \frac{\alpha+\beta}{2} + \cos\left(\frac{j\pi}{N-1}\right)\frac{\beta-\alpha}{2}$.
5:     Set $Y_j = [\mathbf{y}_1(\chi_j), \ldots, \mathbf{y}_{n_{\text{ev}}}(\chi_j)]$.
6:     Solve $B(\chi_j)V_j = -E(\chi_j)Y_j$.
7: end for
8: Set $R = \begin{pmatrix} V_0 & \cdots & V_{N-1} \\ Y_0 & \cdots & Y_{N-1} \end{pmatrix}$.
9: Optionally, orthonormalize the columns of $R$.
10: Compute the $n_{\text{ev}}$ algebraically smallest eigenvalues and associated eigenvectors of
    the eigenvalue problem $(R^T AR)\mathbf{f} = \theta(R^T MR)\mathbf{f}$.
11: return $(\theta_i, PR\mathbf{f}^{(i)}) \approx \left(\lambda_i, \mathbf{u}^{(i)}\right),\ i = 1, \ldots, n_{\text{ev}}$.
```

of $S(\chi_j)$ to form the $s \times n_{ev}$ matrix Y_j. The corresponding upper part of Y_j is then computed as $V_j = -B^{-1}(\chi_j)E(\chi_j)Y_j$, which requires the solution of a linear system with multiple right-hand sides, as shown in Step 6 of the algorithm. Finally, matrices from different Chebyshev nodes are combined, and Rayleigh-Ritz projection is applied to the subspace spanned by this matrix to compute the final approximation, as shown in Steps 9-11 of the algorithm.

This algorithm is based on the hypothesis $\beta < \min(\Lambda(B, M_B))$. The Step 2 of the algorithm is added for this purpose. The intuition behind this step is that, if $\delta_1, \delta_2, \ldots, \delta_d$ are the eigenvalues of (B, M_B), then we have $\lambda_i \leq \delta_i \leq \lambda_{i+n-d}$, $i = 1, \ldots, d$ by a version of the interlacing theorem. Therefore, increasing the value of p makes it more likely that $\lambda_{n_{\text{ev}}} \leq \beta < \delta_1$, since a higher value of p typically leads to a larger separator. In our algorithm, we use a strategy that keeps doubling p until $\beta < \min(\Lambda(B, M_B))$ is satisfied.

There are two levels of parallelism within the `for` loop in Steps 3-7 of the algorithm. First, the V_j and Y_j on each Chebyshev node can be computed independently without information from other Chebyshev nodes. Second, a large portion of the computation of V_j and Y_j can also be computed in parallel thanks to the block structures of A and

M. V_j and Y_j can be further partitioned by row as

$$V_j = \begin{bmatrix} V_{1,j} \\ \vdots \\ V_{p,j} \end{bmatrix}, \qquad Y_j = \begin{bmatrix} Y_{1,j} \\ \vdots \\ Y_{p,j} \end{bmatrix},$$

aligning with block rows in B and C, respectively. We can then write Step 6 of the algorithm into block form as

$$\begin{bmatrix} B_1(\chi_j) & & \\ & \ddots & \\ & & B_p(\chi_j) \end{bmatrix} \begin{bmatrix} V_{1,j} \\ \vdots \\ V_{p,j} \end{bmatrix} = - \begin{bmatrix} E_1(\chi_j) & & \\ & \ddots & \\ & & E_p(\chi_j) \end{bmatrix} \begin{bmatrix} Y_{1,j} \\ \vdots \\ Y_{p,j} \end{bmatrix}. \tag{4.13}$$

Here, we have extended the subscript notation for the block structure to the matrix-valued functions in Equation 4.4 in a natural way. It is obvious that each block $V_{k,j}$ can be computed by solving the following system with a reduced size:

$$B_k(\chi_j)V_{k,j} = -E_k(\chi_j)Y_{k,j}.$$

The p blocks in V can be solved in parallel.

In practice, taking N as $N(k) = 2^k + 1$ for some integer k can be beneficial. We find that selecting $N = 8$ reaches nearly the maximum attainable accuracy on a wide range of problems in practice; see Section 4.2.3. However, all steps in Algorithm 24 must be repeated with a higher value of N if the accuracy of the approximate eigenpairs is not satisfactory unless some Chebyshev nodes remain the same with the new N. Since the Chebyshev nodes of $N(k_1)$ are a subset of those of $N(k_2)$ if $k_2 > k_1$, most of the results can be reused if we increase the number of Chebyshev nodes from $N(k_1)$ to $N(k_2)$.

Another practical consideration is that the number n_{ev} might be unknown. It can be computed by decomposing $A - \alpha M$ and $A - \beta M$ and using Sylvester's law of inertia [123]. Alternatively, we can estimate the value n_{ev} using a spectral density profile of (A, M) [124], and use a n_{ev} slightly higher than the estimation to reduce the chance of missing eigenvalues. One can also use the approximate eigenvectors obtained from Algorithm 24 as the initial subspace for an implicitly-restarted (or thick-restarted) Lanczos method [125, 126] applied to (A, M). This extra step can be used to ensure that all the eigenpairs of (A, M) have been computed and to further improve accuracy. Although M^{-1} is needed, some iterative approaches can be used since the number of solves needed typically should not be large.

4.2.2 Implementation details

In this section, we discuss the implementation of Algorithm 24 in the library `SchurCheb`. In our discussion, we assume a 2D MPI grid with $N_p = p_r p_c$ MPI processes with p_r rows and p_c columns. We denote by G_i^r the MPI communicator associated with the ith row for $i = 0, \ldots, p_r - 1$, and G_j^c the MPI communicator associated with the jth column for $j = 0, \ldots, p_c - 1$.

Data layout

Our parallel implementation utilizes the row dimension of the 2D MPI grid for distributed storage of A and M, and the column dimension of the 2D MPI grid for distributed computing of the N Chebyshev nodes. Therefore, the dimensions of the row and column of the grid satisfy the inequalities $p_r \leq p$ and $p_c \leq N$, respectively. The dimensions are typically selected so that N is an integer multiple of p_c, and p is an integer multiple of p_r. To simplify our discussion, without loss of generality, we assume $N = p_c$ and $p = p_r$.

To begin with, we discuss the data distribution along the row dimension of the grid. We distributed A and M along this dimension such that each column communicator G_j^c holds a complete copy of A and M using row-distributed storage. To be more specific, under our assumption $p_r = p$, the ith processor is assigned data associated with the ith subdomain, i.e.,

$$\texttt{Data held by process } i \texttt{ of } G_j^c\texttt{:} \begin{cases} B_i, M_{B_i} \\ E_i, M_{E_i} \\ C_{i,1}, \ldots, C_{i,p}, M_{C_{i,1}}, \ldots, M_{C_{i,p}} \end{cases}.$$

Ordering the unknowns/equations by increasing MPI rank leads to the following global representation of A (and similarly for M):

$$A = \begin{bmatrix} B_1 & E_1 & & & & & \\ E_1^T & C_{1,1} & & C_{1,2} & & & C_{1,p} \\ & & B_2 & E_2 & & & \\ & C_{2,1} & E_2^T & C_{2,2} & & & C_{2,p} \\ & & & & \ddots & & \\ & & & & & B_p & E_p \\ & C_{p,1} & & C_{p,2} & & E_p^T & C_{p,p} \end{bmatrix}. \tag{4.14}$$

The ordering in Equation 4.14 is more natural from the perspective of parallel computing than that in Equation 4.2, which is more natural for discussing the linear algebra.

We now turn our attention to the data distribution along the column dimension of the grid. We distributed the Chebyshev nodes across this dimension. Under our assumption $p_c = N$, each column communicator receives one Chebyshev node. In particular, the jth Chebyshev node is assigned to G_j^c.

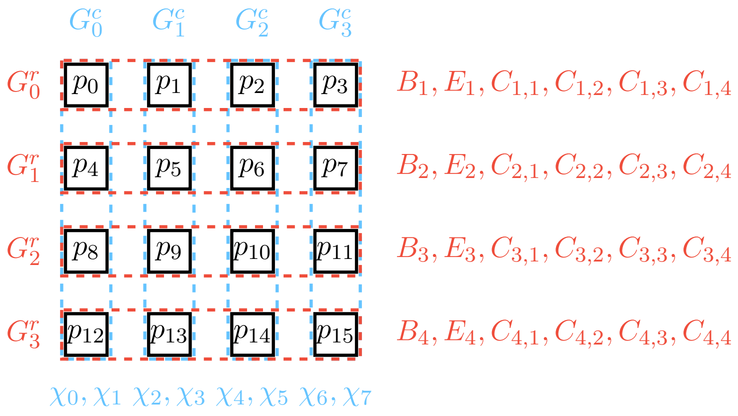
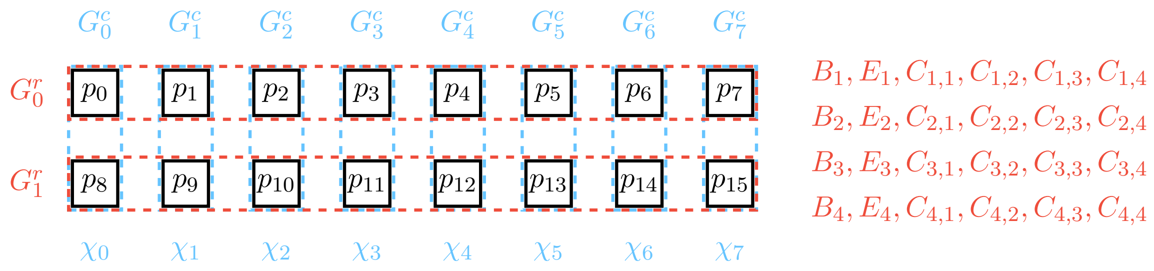


Figure 4.1: Distribution of blocks of A and Chebyshev nodes over a 2D MPI grid with $N_p = 16$, $N = 8$, $p = 4$, and $(p_r, p_c) = (4, 4)$. The distribution of M is identical to that of A.

The data layout strategy can be easily generalized to the situation where $p_c < N$, or $p_r < p$, or both. An illustration of the data distribution on a 2D MPI grid with $N_p = 16$ processes, $N = 8$ Chebyshev nodes, and $p = 4$ subdomains is shown in Figures 4.1 and 4.2 where the dimensions of the grid are $(p_r, p_c) = (4, 4)$ and $(p_r, p_c) =$



G_0^c G_1^c G_2^c G_3^c G_4^c G_5^c G_6^c G_7^c
G_0^r
G_1^r
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
χ_0 χ_1 χ_2 χ_3 χ_4 χ_5 χ_6 χ_7
$B_1, E_1, C_{1,1}, C_{1,2}, C_{1,3}, C_{1,4}$
$B_2, E_2, C_{2,1}, C_{2,2}, C_{2,3}, C_{2,4}$
$B_3, E_3, C_{3,1}, C_{3,2}, C_{3,3}, C_{3,4}$
$B_4, E_4, C_{4,1}, C_{4,2}, C_{4,3}, C_{4,4}$

Figure 4.2: Distribution of blocks of A and Chebyshev nodes over a 2D MPI grid with $N_p = 16$, $N = 8$, $p = 4$, and $(p_r, p_c) = (2, 8)$. The distribution of M is identical to that of A.

$(2, 8)$, respectively. For $(4, 4)$ case we have $p_c < N$, and each column communicator is responsible for processing $8/4 = 2$ Chebyshev nodes, while the computation of each matrix pair (Y_j, V_j) exploits four MPI processes. Contrast this with the $(2, 8)$ case, in which each separate column communicator handles exactly one Chebyshev node, leading to trivial parallelism with respect to the 8 Chebyshev nodes, but the computation of each matrix pair (Y_j, V_j) utilizes just two processes.

In practice, we recommend using $p_c = N$ since this choice generally leads to better parallel efficiency. Parallelism along Chebyshev nodes requires no communication among groups.

Computation of Y_j

The first step of Algorithm 24 that is worth discussing is the computation of the n_{ev} algebraically smallest eigenvectors of the Schur complement matrices $S(\chi_j)$, $j = 0, \ldots, N-1$. Our implementation utilizes the `PARPACK` software library, which is a distributed memory implementation of `ARPACK` [99]. The main distributed memory kernels of `PARPACK` are: (a) a user-defined routine that performs distributed matrix-vector multiplication with $S(\chi_j)$, and (b) orthogonalization of the Krylov basis.

Let us first consider (a) when $p_r = p$. The distribured matrix-vector multiplication between the matrix $S(\chi_j)$ and a distributed vector $f = \begin{bmatrix} f_1^T & \cdots & f_p^T \end{bmatrix}^T \in \mathbb{R}^s$ can be written as:

$$S(\chi_j)f = \begin{bmatrix} \sum\limits_{k\in\mathcal{N}_1} C_{1,k}(\chi_j)f_k \\ \vdots \\ \sum\limits_{k\in\mathcal{N}_p} C_{p,k}(\chi_j)f_k \end{bmatrix} - \begin{bmatrix} E_1(\chi_j)^T B_1(\chi_j)^{-1} E_1(\chi_j) f_1 \\ \vdots \\ E_p(\chi_j)^T B_p(\chi_j)^{-1} E_p(\chi_j) f_p \end{bmatrix}, \tag{4.15}$$

where $\mathcal{N}_i$ denotes the list of subdomains adjacent to subdomain i. Here, we again have extended the subscript notation for the block structure to the matrix-valued functions in Equation 4.4 in a natural way. It is obvious that the second term on the right-hand side of Equation 4.15 can be computed in parallel without communication. On the other hand, the first term of the right-hand side of Equation 4.15 requires point-to-point communication between processes assigned to neighboring subdomains.

Regarding (b), since this part is built in `PARPACK`, we only analyze its cost. Consider the case $p_r = p$. Orthonormalizing the basis vectors computed on each m-step cycle of the implicitly-restarted Arnoldi method via Gram-Schmidt costs $O(sm^2)$ floating-point operations and $O(\log(p_r)m^2)$ point-to-point MPI messages. This communication cost increases proportionally with the number of Chebyshev nodes processed by each column communicator. In particular, when $p_c = 1$, i.e., all available N_p MPI processes are assigned to the default communicator, `PARPACK` requires $O(N\log(N_p)m^2)$ MPI messages just for Gram-Schmidt.

Orthonormalization of the Rayleigh-Ritz basis

Next, we discuss our implementation for the computation of an orthonormal basis of the matrix R for the Rayleigh-Ritz projection. During this step, in order to use all N_p MPI processes, we use the default communicator `MPI_COMM_WORLD`.

After the `for` loop in Steps 3-7 of the algorithm, the (i,j) process of the $p_r \times p_c$ 2D MPI grid holds the submatrices $Y_{i,j}$ and the corresponding $V_{i,j}$. We can represent this

2D distribution of R as:

$$\widehat{R}_{\text{2D}} = \begin{array}{c} \begin{array}{cccc} G_0^c & G_1^c & \cdots & G_{p_c-1}^c \end{array} \\ \left[\begin{array}{cccc} \begin{bmatrix} V_{0,0} \\ Y_{0,0} \end{bmatrix} & \begin{bmatrix} V_{0,1} \\ Y_{0,1} \end{bmatrix} & \cdots & \begin{bmatrix} V_{0,p_c-1} \\ Y_{0,p_c-1} \end{bmatrix} \\ \vdots & \vdots & \vdots & \vdots \\ \begin{bmatrix} V_{p_r-1,0} \\ Y_{p_r-1,0} \end{bmatrix} & \begin{bmatrix} V_{p_r-1,1} \\ Y_{p_r-1,1} \end{bmatrix} & \cdots & \begin{bmatrix} V_{p_r-1,p_c-1} \\ Y_{p_r-1,p_c-1} \end{bmatrix} \end{array}\right] \end{array} \begin{array}{c} G_0^r \\ \vdots \\ G_{p_r-1}^r \end{array} . \tag{4.16}$$

Our objective is to transform $\widehat{R}_{\text{2D}}$ into a $n \times Nn_{\text{ev}}$ matrix $\widehat{R}_{\text{1D}}$, such that one of the N_p processes holds a block row of R with approximately n/N_p rows and Nn_{ev} columns. To achieve this, we employ a two-step procedure. First, we perform a gather reduction on the submatrices $\begin{bmatrix} V_{i,j}^T & Y_{i,j}^T \end{bmatrix}^T$, $j = 0, \ldots, p_c-1$. This reduction is executed independently within each communicator G_i^r, $i = 0, \ldots, p_r - 1$. Second, each process associated with G_i^r discards all rows of the reduced matrix, except for a unique, continuous set of rows. This process leads us to the final 1D distribution of R as depicted below:

$$\widehat{R}_{\text{1D}} = \begin{bmatrix} V_{0,0} & \cdots & V_{0,p_c-1} \\ Y_{0,0} & \cdots & Y_{0,p_c-1} \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ V_{p_r-1,0} & \cdots & V_{p_r-1,p_c-1} \\ Y_{p_r-1,0} & \cdots & Y_{p_r-1,p_c-1} \end{bmatrix} = \begin{bmatrix} R_{0,0} \\ \vdots \\ R_{0,p_c-1} \\ \vdots \\ R_{p_r-1,0} \\ \vdots \\ R_{p_r-1,p_c-1} \end{bmatrix}, \tag{4.17}$$

where $R_{i,j}$ is held by the MPI process of rank $ip_c + j$ associated with `MPI_COMM_WORLD`, i.e., the jth process associated with the row communicator G_i^r. This can be done efficiently in a single line of code by calling `MPI_Alltoall` independently within each communicator G_i^r, $i = 0, \ldots, p_r-1$. A graphical illustration of this 2D-to-1D grid remapping is shown in Figure 4.3.

After completing the 2D-to-1D grid remapping, we proceed to perform distributed block Gram-Schmidt on the columns of $\widehat{R}_{\text{1D}}$ using the default communicator `MPI_COMM_`

`WORLD` and a block size of n_{ev}. Once this process is complete, we reverse the above procedure to map $\widehat{R}_{1\text{D}}$ back to the original 2D layout. For further details on parallel Gram-Schmidt, including a discussion of numerical stability, the reader may refer to [127, 128].

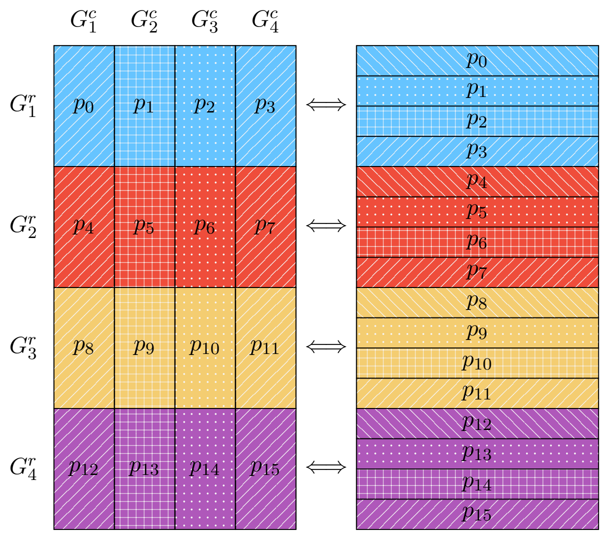


Figure 4.3: 2D-to-1D (and vice-versa) MPI grid mapping. Left: color/pattern layout of a 2D grid of MPI processes with $p_c = p_r = 4$. Right: color/pattern layout of the same grid collapsed in 1D MPI grid topology.

Formation and solution of the projected eigenvalue problem

Finally, we discuss our implementation of the Rayleigh-Ritz projection, more specifically, the formation of projected pencil (R^TAR, R^TMR) and the computation of its eigenpairs.

Since the projected pencil is relatively small (at most Nn_{ev}), we can efficiently compute its eigenvalues in a serial manner using the `DSYGVX` routine from the `LAPACK` library [129]. The subsequent discussion will focus on our approach to forming R^TAR within the 2D distributed memory data layout mentioned earlier. The procedure to form R^TMR is identical and will be treated in the same manner.

We form R^TAR in two phases. Let $R_j = \begin{bmatrix} V_j^T & Y_j^T \end{bmatrix}^T$. In the first phase, we compute $AR = \begin{bmatrix} AR_0 & AR_1 & \cdots & AR_{N-1} \end{bmatrix}$. Since each of the products AR_j, $j = 0, \ldots, N-1$, can be computed independently, the computation of each AR_j only involves communication within each column communicator between processes assigned to neighboring subdomains. Using the global representation of A from Equation 4.14, we write

$$AR_j = \begin{bmatrix} B_1 & E_1 & & & & & \\ E_1^T & C_{1,1} & & C_{1,2} & & & C_{1,p_r} \\ & & B_2 & E_2 & & & \\ & C_{2,1} & E_2^T & C_{2,2} & & & C_{2,p_r} \\ & & & & \ddots & & \\ & & & & & B_{p_r} & E_{p_r} \\ & C_{p_r,1} & & C_{p_r,2} & & E_{p_r}^T & C_{p_r,p_r} \end{bmatrix} \begin{bmatrix} V_{0,j} \\ Y_{0,j} \\ V_{1,j} \\ Y_{1,j} \\ \vdots \\ V_{p_r-1,j} \\ Y_{p_r-1,j} \end{bmatrix}. \qquad (4.18)$$

The second phase multiplies R^T and AR and stores the matrix product in the root process of `MPI_COMM_WORLD`. To achieve this, we apply the following procedure, which is illustrated in Figure 4.4:

1. Apply `MPI_Allgather` on the submatrices $[AR_j]_i$, $j = 0, \ldots, p_c - 1$, across the row communicator G_i^r, where $[AR_j]_i$ denotes the submatrix of AR_j held by the ith process. Each process associated with G_i^r then has its own copy of the matrix $\begin{bmatrix}[AR_0]_i & [AR_2]_i & \cdots & [AR_{p_c-1}]_i\end{bmatrix}$.

2. The ith process associated with the column communicator G_j^c then computes $Z_{i,j} = R_{i,j}^T \begin{bmatrix}[AR_0]_i & [AR_2]_i & \ldots & [AR_{p_c-1}]_i\end{bmatrix}$ and calls `MPI_Reduce` on the data $Z_{i,j}$ associated with the processes in G_j^c.

3. At the end of the previous step, the kth MPI process associated with G_0^r holds the kth block of rows of the matrix $R^T AR$. Finally, all processes in G_0^r call `MPI_Gather`, creating $R^T AR$ in the root process.

4.2.3 Numerical experiments

In this section, we demonstrate the performance of our algorithm. Our algorithm is implemented in the `SchurCheb` library. We performed our experiments on the Minnesota Supercomputing Institute's `Mesabi` cluster. Each node of `Mesabi` is equipped with 64

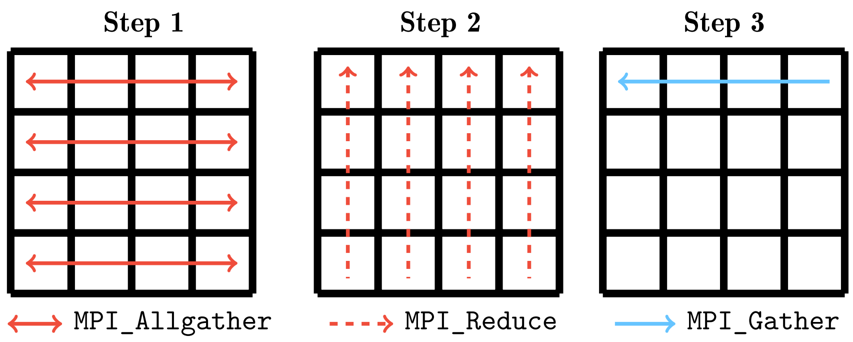


Figure 4.4: Communication pattern for the distributed memory computation of R^TAR and R^TMR using our 2D MPI data layout ($p_r = p_c = 4$). The root process of `MPI_COMM_WORLD` is located in the upper left corner.

GB of system memory and two 12-core 2.5 GHz Intel Xeon E5-2680v3 (Haswell) CPUs. We built our code with the Intel ICC 18.0.0 compiler. We used the Intel Math Kernel Library (MKL) for basic matrix operations, including its sparse matrix routines and its implementation of the standard `BLAS` and `LAPACK` libraries for sequential dense matrix operations. While it is possible to exploit shared-memory parallelism, the experiments described below use just one thread per MPI process. We assume a distributed memory computing environment with $\mathbf{n_p} = p_r p_c$ MPI processes organized in a $p_r \times p_c$ 2D MPI grid with p_r rows and p_c columns. Our parallel implementation utilizes the row dimension of the grid for domain-decomposition data parallelism (i.e., distributed storage of A and M), and the column dimension of the grid for model parallelism (i.e., distribution over the N Chebyshev nodes). Therefore, the row and column dimensions of the grid satisfy the inequalities $p_r \leq p$ and $p_c \leq N$, respectively.

To compute the n_{ev} sought eigenvectors of the spectral Schur complements $S(\chi_j)$, we used `PARPACK` with full orthogonalization and restart dimension $m = 2n_{\text{ev}}$. The linear systems involving the block-diagonal matrix $B(\chi_j)$ were solved with the Intel MKL implementation of the `PARDISO` solver. For the search interval $[\alpha, \beta]$, we set $\alpha = 0$, $\beta = (\lambda_{n_{\text{ev}}} + \lambda_{n_{\text{ev}}+1})/2$ in all experiments.

We compare our algorithm with *a*) `PARPACK` with and without shift-and-invert, and *b*) the Locally Optimal Block Preconditioned Conjugate Gradient (`LOBPCG`) method with AMG preconditioning as implemented in the `BLOPEX` package of `hypre` [130].

Numerical illustration

We first demonstrate the qualitative performance of Algorithm 24 on a set of four small problems:

- "APF4686," a standard eigenvalue problem of dimension $n = 4{,}686$ generated by the ELSES quantum mechanical nanomaterial simulator[1] [131],
- "Kuu/Muu," a generalized eigenvalue problem of dimension $n = 7{,}102$ from the SuiteSparse matrix collection[2] [132],
- "FDmesh," a standard eigenvalue problem generated by a regular 5-point finite difference discretization of the Laplacian on a square, and
- "FEmesh," a generalized eigenvalue problem obtained by discretizing the Laplacian on a square with linear finite elements.

For the latter two, the discretization fineness was chosen to yield matrices of dimension $n \approx 20{,}000$, and the associated pencils have several eigenvalues of multiplicity 2.

Figure 4.5 plots the relative errors in the eigenvalues returned by Algorithm 24 and the corresponding residual norms for the problems "APF4686" (left, $n_{\text{ev}} = 30$) and "Kuu/Muu" (right, $n_{\text{ev}} = 100$) for $N = 2, 4, 6, 8$. Figure 4.6 plots the same quantities for "FDmesh" (left) and "FEmesh" (right), where $n_{\text{ev}} = 100$ in both cases. In agreement with the discussion in this section, increasing N leads to greater accuracy in the approximation of the sought eigenpairs. Moreover, all eigenpairs are approximated to comparable accuracies for a given value of N, i.e., the accuracy of an eigenpair is relatively insensitive to the location of the eigenvalue inside $[\alpha, \beta]$.

Distributed memory experiments setup

We now illustrate the distributed memory efficiency of Algorithm 24 on a variety of larger problems coming from discretizations of the Laplacian as well as general symmetric matrices and pencils from the SuiteSparse collection.[3] Unless otherwise indicated, throughout the rest of this section, we take $n_{\text{ev}} = 100$, and we set the second dimension

[1] http://www.elses.jp
[2] https://sparse.tamu.edu/
[3] Our implementation is available publicly at https://github.com/Hitenze/Schurcheb.

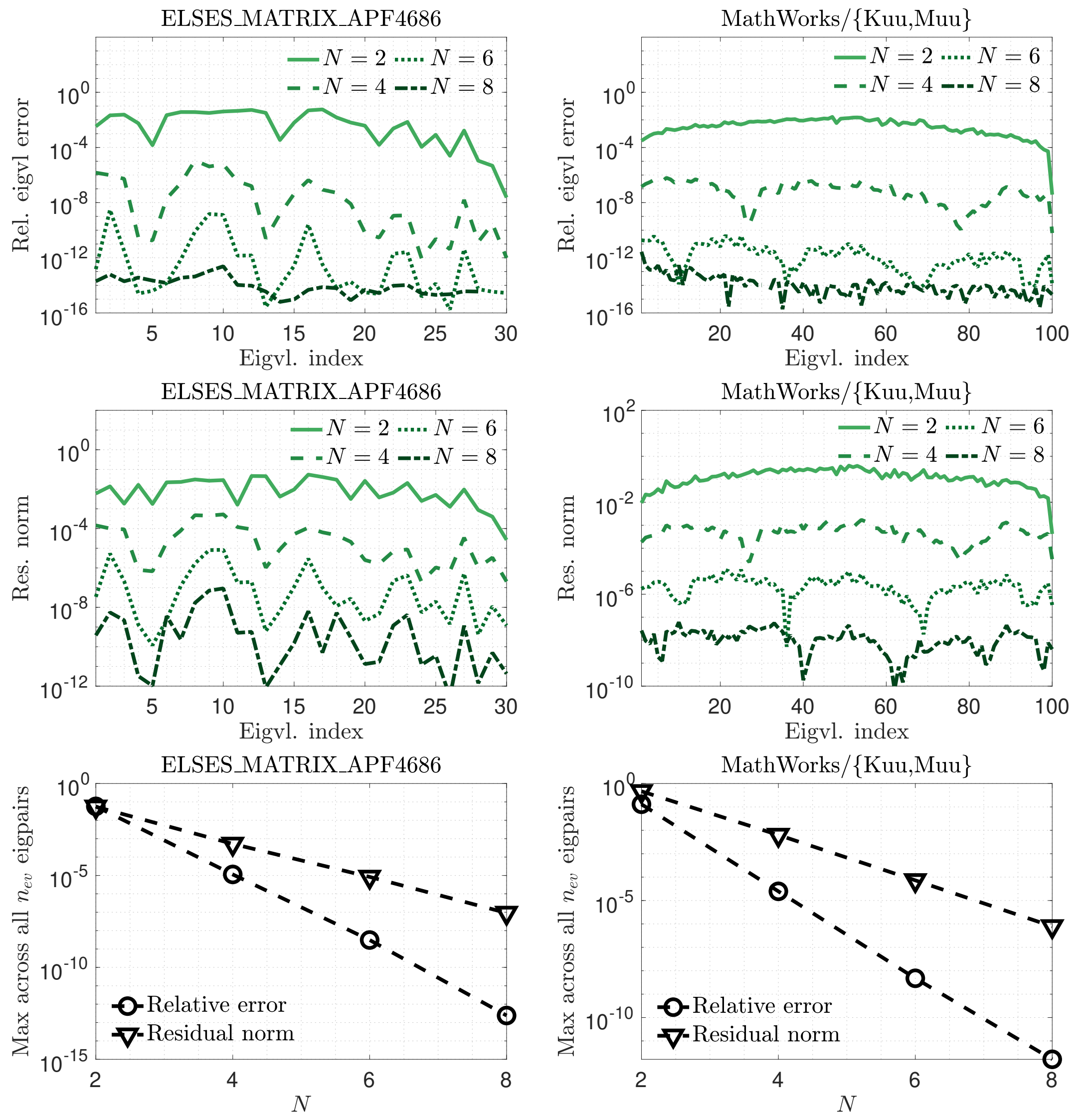


Figure 4.5: Relative errors in the eigenvalues returned by Algorithm 24 (top) and corresponding residual norms (center) for various values of N for the problems "APF4686" (left, $n_{\text{ev}} = 30$) and "Kuu/Muu" (right, $n_{\text{ev}} = 100$). The bottom two figures plot the maximum relative error in the eigenvalues and maximum residual norm across all n_{ev} eigenpairs.

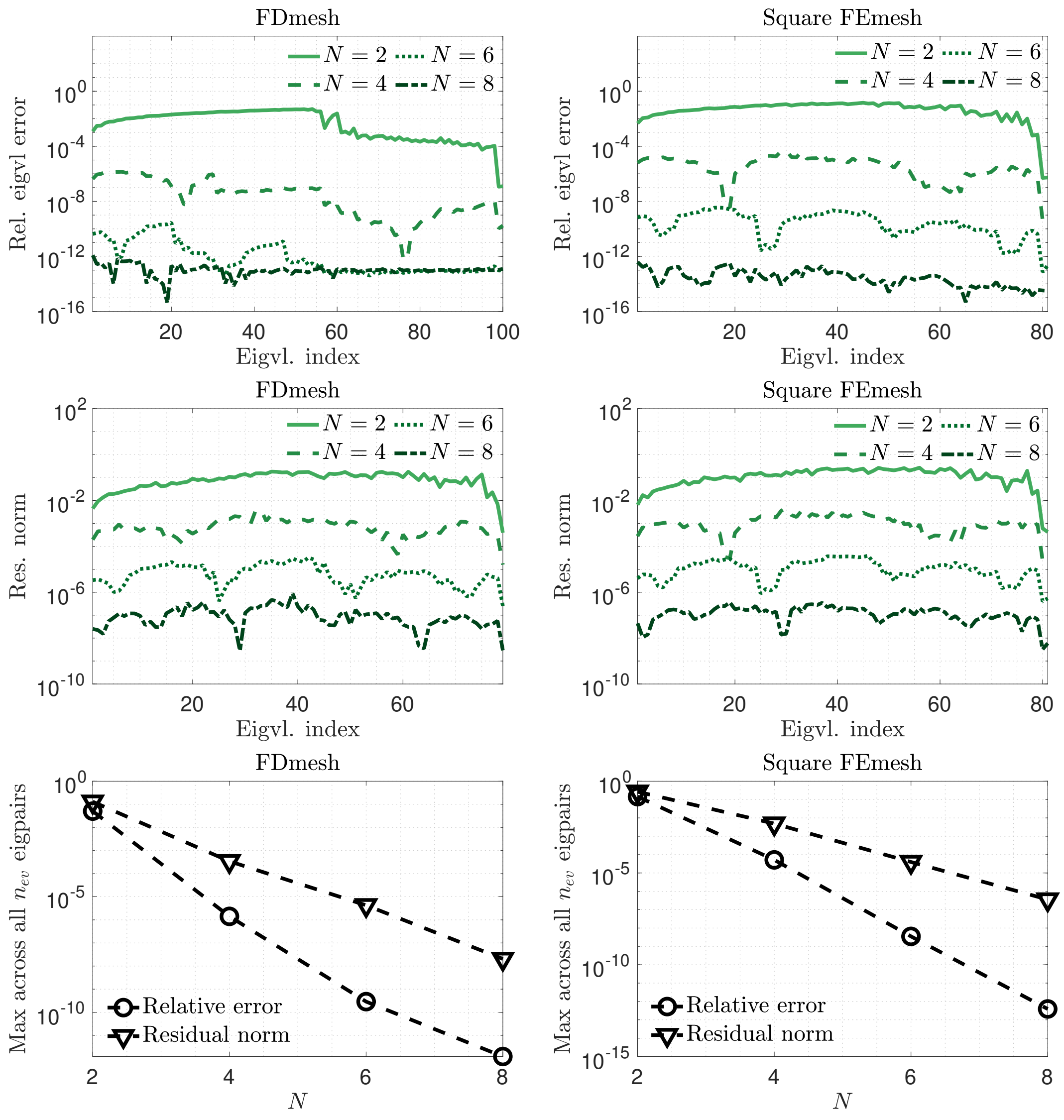


Figure 4.6: Relative errors in the eigenvalues returned by Algorithm 24 (top) and corresponding residual norms (center) for various values of N for the problems "FDmesh" (left) and "FEmesh" (right). The bottom two figures plot the maximum relative error in the eigenvalues and maximum residual norm across all n_{ev} eigenpairs.

of the 2D MPI grid to be $p_c = N$. In most of the tests we report the results with $N = 8$ or $N = 4$. The parallel efficiency of a program executing on $\phi \in \mathbb{N}$ processes is $P(\phi) = T_1/(\phi T_\phi)$, where T_ϕ denotes the wall-clock time for execution on ϕ processes.

We benchmark Algorithm 24 against `PARPACK` applied directly to the pencil (A, M) both with and without shift-and-invert. `PARPACK` requires the application of either M^{-1} (without shift-and-invert) or A^{-1} (with shift-and-invert), and since A and M are distributed, we used a distributed direct solver for these operations. The results reported here were generated using the `MUMPS` package [133], but our code also provides interfaces for `SuperLU_Dist` [134] and the Intel Cluster Sparse Solver (provided in the MKL). For `PARPACK`, we report the wall-clock time and parallel efficiency for a restart length equal to $m = 2n_{\text{ev}}$ with all MPI processes bundled in the default communicator `MPI_COMM_WORLD`. To keep the comparisons fair, the convergence tolerance passed to `PARPACK` for each problem is set to the maximum residual norm returned by Algorithm 24.

Eigenvalue problems from finite difference discretizations

First, we apply Algorithm 24 to matrices arising from finite difference discretizations of the Dirichlet eigenvalue problem,

$$\begin{aligned} -\Delta u &= \lambda u \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{4.19}$$

where Δ denotes the Laplacian and Ω is either the square $(0,1)^2$ in 2D or the cube $(0,1)^3$ in 3D. We use the standard 5- and 7-point stencils in 2D and 3D, respectively. All these eigenvalue problems are standard ones, with M equal to the identity matrix.

Our first set of experiments focuses on the strong scaling of Algorithm 24. We take $n_{\text{ev}} = 100$ and use $N = 4, 8$ Chebyshev nodes. In our results, we refer to Algorithm 24 with $N = 4$ as `SchurCheb(4)` and with $N = 8$ as `SchurCheb(8)`. We first consider three different 2D discretizations with matrix sizes $n = 257 \times 256$, $n = 513 \times 512$, and $n = 1025 \times 1024$, respectively. Table 4.1 lists the maximum relative error in the eigenvalues returned by Algorithm 24. Figure 4.7 (left) plots the parallel efficiency of Algorithm 24 for $N = 8$, where we report separately the parallel efficiencies associated with: (a) computation of the eigenvector matrices Y_j, $j = 0, \dots, N-1$, (b) orthonormalization of

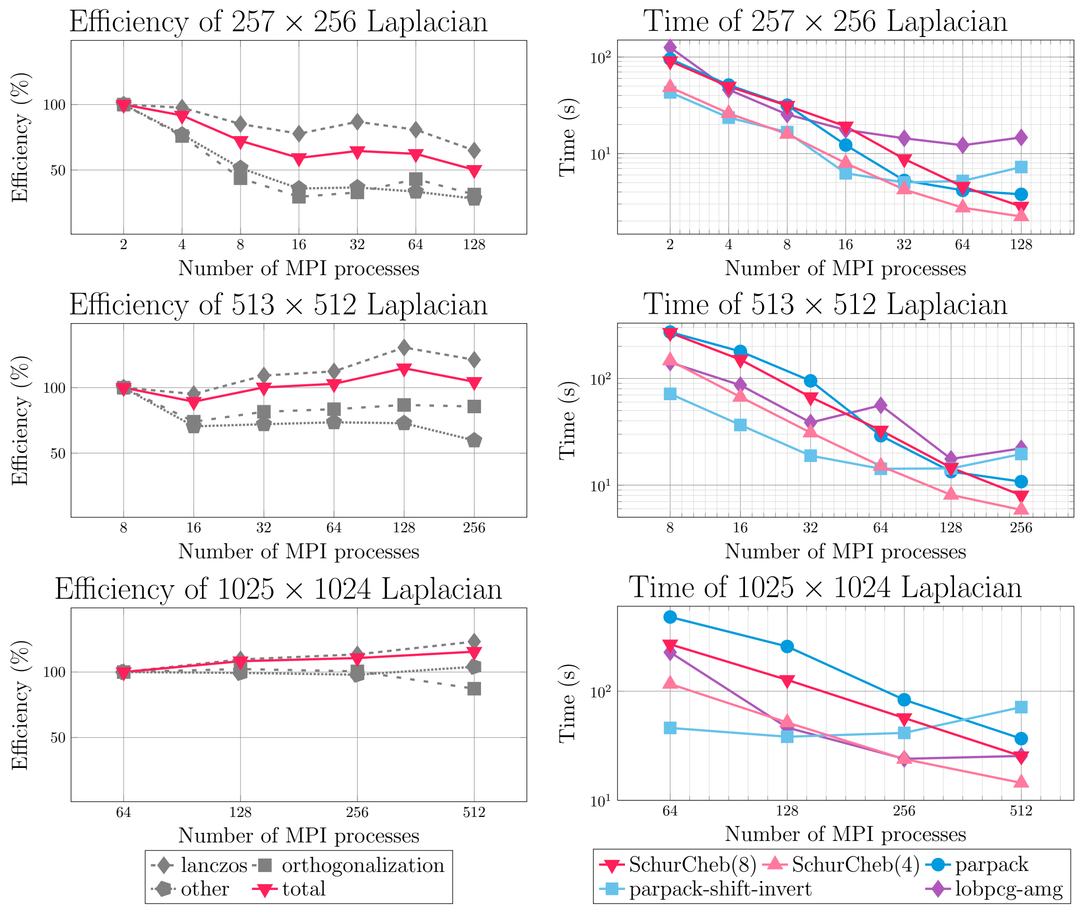


Figure 4.7: Left: parallel efficiency of Algorithm 24 with $n_{\text{ev}} = 100$ and $p_c = N = 8$. Right: wall-clock time comparison between Algorithm 24 with $N = 4, 8$ and PARPACK with and without shift-and-invert. The number of MPI processes ranges from $N_p = 2$ to $N_p = 512$. The number of partitions is set equal to $p = 32$ ($n = 257 \times 256$), $p = 64$ ($n = 513 \times 512$), and $p = 128$ ($n = 1025 \times 1024$), when $N = 8$. The value of p is doubled when $N = 4$ since each column communicator now has twice as many processes.

Table 4.1: Maximum relative error in the eigenvalues returned by Algorithm 24 for the finite difference problems.

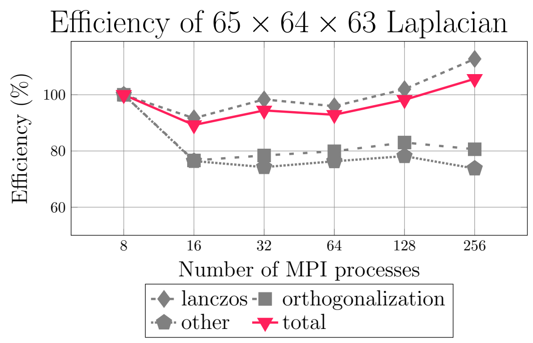
	$n = 257 \times 256$	$n = 513 \times 512$	$n = 1025 \times 1024$	$n = 65 \times 64 \times 63$
`SchurCheb(4)`	5.1×10^{-4}	8.2×10^{-5}	1.4×10^{-4}	9.1×10^{-5}
`SchurCheb(8)`	2.3×10^{-9}	2.9×10^{-11}	2.5×10^{-7}	1.9×10^{-10}

the projection matrix R, and (c) everything else. Since $p_c = N$, the computation of the Y_j is embarrassingly parallel, leading to nearly perfect efficiency for this step. On the other hand, both the orthonormalization of R and the formation of $R^T AR$ require communication among the N_p processes, and their efficiency can deteriorate for larger values of N_p. Note also that the parallel granularity of Algorithm 24 is lower for smaller problem sizes, leading to lower efficiencies compared to larger problems.

Figure 4.7 (right) plots the wall-clock time achieved by Algorithm 24 for $N = 4, 8$, `PARPACK` with and without shift-and-invert, and the Locally Optimal Block Preconditioned Conjugate Gradient (`LOBPCG`) method as implemented in the `BLOPEX` package of `hypre` [130]. The wall-clock times of `LOBPCG` were obtained with AMG preconditioning and we present the best (lowest) times after performing extensive tests involving various choices for the hyperparameters and preconditioners. Regarding the performance of `PARPACK`, note that due to the fact that A comes from a 2D discretization, shift-and-invert is generally very fast when the direct solver scales satisfactorily; however, the efficiency of `MUMPS` falls off faster than that of Algorithm 24 as N_p increases, and for larger values of N_p, Algorithm 24 becomes the fastest and most scalable approach. Similarly, `LOBCPG` is competitive with Algorithm 24 for smaller values of N_p but becomes comparatively slower as N_p increases.

Figure 4.8 plots the same quantities for a 3D discretization matrix of size $n = 65 \times 64 \times 63$. The main difference between the 2D and 3D case is that `PARPACK` without shift-and-invert now converges much faster, leading to lower orthogonalization costs. Moreover, because A is banded, the parallel efficiency of distributed memory sparse matrix-vector products with A remains high even when $N_p = 256$. Nonetheless, Algorithm 24 still attains greater strong scaling efficiency than `PARPACK` (with or without shift-and-invert) and hence will outperform it given enough parallel resources.

As Algorithm 24 does not need to factor A, it requires considerably less storage than



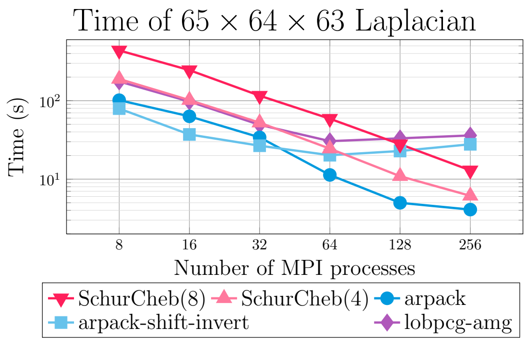


Figure 4.8: Left: parallel efficiency of Algorithm 24 with $n_{\text{ev}} = 100$ and $p_c = N = 8$. Right: wall-clock time comparison between Algorithm 24 with $N = 4, 8$ and `PARPACK` with and without shift-and-invert. The number of MPI processes ranges from $N_p = 8$ to $N_p = 256$. The number of partitions is set to $p = 64$ ($N = 8$) and $p = 128$ ($N = 4$).

Table 4.2: Peak memory consumption of Algorithm 24 and of `PARPACK` with shift-and-invert for the finite difference problems.

	$n = 257 \times 256$ $N_p = 128$	$n = 513 \times 512$ $N_p = 256$	$n = 1025 \times 1024$ $N_p = 512$	$n = 65 \times 64 \times 63$ $N_p = 256$
`SchurCheb(4)`	1.2 GB	2.4 GB	9.3 GB	2.3 GB
`SchurCheb(8)`	2.2 GB	4.6 GB	18.8 GB	4.6 GB
`PARPACK`	21.4 GB	45.0 GB	106.4 GB	46.6 GB

`PARPACK` with shift-and-invert. Table 4.2 lists the global peak memory consumption for both of these algorithms for the finite difference discretization problems just described. Even with $N = 8$ Chebyshev nodes, Algorithm 24 uses 5 to 10 times less memory than shift-and-invert `PARPACK` across all problems.

We now focus on the performance of Algorithm 24 when the problem size n and number of partitions p are fixed and N_p varies proportionally to N. We set $p = p_r = 8$ and $p_c = N$ where $N = 2, 4, \ldots, 16$. For this experiment, we consider the 2D discretizations of sizes $n = 257 \times 256$ and $n = 513 \times 512$ and report the wall-clock times for each major operation of Algorithm 24 in Figure 4.9. The amount of time spent computing the matrices Y_j and V_j is nearly constant since the maximum number of matrix-vector products (iterations) required by `PARPACK` to compute each Y_j, is more or less the same for each N_p (see the solid lines). On the other hand, the amount of time required for orthonormalization and the Rayleigh-Ritz projection both increase due to: (a) higher computational complexity and (b) higher volume of communication among the increasing number of MPI processes.

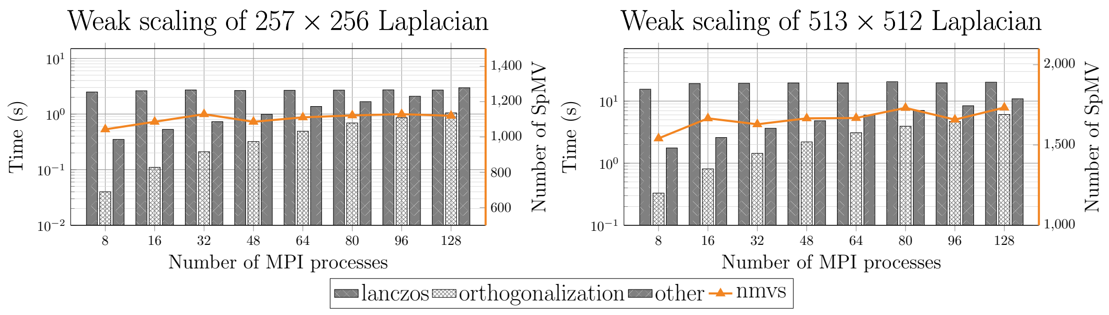


Figure 4.9: Weak scaling with respect to N ($p_r = 8,\ p_c = N$) for two 2D finite difference discretization problems. The number of MPI processes ranges from $N_p = 8$ to $N_p = 128$. The solid red lines denote the maximum number of iterations required by `PARPACK` to compute the matrices $Y_j,\ j = 0, \ldots, N-1$.

Next, we evaluate the performance of Algorithm 24 when computing different numbers of eigenvalues (different n_{ev}) for the same matrix. We consider the 2D discretizations of sizes $n = 257 \times 256$ and $n = 513 \times 512$. In each group of tests, we fix p, p_r, p_c, and N_p and then vary n_{ev}. For the $n = 257 \times 256$ problem, we take $N_p = 128$ and $p_r = N$ and then set $p = 16$ when $N = 8$ and $p = 32$ when $N = 4$. For the $n = 512 \times 512$ problem, we double p and N_p. Figure 4.10 reports the total wall-clock times for Algorithm 24 under these configurations, taking $n_{\text{ev}} = 50, 100, 150, 200$, as well as those for `PARPACK` (with and without shift-and-invert) and `LOBPCG`. The cost of solving the Schur complement eigenvalue problems in Algorithm 24 at each Chebyshev node increases as n_{ev} increases. Nonetheless, Algorithm 24 still attains wall-clock times that are competitive with `PARPACK` and `LOBPCG`.

In the preceding experiments, we took $p_c = N$. As our final experiment in this section, we consider the effect of varying the 2D MPI grid topology. We consider the 2D discretizations of sizes $n = 513{\times}512$. We take $N = 8$, $N_p = p = 128$, $n_{\text{ev}} = 100$, and vary the topology as $(p_r, p_c) = (128, 1), (64, 2), (32, 4), (16, 8)$. Table 4.3 lists a breakdown of the wall-clock times for the various parts of Algorithm 24 for each topology. The topology $(p_r, p_c) = (128, 1)$ processes the N Chebyshev nodes sequentially, one after the other, but uses all N_p MPI processes during the computation of each matrix pair (Y_j, V_j), $j = 0, \ldots, N-1$, taking on average $(26.08 + 0.35)/8 \approx 3.3$ seconds for each. At the other extreme, the topology $(p_r, p_c) = (16, 8)$ processes the N Chebyshev nodes

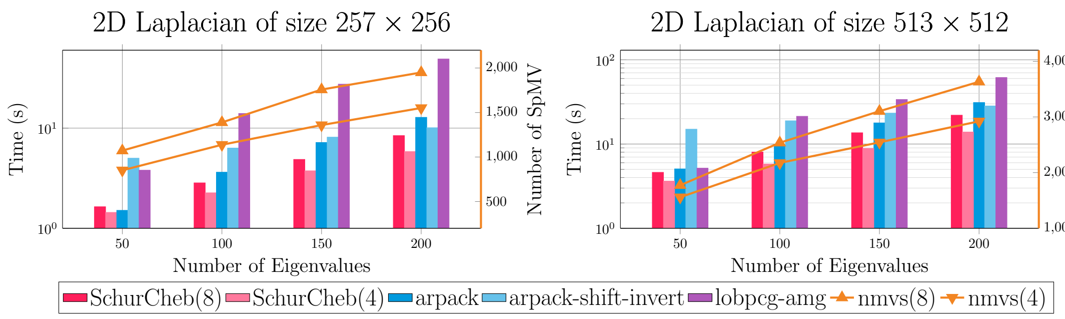


Figure 4.10: Test with different n_{ev} for two 2D finite difference discretization problems. The number of MPI processes are $N_p = 128$ and $N_p = 256$, respectively. The solid red lines denotes the maximum number of iterations required by `PARPACK` to compute the matrices Y_j, $j = 0, \dots, N-1$ in Algorithm 24.

Table 4.3: Wall-clock time breakdown of Algorithm 24 for various 2D MPI grid topologies. (RR: Rayleigh-Ritz, GS: Gram-Schmidt).

(p_r, p_c)	Setup	$Y_{0,\dots,N-1}$	$V_{0,\dots,N-1}$	GS	RR	`DSYGVX`	Total
(128,1)	1.42	26.08	0.35	1.41	1.76	0.14	31.17
(64,2)	0.68	18.06	0.36	1.94	1.81	0.14	23.15
(32,4)	0.32	13.95	0.35	1.71	1.91	0.14	18.41
(16,8)	0.18	13.21	0.35	1.65	2.03	0.14	17.61

completely in parallel, but now computing each (Y_j, V_j) requires more time—in the worst case, approximately 4 times as much ($13.21 + 0.35 = 13.56$ seconds)—since only $p_r = 16$ processes are available for parallelization of those computations. Nevertheless, the total time to solution is nearly halved with $(p_r, p_c) = (16, 8)$ versus $(p_r, p_c) = (128, 1)$. Thus, in agreement with our previous results, setting $p_c = N$ is best unless the smaller value of p_r creates a memory bottleneck.

Eigenvalue problems from finite element discretizations

To illustrate the performance of Algorithm 24 for generalized eigenvalue problems, we again consider matrices arising from discretizations of Equation 4.19 but with linear finite elements instead of finite differences. In 2D, we consider the square $\Omega = (0, 1)^2$ and the disc $\Omega = \{(x, y) \; : \; x^2 + y^2 \leq 1\}$, both meshed with unstructured triangular elements. In 3D, we consider the cube $\Omega = (0, 1)^3$, meshed with unstructured tetrahedra.

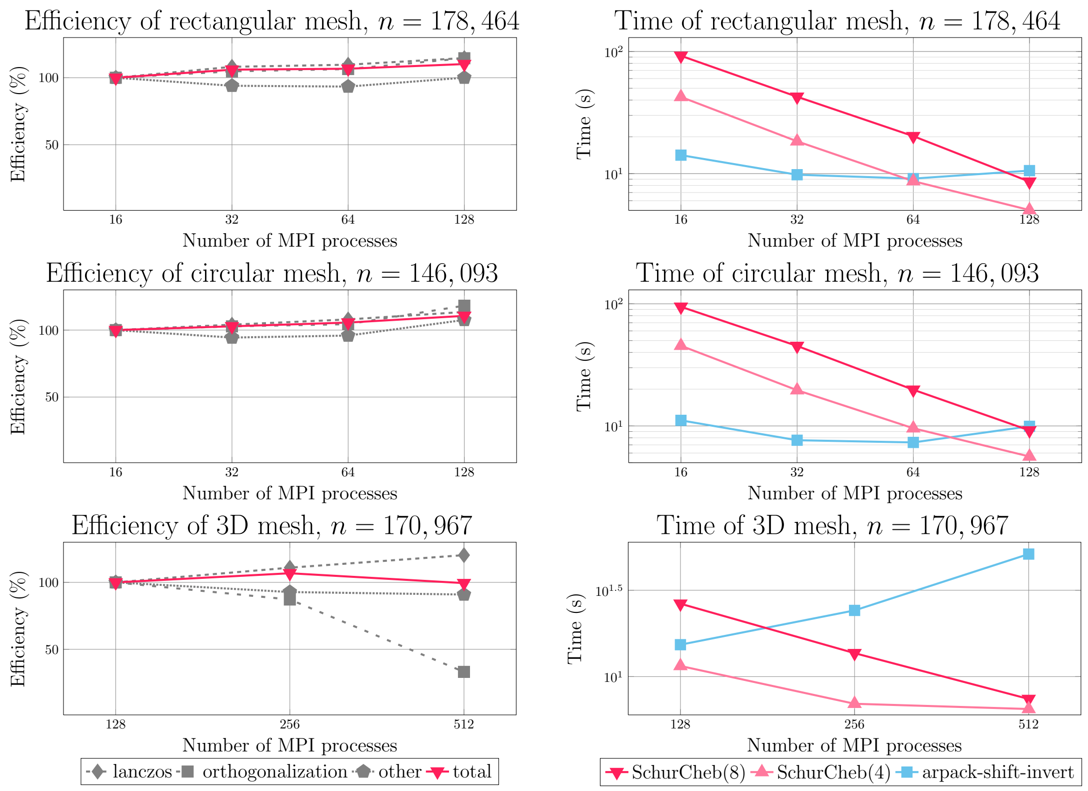


Figure 4.11: Left: parallel efficiency of Algorithm 24 applied to the finite element problems with $n_{\text{ev}} = 100$ and $p_c = N = 8$. Right: wall-clock time comparison between Algorithm 24 with $N = 4$ and $N = 8$, and `PARPACK` with shift-and-invert. The number of MPI processes ranges from $N_p = 8$ to $N_p = 512$. The number of partitions is set equal to $p = 16$ for the 2D meshes and $p = 64$ for the 3D mesh.

Figure 4.11 plots the parallel efficiency of Algorithm 24 (left) and associated wall-clock times as N_p varies. We also plot the wall-clock time of `PARPACK` with shift-and-invert but omit results for `PARPACK` without shift-and-invert, which required an excessive amount of time to converge for these problems. The small sizes of the problems ($n \approx 150{,}000$) have chosen intentionally in order to simulate an environment with an abundance of parallel resources. As in the experiments of the previous section, Algorithm 24 attains high parallel efficiency and scales better than `PARPACK`. The efficiency of the orthogonalization step in Algorithm 24 dropped below 50% for the 3D case when $N_p = 512$ due to a large communication-to-computation ratio for the Gram-Schmidt process; nevertheless, the overall efficiency is still close to 100%.

Next, we show the results of a test similar to one in the previous section, wherein Algorithm 24 is applied to a given problem for increasing values of n_{ev}. As before, we fix p, p_r, p_c, and N_p for each group of tests, and vary n_{ev} as $n_{\text{ev}} = 50, 100, 150, 200$. We use the same finite element problems of the previous experiment set $p_c = N$. When $N = 8$, we use $N_p = 128$ and $p = 16$ for the 2D domains and $N_p = 512$ and $p = 64$ for the 3D domains. When $N = 4$, we double p. The results are reported in Figure 4.12. Again, Algorithm 24 attains times to solution that are competitive with `PARPACK`, even though the cost of solving the local eigenvalue problems at each Chebyshev node increases with n_{ev}.

Finally, Table 4.4 lists the wall-clock times for Algorithm 24 and `PARPACK` with shift-and-invert on a set of larger finite element problems. For Algorithm 24 we report the wall-clock times for the case $N_p = 512$ and $p_c = N = 4$; for `PARPACK`, we report the best (lowest) wall-clock time obtained over several runs with different N_p. Algorithm 24 was twice as fast for the 2D problems, and about as fast as `PARPACK` for the 3D problem. Note, though, that in addition to having superior[4] scalability, Algorithm 24 also uses much less memory.

Eigenvalue problems from the SuiteSparse collection

Finally, to demonstrate the performance of Algorithm 24 for more general matrices, we apply it to several problems taken from the SuiteSparse matrix collection with sizes ranging from $n = 66,172$ to $n = 1,222,045$. Additional details are given in Table 4.5.

[4] The best wall-clock time of `PARPACK` for the 3D mesh problem was achieved for $N_p = 128$.

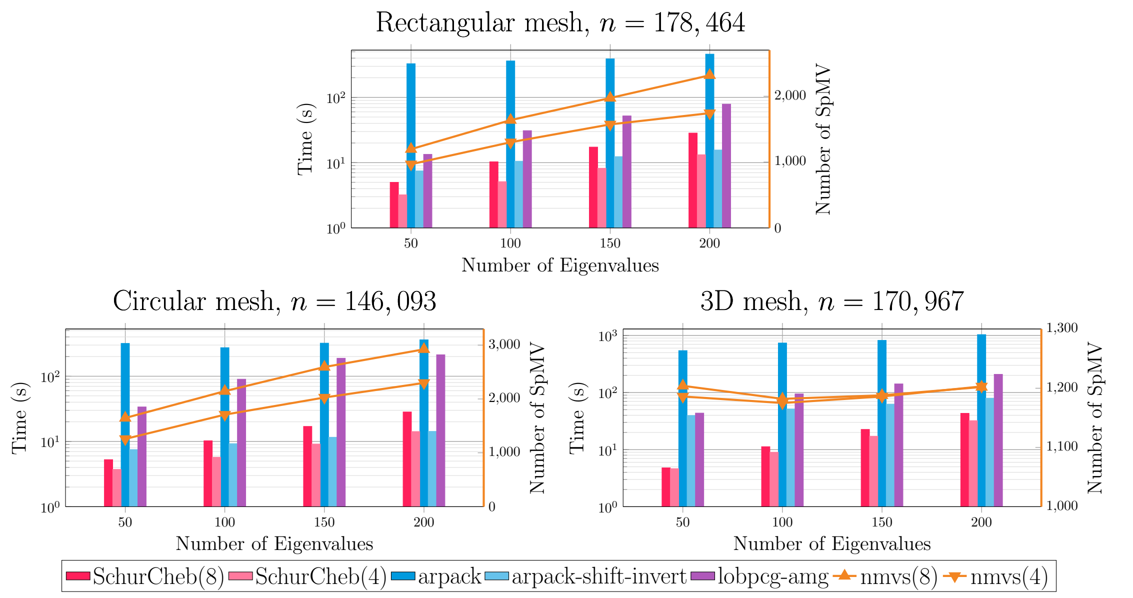


Figure 4.12: Test with different n_{ev} for three finite element problems. The numbers of MPI processes are $N_p = 128$ for the 2D domains and $N_p = 512$ for the 3D domain. The solid red lines denotes the maximum number of iterations required by `PARPACK` to compute the matrices Y_j, $j = 0, \ldots, N-1$. in Algorithm 24.

The "qa8fk/qa8fm" problem is a generalized eigenvalue problem; the other four are standard problems (M is the identity matrix).

Figure 4.13 plots the parallel efficiency (left) and wall-clock time (right) for Algorithm 24 on each of these problems. For comparison, we also plot the wall-clock time of `PARPACK` with and without shift-and-invert. As in the previous experiments, Algorithm 24 maintains high parallel efficiency up to 512 MPI processes, and, provided enough parallel resources, outperforms `PARPACK`. Additionally, Algorithm 24 is more memory efficient than shift-and-invert `PARPACK` as N_p increases; Table 4.6 lists the peak memory consumption for both algorithms for the maximum N_p used in each group of tests for each problem. Finally, Table 4.7 lists the maximum error in the eigenvalues returned by Algorithm 24 for $N = 4$ and $N = 8$.

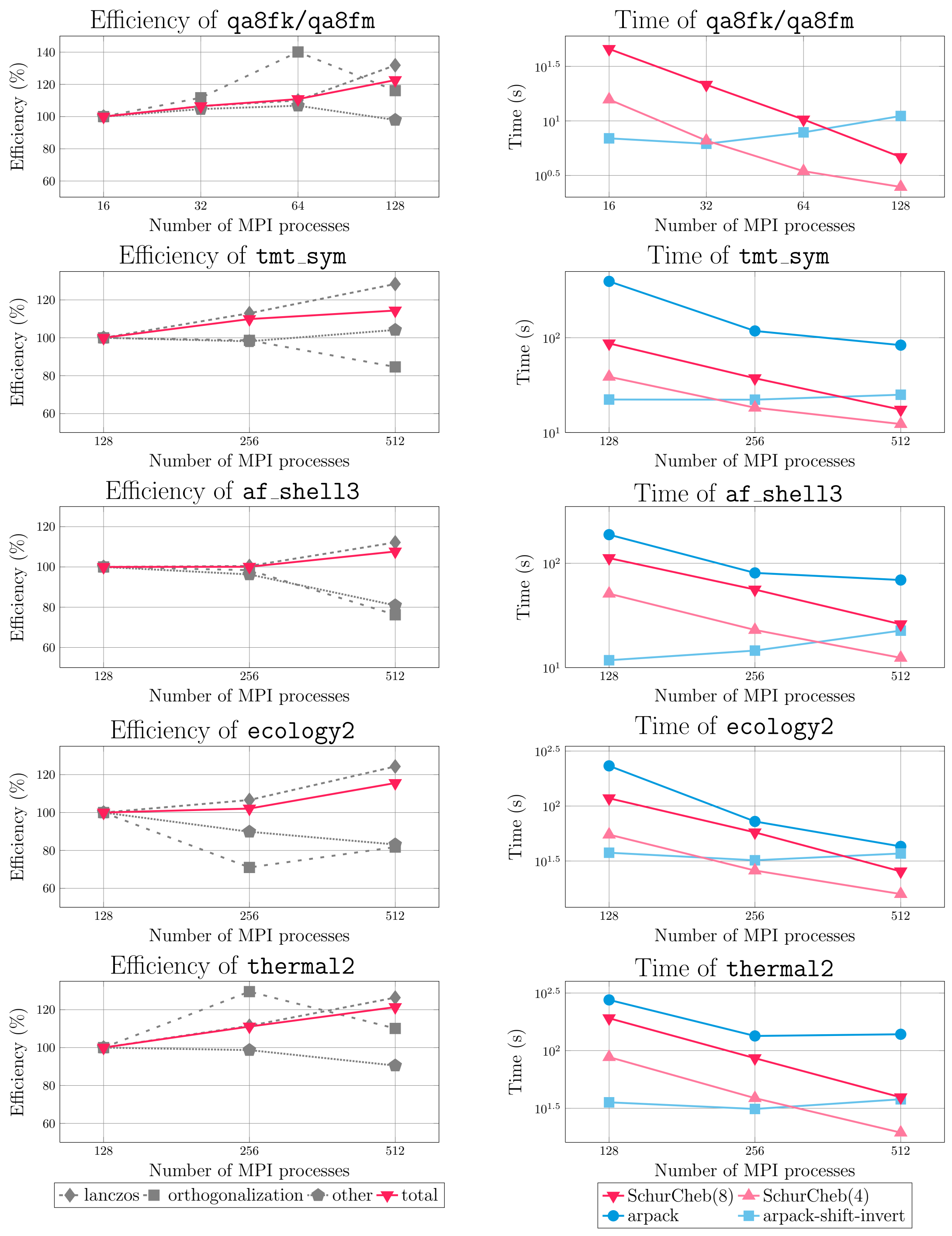


Figure 4.13: Left: parallel efficiency of Algorithm 24 with $n_{\text{ev}} = 100$ and $p_c = N = 8$. Right: wall-clock time comparison between Algorithm 24 with $N = 4$ and $N = 8$, and `PARPACK` with and without shift-and-invert. The number of MPI processes ranges from $N_p = 16$ to $N_p = 512$.

Table 4.4: Total wall-clock time for Algorithm 24 and `PARPACK` with shift-and-invert for the finite element problems with $N_p = 512$, $p = 128$, and $p_c = N$.

	2D square $n = 1,086,615$	2D disc $n = 845,397$	3D cube $n = 1,351,083$
`SchurCheb(4)`	17.2 s	18.3 s	90.1 s
`PARPACK`	33.6 s	25.9 s	90.3 s

Table 4.5: Problems from the SuiteSparse matrix collection. Here, n denotes the size of the pencil (A, M); nnz(.): counts the number of nonzero entries in its arguments; and p denotes the number of partitions for the case $N = 8$.

Dataset	n	p	nnz$(A)/n$	nnz$(M)/n$	Application
qa8fk/qa8fm	66,172	16	25.1	25.1	3D acoustics
af_shell3	504,855	64	34.8	1.0	structural problem
tmt_sym	726,713	64	6.99	1.0	electromagnetics
ecology2	999,999	64	5.00	1.0	2D/3D problem
thermal2	1,228,045	64	6.99	1.0	thermal problem

4.2.4 Conclusion

This section presents a distributed memory Rayleigh-Ritz projection algorithm to compute a few of the smallest eigenvalues and associated eigenvectors of a sparse, symmetric matrix pencil. The algorithm introduces embarrassing parallelism by recasting the problem as one of approximating univariate, vector-valued functions via Chebyshev approximation. Our experiments demonstrated that the proposed algorithm attains good parallel efficiency, superior to `PARPACK`.

In the future, we plan to develop a version of this algorithm based on generalized spectral Schur complements, in which the matrix Y_j is formed by computing a few eigenvectors of the pencil $\left(S(\chi_j), -S'(\chi_j)\right)$ instead of $S(\chi_j)$ alone. This may allow one to reduce the value of N, permitting the use of more parallel resources within each column MPI communicator. We also plan on extending the implementation of our current algorithm so that the computations local to each MPI process are performed using graphics processing units. Finally, we plan on applying our software to problems from real-world applications, e.g., frequency response analysis.

Table 4.6: Peak memory consumption of Algorithm 24 and of `PARPACK` with shift-and-invert for the SuiteSparse problems.

	qa8	af_shell3	tmt_sym	ecology2	thermal2
	$N_p = 128$	$N_p = 512$	$N_p = 512$	$N_p = 512$	$N_p = 512$
`SchurCheb(4)`	0.7 GB	5.9 GB	6.7 GB	8.9 GB	11.2 GB
`SchurCheb(8)`	1.4 GB	11.9 GB	13.2 GB	17.5 GB	22.2 GB
`PARPACK`	21.7 GB	47.7 GB	50.8 GB	58.7 GB	56.5 GB

Table 4.7: Maximum relative error in the eigenvalues returned by Algorithm 24 for the SuiteSparse problems.

	qa8	af_shell3	tmt_sym	ecology2	thermal2
`SchurCheb(4)`	3.2×10^{-4}	2.1×10^{-4}	1.6×10^{-4}	1.8×10^{-5}	9.1×10^{-5}
`SchurCheb(8)`	1.0×10^{-8}	3.8×10^{-10}	6.5×10^{-8}	8.9×10^{-9}	1.9×10^{-10}

4.3 A Parallel Block Givens QR Decomposition Algorithm

In the previous section, we explored a parallel eigenvalue solver designed specifically for computing a subset of the smallest eigenvalues in generalized eigenvalue problems. Now, let's shift our focus to Step 9 of Algorithm 24, which necessitates the orthonormalization of a matrix $R \in \mathbb{R}^{m \times n}$. In this final section of the chapter, we will present an algorithm that leverages block Givens rotation for efficient computation of the QR decomposition. This algorithm provides an effective means of accomplishing the orthonormalization process required in Step 9. By employing block Givens rotation, we can enhance the computational efficiency and numerical stability of the Givens QR decomposition. Through the subsequent analysis and evaluation of our proposed algorithm, we aim to demonstrate its effectiveness compared to Givens QR. Our proposed algorithm is also suitable for parallel computing. We will discuss both OpenMP and CUDA acceleration of our algorithm.

4.3.1 QR decomposition

QR decomposition is a fundamental matrix factorization technique that is crucial in numerous computational applications. In this section, we focus on the QR decomposition of a matrix $A \in \mathbb{R}^{m \times n}$, assuming that $m \geq n$. However, it's worth noting that our proposed algorithm can also easily be applied to matrices with $m < n$.

QR decomposition computes a unitary matrix $Q \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $R \in \mathbb{R}^{m\times n}$ such that $QR = A$. In some scenarios, it is more efficient and practical to compute a thin QR decomposition. The thin QR decomposition yields a matrix $Q \in \mathbb{R}^{m\times n}$ with orthonormal columns and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $QR = A$. This variant is particularly useful when the full unitary matrix Q is not required. In the following discussion, we explore several commonly used QR decomposition algorithms and examine their respective computational costs [135, 136, 137].

Gram-Schmidt QR

We will first discuss the application of the Gram-Schmidt process to compute the QR decomposition. Specifically, our focus lies on two main algorithms: the classical Gram-Schmidt algorithm and the modified Gram-Schmidt QR algorithm. These algorithms serve as the foundation for other similar Gram-Schmidt QR variants. Both the classical Gram-Schmidt algorithm and the modified Gram-Schmidt QR algorithm aim to compute the matrices Q and R column by column, utilizing the Gram-Schmidt process. To illustrate the steps of the modified Gram-Schmidt QR algorithm, we present Algorithm 25.

Algorithm 25 Modified Gram-Schmidt QR

```
1: for $i = 1$ to $n - 1$ do
2:     $\mathbf{q}_i \leftarrow \mathbf{a}_i$
3:     for $j = 1$ to $i - 1$ do
4:         $r_{i,j} \leftarrow (\mathbf{q}_i, \mathbf{q}_j)$
5:         $\mathbf{q}_i \leftarrow \mathbf{q}_i - r_{i,j}\mathbf{q}_j$
6:     end for
7:     $r_{i,i} \leftarrow (\mathbf{q}_i, \mathbf{q}_i)$
8:     $\mathbf{q}_i \leftarrow \mathbf{q}_i / r_{i,i}$          ▷ Assume $r_{i,i} \neq 0$
9: end for
```

The computation of modified Gram-Schmidt QR requires a large number of synchronization steps when computing the inner product, which harms its parallel performance on distributed memory machines [137]. On the other hand, the classical Gram-Schmidt QR algorithm is less accurate, but only one synchronization step is required for computing each column of Q. However, an additional re-orthogonalization step might be necessary to achieve the desired factorization accuracy, incurring nearly twice the computational cost of the algorithm. Although having several issues, Gram-Schmidt QR algorithms are still widely used due to their simplicity in programming.

Householder QR

Householder QR is another approach for computing the QR decomposition, employing a sequence of matrix-vector multiplications. The algorithm applies a sequence of n unitary rank one updates in the form $P_i := I - \beta_i v_i v_i^H$ to transform matrix A into an upper triangular form. Each update P_i only affects the submatrix $A(i : m, i : n)$ (in `MATLAB` notation). The QR decomposition can then be expressed as:

$$A = \underbrace{(P_n \dots P_2 P_1)^T}_{Q} \underbrace{(P_n \dots P_2 P_1) A}_{R}. \tag{4.20}$$

The Householder QR algorithm is outlined in Algorithm 26. It's important to note that the algorithm does not provide an explicit formula for Q. To compute Q explicitly, the multiplication $\prod_{i=n}^{1}(I - \beta_i \mathbf{v}_i \mathbf{v}_i^H)$ must be performed. When computing the economic size QR, storing the P_i matrices is necessary, and multiplication should be performed in reverse order, starting from P_n, especially when $m \gg n$.

Algorithm 26 Householder QR

```
1: for $i = 1$ to $n$ do
2:     $(\mathbf{v}_i, \beta_i)$ =house($A(i : m, i : n)$)        ▷ MATLAB notation
3:     $A(i : m, i : n) \leftarrow (I - \beta_i \mathbf{v}_i \mathbf{v}_i^H) A(i : m, i : n)$
4: end for
```

Efficiency improvements can be achieved by utilizing the block Householder QR algorithm [138, 137]. The nested blocking strategy, used in the `LAPACK` routine `dgeqrf`, can further enhance performance. By combining the first several reflectors together as $(I - V_1 W_1^H) := \prod_{i=1}^{r}(I - \beta_i \mathbf{v}_i \mathbf{v}_i^H)$ before updating $A(:, r + 1 : n)$, the block Householder QR algorithm introduces minor overhead while significantly enhancing parallel performance, particularly on GPUs.

Givens QR

The Givens QR algorithm is another commonly used method for computing the QR decomposition. It is based on Givens rotations, which perform a rotation to zero out individual entries in the matrix:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} h \\ g \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}. \tag{4.21}$$

One way to implement the Givens QR algorithm is to apply these rotations column by column, starting from the first column to zero-out entries. However, alternative strategies can be employed to improve the parallel efficiency of the Givens QR algorithm [139]. Additionally, the Givens QR algorithm can be extended to a block version by generalizing the rotation through block QR decomposition [136]. In this case, a QR factorization is applied to eliminate blocks instead of individual entries, following a similar procedure to the standard Givens QR. The block version can be expressed as:

$$Q^T \begin{pmatrix} H \\ G \end{pmatrix} = \begin{pmatrix} R \\ O \end{pmatrix} \tag{4.22}$$

If the matrix A is partitioned into $m_r \times n_r$ blocks, the QR factorization can be used to eliminate one block at a time. Starting from the first block column, the elimination process can be performed in a similar manner to the standard Givens QR.

To conclude this section, we compare the computation costs of different QR algorithms for computing the economic size QR factorization. Here is a summary of the total FLOP count for each algorithm. The cost of the modified Gram-Schmidt QR is $2mn^2$, and the modified Gram-Schmidt QR always forms the Q and R. The cost of each update is $4m$ while there is a total of $n^2/2$ updates. If we use classical Gram-Schmidt QR with one re-orthogonalization step, the total cost will increase to $4mn^2$. The cost of the (block) Householder QR is $2mn^2 - 2n^2/3$ if we do not form Q explicitly. The cost will be doubled to $4mn^2 - 4n^2/3$ if Q is explicitly wanted and the cost of block Householder QR is similar. The cost of the (block) Givens QR is twice that of the (block) Householder QR.

4.3.2 A block Givens QR algorithm

As discussed in the previous section, the cost of block Givens QR is much higher than block Householder QR. In this section, we proposed a new block Givens QR algorithm that compresses the rotation matrix, which could reduce the computation cost.

Consider the problem of finding a matrix P such that

$$P \begin{pmatrix} H \\ G \end{pmatrix} = \begin{pmatrix} X \\ O \end{pmatrix} \tag{4.23}$$

where $P \in \mathbb{R}^{m\times m}$, $H \in \mathbb{R}^{m_1\times m_1}$, and $G \in \mathbb{R}^{m_2\times m_1}$ with $m := m_1 + m_2$. In this manuscript, we construct P using the generalized SVD (GSVD). The existence of generalized SVD is already proven under a more general condition with $H \in \mathbb{R}^{m_1\times n}$ and $G \in \mathbb{R}^{m_2\times n}$ with the same number of columns. We only need to consider the case where $n = m_1$. Thus, we only discuss the GSVD with $m_2 \geq n \geq m_1$ to simply our discussion. In the case where $m_2 < m_1$ we can swap H and G when computing the GSVD. The result under this constraint is shown in the following theorem.

Theorem 4.3.1. *Given two matrices $H \in \mathbb{R}^{m_1\times n}$ and $G \in \mathbb{R}^{m_2\times n}$ with the same number of columns with $m_2 \geq n \geq m_1$. There exist two diagonal matrices $C \in \mathbb{R}^{m_1\times n}$ and $S \in \mathbb{R}^{m_2\times n}$; and two unitary matrices $U \in \mathbb{R}^{m_1\times m_1}$ and $V \in \mathbb{R}^{m_2\times m_2}$; and a matrix $X \in \mathbb{R}^{n\times n}$ such that*

$$H = UCX^T \tag{4.24}$$

$$G = VSX^T. \tag{4.25}$$

In addition, $C^TC + S^TS = I_n$.

Proof. Given that $m := m_1 + m_2 > n$, via computing the economic size QR decomposition, we can have

$$QR = \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} R = \begin{pmatrix} H \\ G \end{pmatrix}, \tag{4.26}$$

where $Q_1 \in \mathbb{R}^{m_1\times n}$, $Q_2 \in \mathbb{R}^{m_2\times n}$, and $R \in \mathbb{R}^{n\times n}$. With the SVD decomposition UCW^T of Q_1, we can write $H = UC(R^TW)^T$ with the diagonal of C sorted in descending order.

Define $X := R^T W$, we have $H = UCX^T$. Next, we hope find V and S to write G as $G = VSX^T$.

Since the singular values of Q_1 are less than or equal to one, assume C has l entries equal to one, we can write C and $C^T C$ in the form

$$C = \begin{pmatrix} I_l & O & O \\ O & \hat{C} & O \end{pmatrix}, C^T C = \begin{pmatrix} I_l & & \\ & \hat{C}^T \hat{C} & \\ & & O_{n-m_1} \end{pmatrix}. \tag{4.27}$$

With the constrain $m_2 \geq n$, we know that there exists a unique diagonal matrix $S \in \mathbb{R}^{m_2 \times n}$ with non-negative diagonal entries such that $C^T C + S^T S = I_n$. We can write S and $S^T S$ in the form

$$S = \begin{pmatrix} O_l & O \\ O & \hat{S} \\ O & O \end{pmatrix}, S^T S = \begin{pmatrix} O_l & \\ & \hat{S}^T \hat{S} \end{pmatrix}, \tag{4.28}$$

where $\hat{S}$ has a block structure

$$\hat{S} = \begin{pmatrix} \hat{S}_{11} & \\ & I_{n-m_1} \end{pmatrix}. \tag{4.29}$$

We can also write Q_2 and W in block form as

$$Q_2 = \begin{pmatrix} O_{m_2 \times l} & \hat{Q}_2 \end{pmatrix}, W = \begin{pmatrix} W_{11} & \\ & W_{22} \end{pmatrix}, \tag{4.30}$$

where $W_{11} \in \mathbb{R}^{l \times l}$ and $O_{m_2 \times l}$ is a zero matrix of size $m_2 \times l$.

Define $\hat{V} := \hat{Q}_2 W_{22} \hat{S}^{-1}$, we have $\hat{V}^T \hat{V} = I_{n-l}$. This is because we can write $W^T Q_2^T Q_2 W = I - W^T Q_1^T Q_1 W = S^T S$. Using the block structure of Q_2 and W we know $W_{22}^T \hat{Q}_2^T \hat{Q}_2 W_{22} = \hat{S}^T \hat{S}$. Thus, we can expand $\hat{V}$ to a unitary matrix V by adding l basis vectors on its left and $m_2 - n$ basis vectors on its right as

$$V = \begin{pmatrix} \hat{V}_l & \hat{V} & \hat{V}_r \end{pmatrix} \tag{4.31}$$

We can easily verify that $VS = QW$, and thus $VSX^T = Q_2R = G$. Since we can always find those U, V, C, S, and X matrices, the GSVD stated in the theorem always exists. □

The proof of the theorem above gives a GSVD algorithm, which is summarized in 27.

Algorithm 27 Generalized SVD ($m_2 \geq n \geq m_1$)

```
1: Compute the economic QR decomposition $\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} R = \begin{pmatrix} H \\ G \end{pmatrix}$
2: Compute SVD decomposition $UCW^T = Q_1$, the diagonal entries of $C$ are ordered
   in descending order.
3: Compute $X \leftarrow R^T W$
4: Compute matrix $S$
5: Compute $\hat{V} \leftarrow \hat{Q}_2 W_{22} \hat{S}^{-1}$
6: Expand $\hat{V}$ into unitary matrix $V$
```

With the above theorem, we can start constructing the block version of the Givens rotation. When $m_1 = m_2$, C and S are both square matrices, we can define the rotation matrix in the form

$$T = \begin{pmatrix} C & S^T \\ -S & C \end{pmatrix} \begin{pmatrix} U^T & \\ & V^T \end{pmatrix}. \tag{4.32}$$

We can easily verify that applying T can eliminate the G block since $-SU^TUCX^T + CV^TVSX^T = O$.

We hope to generalize this into the case where $m_1 \neq m_2$. This can be done by replacing the matrix T with a new matrix

$$T_0 = \begin{pmatrix} C & S^T \\ -S & C_h \end{pmatrix} \begin{pmatrix} U^T & \\ & V^T \end{pmatrix}, \tag{4.33}$$

where C_h is a matrix of size $m_2 \times m_2$. To eliminate the G block, we need to enforce $SC = C_hS$.

There are two different possible relationships between m_1 and m_2, and we are going to discuss them one by one.

- $m_1 > m_2$. In this case, C is larger than C_h. Since S is a diagonal matrix in

$\mathbb{R}^{m_2 \times m_1}$, we can simply define C_h as the supmatrix $C(1:m_1, 1:m_1)$.

- $m_1 < m_2$. In this case, C is smaller than C_h. We can simply expand C to C_h as

$$C_h = \begin{pmatrix} C & \\ & I \end{pmatrix}. \tag{4.34}$$

By constructing C_h in this way, we have the following lemma.

Lemma 4.3.2. *If we construct C_h using the above approach, we have*

1. $SC = C_h S$.
2. $S^T S + C^T C = I$.
3. $SS^T + C_h C_h^T = I$.

Computing block Givens rotation in this way is much cheaper. Consider the case when $m_1 = m_2 = r$. The block Givens QR using Equation 4.22, the rotation matrix Q is a dense $2r \times 2r$ matrix. On the other hand, our proposed algorithm explore structure within the rotation matrix, and the rotation matrix consists of two $r \times r$ dense matrix and two $r \times r$ diagonal matrix. The matrix-matrix multiplication with our rotation is much cheaper. By using our formula, we can approximately halve the cost of the QR decomposition compared to other Givens QR approach when the block size is not too small.

4.3.3 Implementation details

In this section, we delve into the practical implementation of our proposed block Givens QR algorithm. We present implementations for both the CPU and GPU architectures, utilizing OpenMP and CUDA, respectively.

Generalized SVD

One of the key components of our proposed block Givens QR algorithm is the computation of the GSVD as shown in Algorithm 27. In this section, we assume $m_1 = m_2 = r$ for simplicity, which is used in both our CPU and GPU implementations. For matrices

H and G with insufficient size, we fill them with zeros to obtain $r \times r$ matrices before computation.

Although the existence of GSVD is proven in Algorithm 27, the result might be inaccurate if steps 4-5 are not handled properly. When H and G are square matrices, a naive approach to compute S is to use the formula $S = \sqrt{I - C^2}$. However, if a diagonal entry $C_{i,i}$ of C is close to one, $1 - C_{i,i}^2$ will be close to zero, leading to small values for the corresponding $S_{i,i}$. Consequently, when computing $\hat{V}$ using $\hat{Q}_2 W_{22} \hat{S}^{-1}$, the ith column of $\hat{V}$ is likely to have significant numerical errors due to division by a small value. Therefore, it is crucial to handle large and small entries on the diagonal of C differently.

In our implementation, we adopt a strategy similar to the one used in the `gsvd` routine in `MATLAB`. The first step is to compute the QR decomposition

$$\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} R = \begin{pmatrix} H \\ G \end{pmatrix}, \tag{4.35}$$

followed by the SVD decomposition $Q_1 = UCW^T$. Here Q_1, Q_2, R, U, C, and W are all $r \times r$ matrices. We assume that the diagonal entries of C are ordered in ascending order, and the SVD can be written in the block form:

$$Q_1 = \begin{pmatrix} U_1 & \hat{U}_2 \end{pmatrix} \begin{pmatrix} C_1 & \\ & \hat{C}_2 \end{pmatrix} \begin{pmatrix} W_1^T \\ \hat{W}_2^T \end{pmatrix}, \tag{4.36}$$

where $C_1 \in \mathbb{R}^{r_1 \times r_1}$ and $\hat{C}_2 \in \mathbb{R}^{r_2 \times r_2}$ with $r_1 + r_2 = r$. Here, r_1 represents the total number of $C_{i,i}$ values that are less than $\sqrt{2}/2$, and r_2 represents the total number of $C_{i,i}$ values that are greater than or equal to $\sqrt{2}/2$. We can easily verify that for $i \leq r_1$, $S_{i,i} > C_{i,i}$. Note that the hat symbol is used on matrices that are subject to update during later steps.

Next, we compute the matrix $Z := Q_2 W$. According to Theorem 4.3.1, we have $Z = VS$, and we can write Z in block form as follows:

$$Z = \begin{pmatrix} V_1 & V_2 \end{pmatrix} \begin{pmatrix} S_1 & \\ & S_2 \end{pmatrix}. \tag{4.37}$$

To avoid errors introduced by S_2, instead of computing S first, we perform a full QR decomposition of the first r_1 columns of Z corresponding to the large $S_{i,i}$ values. This yields an alternative decomposition of Z:

$$Z = \begin{pmatrix} \hat{V}_1 & \hat{V}_2 \end{pmatrix} \begin{pmatrix} \hat{S}_1 & \hat{S}_{1,2} \\ & \hat{S}_2 \end{pmatrix}. \tag{4.38}$$

Since Z has orthogonal columns and S_1 corresponds to the numerically stable part, we can discard the off-diagonal entries of $\hat{S}1$ and the off-diagonal block $\hat{S}1,2$, resulting in:

$$Z = \begin{pmatrix} V_1 & \hat{V}_2 \end{pmatrix} \begin{pmatrix} S_1 & \\ & \hat{S}_2 \end{pmatrix}. \tag{4.39}$$

The next step is to compute an SVD decomposition $\hat{S}_2 = U_S S_2 V_S^T$. The final V and S are then

$$V = \begin{pmatrix} V_1 & V_2 \end{pmatrix} := \begin{pmatrix} V_1 & \hat{V}_2 U_S^T \end{pmatrix} \quad \text{and} \quad S = \begin{pmatrix} S_1 & \\ & S_2 \end{pmatrix}. \tag{4.40}$$

The right singular vectors V_S of $\hat{S}_2$ is then used to update U, W, and C. The updated matrix W is given by:

$$W = \begin{pmatrix} W_1 & W_2 \end{pmatrix} := \begin{pmatrix} W_1 & \hat{W}_2 V_S \end{pmatrix}. \tag{4.41}$$

We then use the updated matrix W to update matrices U and C. By performing a QR decomposition $Q_C R_C = \hat{C}_2 V_S$, we obtain C_2 by dropping all off-diagonal entries from the R_C matrix. Additionally, we update $\hat{U}_2$ using Q_C to obtain U_2.

We summarize the key components of the implementation in Algorithm 28.

In our CPU implementation of the GSVD, we have developed a simple sequential version using the QR and SVD routines provided by `LAPACK`. To explore multi-core parallelism, we perform multiple GSVDs concurrently, which will be discussed in detail later.

For the GPU implementation, we need to exploit parallelism within the GSVD computation further to leverage the many-core processors effectively. We perform GSVD on a batch of small matrices of the same size with multiple threads working on one

Algorithm 28 Generalized SVD of $r \times r$ matrices

```
1: Compute the economic QR decomposition $\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} R = \begin{pmatrix} H \\ G \end{pmatrix}$
2: Compute SVD decomposition
       $$Q_1 = UCW^T = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} C_1 & \\ & C_2 \end{pmatrix} \begin{pmatrix} W_1^T \\ W_2^T \end{pmatrix},$$
   the diagonal entries of $C$ are ordered in ascending order
3: Compute $Z = \begin{pmatrix} Z_1 & Z_2 \end{pmatrix} = Q_2 W$
4: Compute the full QR decomposition $UR_Z = Z_2$ and apply dropping to obtain
       $$Z = \begin{pmatrix} V_1 & V_2 \end{pmatrix} \begin{pmatrix} S_1 & \\ & S_2 \end{pmatrix}$$
5: Compute the SVD of $S_2$ and update $U$, $V$, $C$, $S$, and $W$
6: Compute $X = R^T W$
```

matrix. Since there are no existing CUDA-based batched GSVD routines available, we have implemented our own batched GSVD routine.

In our GPU implementation, we have fixed the block size to be $r = 16$. Consequently, the matrix $[H, G]$ has a size of 32×16. As outlined in Algorithm 28, two essential components of our implementation are a batched QR routine and a batched SVD routine. We utilize the batched SVD routine `cusolverDnDgesvdjBatched` from `cuSOLVER` in our current implementation. However, we have implemented our own batched QR routine since the existing batched QR routines do not align with our requirements.

Our batched QR algorithm is based on the Householder QR algorithm. During the batched QR computation of $[H, G]$, when launching the kernel, we select a block size of 32×8 with 8 warps (on the current architectures). Each warp is assigned to a matrix of size 32×16, meaning that each block handles the QR decomposition of 8 matrices. Each thread within the warp loads a row of the matrix into its register and operates on the same row during the decomposition process. We compute the inner product using the warp reduction function `__shfl_xor_sync`. After the decomposition, the upper part of the matrix in the register represents the R matrix, while the lower part is updated with the reflectors. To form the thin Q matrix, we apply the reflector to the first 16 columns of I_{32}. For the batched QR of S_2 and C_2, we include extra zeros and load them as 16×16 matrices into the register. In this case, we launch the kernel with a block size of 16×16, and each warp computes the QR of two matrices.

•	•	•	•	•	•	•	•
15	•	•	•	•	•	•	•
14	16	•	•	•	•	•	•
13	15	17	•	•	•	•	•
12	14	16	18	•	•	•	•
11	13	15	17	19	•	•	•
10	12	14	16	18	20	•	•
9	11	13	15	17	19	21	•
8	10	12	14	16	18	20	22
7	9	11	13	15	17	19	21
6	8	10	12	14	16	18	20
5	7	9	11	13	15	17	19
4	6	8	10	12	14	16	18
3	5	7	9	11	13	15	17
2	4	6	8	10	12	14	16
1	3	5	7	9	11	13	15

•	•	•	•	•	•	•	•
4	•	•	•	•	•	•	•
3	6	•	•	•	•	•	•
3	5	8	•	•	•	•	•
2	5	7	10	•	•	•	•
2	4	7	9	12	•	•	•
2	4	6	9	11	14	•	•
2	4	6	8	10	13	16	•
1	3	5	8	10	12	15	18
1	3	5	7	9	11	14	17
1	3	5	7	9	11	13	16
1	3	4	6	8	10	12	15
1	2	4	6	8	10	12	14
1	2	4	5	7	9	11	13
1	2	3	5	7	9	11	13
1	2	3	4	6	8	10	12

Figure 4.14: Example of the order for the Givens QR of a 16×8 matrix using the SK order (left) and the greedy order (right). During the ith step, entries with the order number i will be zeroed-out.

Block Givens QR

With the availability of the building block GSVD, the next step is to determine an optimal elimination order to maximize concurrency. In our CPU implementation, we can utilize established algorithms for standard Givens QR. Commonly used strategies include the Sameh Kuck (SK) scheme [140] and the pipeline Givens algorithm [139]. As an example, the SK order for a 16×8 matrix is illustrated in the left panel of Figure 4.14. After all the blocks in the strick lower triangular part are zeroed out, we compute the QR decomposition to diagonal blocks and finalize the computation. Once all the blocks in the strictly lower triangular part are zeroed out, we proceed with the QR decomposition of the diagonal blocks and finalize the computation. If a thin QR decomposition is desired, we store the U, V, C, and S matrices and compute the Q matrix after forming R.

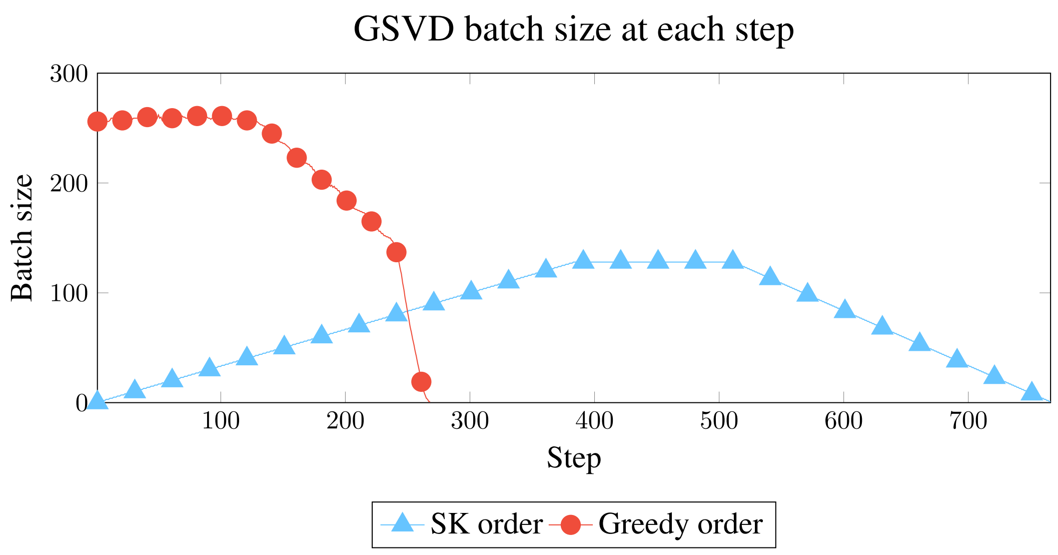


Figure 4.15: Comparision of the GSVD batch size during the block Givens QR of a matrix divided into 512×128 blocks using SK order and the greedy order.

In our GPU implementation, we aim to maximize concurrency at each step by determining an optimal elimination order. We first compute an order matrix on the host, which indicates the blocks to eliminate during the factorization. To achieve this, we employ a greedy approach that maximizes the number of GSVD operations that can be applied concurrently. Starting from the bottom left of the matrix, we select as many blocks as possible during each step. This ordering strategy is illustrated in the right panel of Figure 4.14. As shown in the figure, this approach significantly increases concurrency, particularly in the initial steps. In Figure 4.15, we compare the batch size during each step of the block Givens QR for a matrix with 512×128 blocks using both the SK order and our greedy order. By employing this strategy, we can fully benefit from the batched GSVD kernel. Once the GSVD is obtained, applying the rotations is straightforward. In our implementation, we launch the kernel with a block size of 16×16, where each block performs a block rotation on a matrix of size 32×16.

4.3.4 Numerical experiments

In the following section, we demonstrate the efficiency and effectiveness of our newly developed block Givens QR algorithm. The algorithm was implemented in C++, utilizing double-precision arithmetic for precise computation. The testing environment was provided by the `tacfarinas` machine at the University of Minnesota. This machine operates on an Ubuntu 22.04 system powered by an Intel Core i7-8700 CPU. Given that our current code version hasn't undergone optimization, we utilized the non-optimized routines from the reference versions of the `BLAS` and `LAPACK` libraries for linear algebra operations. This approach aids in maintaining a fair comparison by allowing the evaluation to focus more on the intrinsic efficiency of our algorithm rather than the impact of the underlying linear algebra operations. We implemented our algorithm for both row-major matrices and column-major matrices.

In the initial set of experiments, we assess the impact of block size on the algorithm's performance. For these tests, we constrain the number of OpenMP threads to one and vary the block size in a range from 2 to 128. The test matrix dimensions are fixed at 4096×1024. To gauge the effectiveness of our approach, we compare the results from our block Givens QR with those obtained using the standard Givens QR and the block Householder QR methods provided in the `LAPACK` library. All tests are conducted by computing the thin QR and explicitly forming the matrix Q.

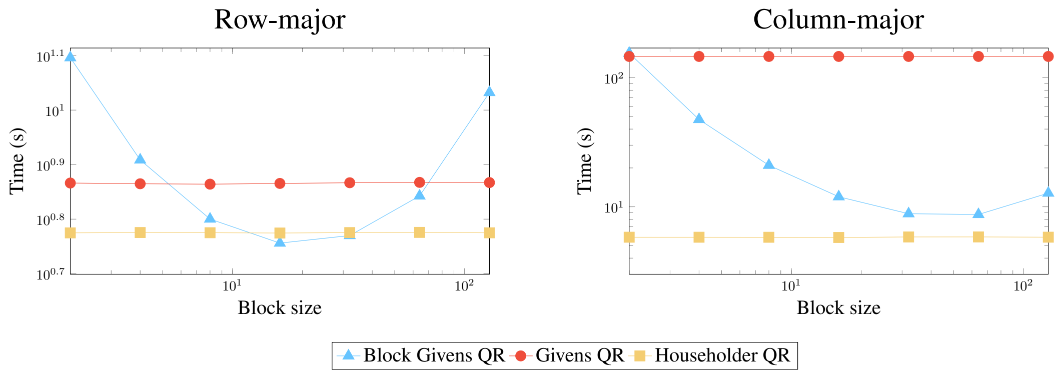


Figure 4.16: Total wall-clock time for computing the thin QR of a 4096×1024 matrix using block Givens QR with different block sizes. The results are compared with the standard Givens QR and the block Householder QR in `LAPACK`.

Figure 4.16 showcases the results. We observe that the total wall-clock time initially decreases and later increases with the escalation of block size for both row-major and column-major matrices. This is because as the block size increases, there is a reduction in the total cost of applying rotation, and memory is utilized more efficiently. However, the efficiency drops when the block size becomes too large, leading to an inefficient SVD computation and hence, an increase in the total wall-clock time.

From our results, it's evident that our block Givens QR outperforms block Householder QR from `LAPACK` when the input is row-major, given the block size is suitably selected. However, in the case of column-major inputs, the block Householder QR consistently exhibits higher efficiency due to its inherent row-wise rotation application in block Givens. Nevertheless, our block Givens QR notably outshines the standard Givens QR for all tested block sizes, except block size 2.

In the subsequent set of experiments, we hold the block size constant at 16 and test problems of varying sizes. The same algorithms used in the previous experiment are compared, with the exception of the column-major standard Givens QR, which is omitted due to its extreme inefficiency.

Table 4.8: Comparison of total wall-clock times required for computing the thin QR of matrices with varying dimensions using the block Givens QR algorithm with a fixed block size of 16. The column-major standard Givens QR is omitted due to its inefficiency. The results are benchmarked against the standard Givens QR and the block Householder QR from `LAPACK`.

m	n	**Row-major**			**Column-major**	
		`Givens`	`House`	`BGivens`	`House`	`BGivens`
4096	1024	7.4	6.0	5.8	5.8	12.0
4096	2048	27.8	21.6	17.8	21.2	42.4
4096	4096	93.7	69.0	52.8	68.1	133.3
1024	1024	1.1	1.0	1.0	1.0	2.0
2048	1024	3.0	2.6	2.5	2.6	5.3
4096	1024	7.4	6.0	5.8	5.8	12.0
8192	1024	16.0	12.6	12.0	12.3	25.4
16384	1024	33.2	26.2	24.6	25.3	52.8

We present the results in Table 4.8, where `Givens`, `House`, and `BGivens` represents standard Givens QR, Householder QR, and block Givens QR, respectively. The results presented in the table demonstrate that the block Givens QR consistently exhibits superior performance for row-major matrices. When working with tall and thin matrices, the performance of the row-major block Givens is comparable to that of the block Householder QR. However, as the number of columns increases, the row-major block Givens QR begins to outperform the block Householder QR significantly.

4.3.5 Conclusion

In conclusion, we presented a novel block Givens QR algorithm based on GSVD, offering advantages in terms of reduced computation cost and memory usage compared to the standard (block) Givens QR algorithm. Through extensive experiments, we demonstrated that our proposed algorithm achieves comparable performance to the well-established Householder QR algorithm.

We plan to further enhance our implementation by developing our own batch SVD kernel. Recent studies have highlighted the potential of optimized batched SVD implementations to outperform existing solutions like `cuSOLVER` [138]. By improving the performance of the batched GSVD kernel, we anticipate even greater efficiency and speed in our algorithm. Moreover, we aim to add scalability tests to our algorithms to gauge their effectiveness in larger-scale scenarios. Additionally, we intend to investigate the application of our algorithms in updating and downdating QR [141, 142], exploring their effectiveness and potential benefits in these contexts.

Chapter 5

Conclusion and Discussion

In conclusion, this dissertation focused on parallel Schur complement algorithms for the solution of linear systems and eigenvalue problems in distributed memory systems with GPUs. We summarized the main contributions of each chapter and discussed potential avenues for future research.

In Chapter 3, we studied several parallel Schur complement algorithms for the solution of sparse linear systems. This dissertation focus on ILU-based strategies, which are knwon for their robustness.

However, compared to AMG, ILU-based strategies have several limitations. The first limitation is that AMG is provable optimal for Poisson-like problems. Thus, AMG typically outperforms ILU for problems arising from PDEs with elliptic properties, especially for large problems. To address this limitation, in the second section of this chapter, we presented a two-level multiplicative Schur complement method. We show that combining it with the modified ILU approach can significantly improve the performance of the ILU preconditioner. The proposed algorithm, including several extra options, is implemented in the package `hypre` with GPU acceleration available. Future work will consider implementing CUDA kernels that support permutation arrays and modified ILU to improve the GPU performance of our implementation.

The second limitation of ILU-based strategies is their sequential nature. We improved the GeMSLR, a Schur complement low-rank preconditioning algorithm, to address this limitation. We modified the reordering scheme and updated the formula for computing the low-rank correction. The final algorithm is implemented in the package `parGeMSLR` with GPU acceleration available. Our implementation demonstrates good parallel efficiency as well as good convergence performance on both model problems and real-world problems. This is presented in the third section of this chapter. In the future, we plan to improve the performance of our thick-restart Arnoldi with its block version and study the performance of our preconditioner when solving linear systems with multiple right-hand sides.

In the last chapter of this section, we studied the parallel Schur complement low-rank approach for SPD matrices. We show that by using a Hermite-Remez approximation, we can construct a polynomial that could be used to accelerate the construction of low-rank correction terms and reduce the rank. We also show that low-rank correction terms could be used to provide approximate inverse with higher accuracy. We plan to evaluate the performance of our algorithm with more real-world applications in the future.

Chapter 4 focuses on a parallel Schur complement algorithm for solving symmetric eigenvalue problems. We focus on the problem of searching for algebraically smallest eigenvalues and associated eigenvectors of sparse matrix pencil (A, M), which is useful in many applications, including building low-rank correction terms discussed in the previous section. Commonly used Lanczo-based algorithms generally require a distributed-memory factorization of A or M, limiting their parallel efficiency. We proposed a parallel Schur complement algorithm based on the Chebyshev approximation to address the above limitation. Our proposed algorithm transfers the generalized eigenvalue problem into several local standard eigenvalue problems. Our algorithm also introduces model parallelism in addition to data parallelism by computing problems on different Chebyshev nodes in parallel. The final algorithm is implemented in the package `schurCheb`. Our algorithm is compared to `PARPACK` and found to perform better in many situations. In the future, we plan to study the GPU implementation of our algorithm and apply our algorithm to problems from real-world applications.

In the next section of this chapter, a block Givens QR algorithm that could be used for orthogonalization is outlined. The proposed algorithm uses block Givens rotations to compute the QR decomposition of a matrix. Compared to the classical (block) Givens QR, our proposed algorithm could significantly reduce the total computation cost and is suitable for multi-core and many-core architectures. Our next step is to implement our own batched SVD kernel to speed up the computation of GSVD on GPU. We also plan to study the usefulness of our proposed algorithm in other applications, including updating and downdating QR.

References

[1] Shifan Zhao, Tianshi Xu, Edmond Chow, and Yuanzhe Xi. An adaptive factorized Nyström preconditioner for regularized kernel matrices, April 2023. arXiv:2304.05460 [cs, math].

[2] Timothy A Davis. *Direct methods for sparse linear systems.* SIAM, 2006.

[3] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, January 2003.

[4] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

[5] Tianshi Xu, Ruipeng Li, and Daniel Osei-Kuffuor. A two-level GPU-accelerated incomplete LU preconditioner for general sparse linear systems, March 2023. arXiv:2303.08881 [cs, math].

[6] Tianshi Xu, Vassilis Kalantzis, Ruipeng Li, Yuanzhe Xi, Geoffrey Dillon, and Yousef Saad. parGeMSLR: A parallel multilevel Schur complement low-rank preconditioning and solution package for general sparse matrices. *Parallel Computing*, 113:102956, October 2022.

[7] Tianshi Xu, Anthony P. Austin, Vassilis Kalantzis, and Yousef Saad. A parallel algorithm for computing partial spectral factorizations of matrix pencils via Chebyshev approximation, 2023. To Appear.

[8] Huan He, Shifan Zhao, Yuanzhe Xi, Joyce Ho, and Yousef Saad. GDA-AM: On the effectiveness of solving min-imax optimization via Anderson mixing. In *International Conference on Learning Representations*, 2022.

[9] Huan He, Ziyuan Tang, Shifan Zhao, Yousef Saad, and Yuanzhe Xi. NLTGCR: A class of nonlinear acceleration procedures based on conjugate residuals, May 2023. arXiv:2306.00325 [cs, math].

[10] Joseph F Traub and H Woźniakowski. On the optimal solution of large linear systems. *Journal of the ACM*, 31(3):545–559, June 1984.

[11] Arthur W Chou. On the optimality of Krylov information. *Journal of Complexity*, 3(1):26–40, March 1987.

[12] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(1):17–29, 1951.

[13] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255, October 1950.

[14] Anne Greenbaum, Vlastimil Pták, and Zdenvek Strakoš. Any nonincreasing convergence curve is possible for GMRES. *SIAM Journal on Matrix Analysis and Applications*, 17(3):465–469, July 1996.

[15] John W Ruge and Klaus Stüben. Algebraic Multigrid. In *Multigrid Methods*, Frontiers in Applied Mathematics, pages 73–130. Society for Industrial and Applied Mathematics, January 1987.

[16] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition*. Society for Industrial and Applied Mathematics, second edition, January 2000.

[17] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, April 2002.

[18] Antonio Aricò and Marco Donatelli. A V-cycle Multigrid for multilevel matrix algebras: proof of optimality. *Numerische Mathematik*, 105(4):511–547, February 2007.

[19] Matthias Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338(1):201–218, November 2001.

[20] Na Li, Yousef Saad, and Edmond Chow. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing*, 25(2):716–728, January 2003.

[21] Friedrich L. Bauer. Das verfahren der treppeniteration und verwandte verfahren zur lösung algebraischer eigenwertprobleme. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, 8(3):214–235, May 1957.

[22] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems: Revised Edition.* Society for Industrial and Applied Mathematics, January 2011.

[23] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix Market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137. Springer US, Boston, MA, 1997.

[24] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide, Third Edition.* Society for Industrial and Applied Mathematics, Philadelphia, PA, December 2016.

[25] Randolph E. Bank and Christian Wagner. Multilevel ILU decomposition. *Numerische Mathematik*, 82:543–576, June 1999.

[26] E. F. F. Botta and F. W. Wubs. Matrix renumbering ILU: An effective algebraic multilevel ILU preconditioner for sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):1007–1026, January 1999.

[27] Zhongze Li, Yousef Saad, and Masha Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10(5-6):485–509, July 2003.

[28] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, January 2015.

[29] Mardochée Magolu monga Made, Robert Beauwens, and Guy Warzée. Preconditioning of discrete Helmholtz operators perturbed by a diagonal complex matrix. *Communications in Numerical Methods in Engineering*, 16(11):801–817, 2000.

[30] Yogi A Erlangga, Cornelis Vuik, and Cornelis W Oosterlee. Comparison of multigrid and incomplete LU shifted-Laplace preconditioners for the inhomogeneous Helmholtz equation. *Applied Numerical Mathematics*, 56(5):648–666, May 2006.

[31] Martin B Van Gijzen, Yogi A Erlangga, and Cornelis Vuik. Spectral analysis of the discrete Helmholtz operator preconditioned with a shifted Laplacian. *SIAM Journal on Scientific Computing*, 29(5):1942–1958, January 2007.

[32] Yogi A Erlangga, Cornelis W Oosterlee, and Cornelis Vuik. A novel multigrid based preconditioner for heterogeneous Helmholtz problems. *SIAM Journal on Scientific Computing*, 27(4):1471–1492, January 2006.

[33] Yuanzhe Xi and Yousef Saad. A rational function preconditioner for indefinite sparse linear systems. *SIAM Journal on Scientific Computing*, 39(3):A1145–A1167, January 2017.

[34] Xiao Liu, Yuanzhe Xi, Yousef Saad, and Maarten V. De Hoop. Solving the three-dimensional high-frequency Helmholtz equation using contour integration and polynomial preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 41(1):58–82, January 2020.

[35] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, February 2013.

[36] Jan Mandel and Clark R. Dohrmann. Convergence of a balancing domain decomposition by constraints and energy minimization. *Numerical Linear Algebra with Applications*, 10(7):639–659, October 2003.

[37] Charbel Farhat, Michel Lesoinne, Patrick LeTallec, Kendall Pierson, and Daniel Rixen. FETI-DP: a dual-primal unified FETI method part I: A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50(7):1523–1544, March 2001.

[38] Alexander Heinlein, Axel Klawonn, Martin Lanser, and Janine Weber. Combining machine learning and adaptive coarse spaces—a hybrid approach for robust FETI-DP methods in three dimensions. *SIAM Journal on Scientific Computing*, 43(5):S816–S838, January 2021.

[39] Nicole Spillane, Victorita Dolean, Patrice Hauret, Frédéric Nataf, Clemens Pechstein, and Robert Scheichl. Abstract robust coarse spaces for systems of PDEs via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770, April 2014.

[40] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, January 1999.

[41] David Hysom and Alex Pothen. Efficient parallel computation of ILU(k) preconditioners. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, SC '99, pages 29–es, New York, NY, USA, January 1999. Association for Computing Machinery.

[42] George Karypis and Vipin Kumar. Parallel threshold-based ILU factorization. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '97*, pages 1–24, San Jose, CA, 1997. ACM Press.

[43] Yousef Saad and Jun Zhang. BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 21(1):279–299, January 1999.

[44] Italo Cristiano L Nievinski, Michael Souza, Paulo Goldfeld, Douglas Adriano Augusto, José Roberto P Rodrigues, and Luiz Mariano Carvalho. Parallel implementation of a two-level algebraic ILU(k)-based domain decomposition preconditioner. *TEMA (São Carlos)*, 19:59–77, 2018.

[45] Geoffrey Dillon, Vassilis Kalantzis, Yuanzhe Xi, and Yousef Saad. A Hierarchical Low Rank Schur Complement Preconditioner for Indefinite Linear Systems. *SIAM Journal on Scientific Computing*, 40(4):A2234–A2252, January 2018.

[46] Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, István Reguly, Nikolai Sakharnykh, Vijay Sellappan, and Robert Strzodka. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, January 2015.

[47] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse Cholesky factorization on GPUs. *Parallel Computing*, 59:140–150, November 2016.

[48] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. A distributed CPU-GPU sparse direct solver. In *Euro-Par 2014 Parallel Processing*, volume 8632, pages 487–498. Springer International Publishing, Cham, 2014.

[49] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In *Computational Science ICCS 2009*, volume 5544, pages 864–873. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[50] Michael A Clark, Ronald Babich, Kipton Barros, Richard C Brower, and Claudio Rebbi. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):1517–1528, September 2010.

[51] Christian Richter, Sebastian Schops, and Markus Clemens. GPU acceleration of algebraic multigrid preconditioners for discrete elliptic field problems. *IEEE Transactions on Magnetics*, 50(2):461–464, February 2014.

[52] Rajesh Gandham, Kenneth Esler, and Yongpeng Zhang. A GPU accelerated aggregation algebraic multigrid method. *Computers & Mathematics with Applications*, 68(10):1151–1160, November 2014.

[53] Luc Buatois, Guillaume Caumon, and Bruno Lévy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24(3):205–223, June 2009.

[54] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. ViennaCL—linear algebra library for multi- and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, January 2016.

[55] PARALUTION Labs. Paralution v1.1.0, 2016. `http://www.paralution.com/`.

[56] Simon Gawlok, Philipp Gerstner, Saskia Haupt, Vincent Heuveline, Jonas Kratzke, Philipp Lösel, Katrin Mang, Mareike Schmidtobreick, Nicolai Schoch, Nils Schween, Jonathan Schwegler, Chen Song, and Martin Wlotzka. HiFlow3

technical report on release 2.0. *Preprint Series of the Engineering Mathematics and Computing Lab*, pages No 06 (2017): HiFlow3 – Technical Report on Release 2.0, November 2017.

[57] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34(150):473–497, 1980.

[58] Norman E. Gibbs, William G. Poole, Jr., and Paul K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.

[59] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. Parallel graph coloring with applications to the incomplete-LU factorization on the GPU. Technical report, Nvidia, January 2015.

[60] Edgar A. León. Mpibind: a memory-centric affinity algorithm for hybrid applications. In *Proceedings of the International Symposium on Memory Systems*, pages 262–264, Alexandria Virginia, October 2017. ACM.

[61] Quan M. Bui, François P. Hamon, Nicola Castelletto, Daniel Osei-Kuffuor, Randolph R. Settgast, and Joshua A. White. Multigrid reduction preconditioning framework for coupled processes in porous and fractured media. *Computer Methods in Applied Mechanics and Engineering*, 387:114111, December 2021.

[62] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, and Stefano Zampini. MFEM: A modular finite element library. *Computers & Mathematics with Applications*, 2020.

[63] MFEM: Modular finite element methods [Software]. `mfem.org`.

[64] GLVis: Opengl finite element visualization tool. `glvis.org`.

[65] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, 33(5):2864–2887, January 2011.

[66] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.

[67] Hartwig Anzt, Edmond Chow, and Jack Dongarra. ParILUT—a new parallel threshold ILU factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, January 2018.

[68] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. ParILUT - a parallel threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–241, Rio de Janeiro, Brazil, May 2019. IEEE.

[69] Ruipeng Li, Yuanzhe Xi, and Yousef Saad. Schur complement-based domain decomposition preconditioners with low-rank corrections. *Numerical Linear Algebra with Applications*, 23(4):706–729, August 2016.

[70] Yuanzhe Xi, Ruipeng Li, and Yousef Saad. An algebraic multilevel preconditioner with low-rank corrections for sparse symmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 37(1):235–259, January 2016.

[71] Laura Grigori, Frédéric Nataf, and Soleiman Yousef. Robust algebraic Schur complement preconditioners based on low rank corrections. Research Report RR-8557, INRIA, July 2014.

[72] Hussam Al Daas, Tyrone Rees, and Jennifer Scott. Two-level Nyström–Schur preconditioner for sparse symmetric positive definite matrices. *SIAM Journal on Scientific Computing*, 43(6):A3837–A3861, January 2021.

[73] Pascal Hénon and Yousef Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM Journal on Scientific Computing*, 28(6):2266–2293, January 2006.

[74] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, October 1996.

[75] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[76] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998.

[77] Qingqing Zheng, Yuanzhe Xi, and Yousef Saad. A power Schur complement low-rank correction preconditioner for general sparse linear systems. *SIAM Journal on Matrix Analysis and Applications*, 42(2):659–682, January 2021.

[78] Katarzyna Swirydowicz, Julien Langou, Shreyas Ananthan, Ulrike Yang, and Stephen Thomas. Low synchronization Gram-Schmidt and generalized minimal residual algorithms. *Numerical Linear Algebra with Applications*, page e2343, 2020.

[79] Yogi A Erlangga, Cornelis Vuik, and Cornelis Willebrordus Oosterlee. On a class of preconditioners for solving the Helmholtz equation. *Applied Numerical Mathematics*, 50(3-4):409–425, September 2004.

[80] Daniel Osei-Kuffuor and Yousef Saad. Preconditioning Helmholtz linear systems. *Applied Numerical Mathematics*, 60(4):420–431, April 2010.

[81] Valeria Simoncini and Efstratios Gallopoulos. An iterative method for nonsymmetric systems with multiple right-hand sides. *SIAM Journal on Scientific Computing*, 16(4):917–933, July 1995.

[82] Hussam Al Daas, Laura Grigori, Pascal Hénon, and Philippe Ricoux. Enlarged GMRES for solving linear systems with one or multiple right-hand sides. *IMA Journal of Numerical Analysis*, 39(4):1924–1956, October 2019.

[83] Vassilis Kalantzis, Constantine Bekas, Alessandro Curioni, and Efstratios Gallopoulos. Accelerating data uncertainty quantification by solving linear systems with multiple right-hand sides. *Numerical Algorithms*, 62(4):637–653, April 2013.

[84] Vassilis Kalantzis, A. Cristiano I. Malossi, Costas Bekas, Alessandro Curioni, Efstratios Gallopoulos, and Yousef Saad. A scalable iterative dense linear system

solver for multiple right-hand sides in data analytics. *Parallel Computing*, 74:136–153, May 2018.

[85] Eugene Remez. Sur le calcul effectif des polynômes dapproximation de Tchebichef. *CR Acad. Sci. Paris*, 199:337–340, 1934.

[86] Eugene Remez. Sur un procédé convergent dapproximations successives pour déterminer les polynômes dapproximation. *CR Acad. Sci. Paris*, 198:2063–2065, 1934.

[87] Eugene Remez. Sur la détermination des polynômes dapproximation de degré donnée. *Comm. Soc. Math. Kharkov*, 10(196):41–63, 1934.

[88] Alfred Haar. Die minkowskische geometrie und die annäherung an stetige funktionen. *Mathematische Annalen*, 78(1):294–311, December 1917.

[89] Michael James David Powell. *Approximation theory and methods.* Cambridge university press, 1981.

[90] Sadegh Jokar and Bahman Mehri. The best approximation of some rational functions in uniform norm. *Applied Numerical Mathematics*, 55(2):204–214, October 2005.

[91] Ruipeng Li, Yuanzhe Xi, Lucas Erlandson, and Yousef Saad. The eigenvalues slicing library (EVSL): Algorithms, implementation, and software. *SIAM Journal on Scientific Computing*, 41(4):C393–C415, January 2019.

[92] Andrea Franceschini, Victor Antonio Paludetto Magri, Massimiliano Ferronato, and Carlo Janna. A robust multilevel approximate inverse preconditioner for symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 39(1):123–147, January 2018.

[93] Dimitris Berberidis and Georgios B. Giannakis. Data sketching for large-scale Kalman filtering. *IEEE Transactions on Signal Processing*, 65(14):3688–3701, July 2017.

[94] Edmond Chow and Yousef Saad. Preconditioned Krylov subspace methods for sampling multivariate Gaussian distributions. *SIAM Journal on Scientific Computing*, 36(2):A588–A608, January 2014.

[95] Peter Laflin, Alexander V. Mantzaris, Fiona Ainley, Amanda Otley, Peter Grindrod, and Desmond J. Higham. Discovering and validating influence in a dynamic online social network. *Social Network Analysis and Mining*, 3(4):1311–1323, December 2013.

[96] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, December 2007.

[97] Jeffrey K. Bennighof and R. B. Lehoucq. An automated multilevel substructuring method for eigenspace computation in linear elastodynamics. *SIAM Journal on Scientific Computing*, 25(6):2084–2106, January 2004.

[98] Ward Heylen, Stefan Lammens, and Paul Sas. *Modal analysis theory and testing*, volume 200. Katholieke Universiteit Leuven Leuven, Belgium, 1997.

[99] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Society for Industrial and Applied Mathematics, January 1998.

[100] Kristyn J Maschho and Danny Sorensen. A portable implementation of ARPACK for distributed memory parallel architectures. In *Proceedings of the Copper Mountain Conference on Iterative Methods*, volume 1. Citeseer, 1996.

[101] Andreas Stathopoulos and James R. McCombs. PRIMME: preconditioned iterative multimethod eigensolvermethods and software description. *ACM Transactions on Mathematical Software*, 37(2):1–30, April 2010.

[102] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov. Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in hypre and PETSc. *SIAM Journal on Scientific Computing*, 29(5):2224–2239, January 2007.

[103] Vassilis Kalantzis, James Kestyn, Eric Polizzi, and Yousef Saad. Domain decomposition approaches for accelerating contour integration eigenvalue solvers for symmetric eigenvalue problems. *Numerical Linear Algebra with Applications*, 25(5), October 2018.

[104] James Kestyn, Vasileios Kalantzis, Eric Polizzi, and Yousef Saad. PFEAST: A high performance sparse eigenvalue solver using distributed-memory linear solvers. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 178–189, Salt Lake City, UT, USA, November 2016. IEEE.

[105] Eric Polizzi. Density-matrix-based algorithm for solving eigenvalue problems. *Physical Review B*, 79(11):115112, March 2009.

[106] Tetsuya Sakurai, Yasunori Futamura, Akira Imakura, and Toshiyuki Imamura. Scalable Eigen-Analysis Engine for Large-Scale Eigenvalue Problems. In *Advanced Software Technologies for Post-Peta Scale Computing*, pages 37–57. Springer Singapore, Singapore, 2019.

[107] David B. Williams-Young, Paul G. Beckman, and Chao Yang. A shift selection strategy for parallel shift-invert spectrum slicing in symmetric self-consistent eigenvalue computation. *ACM Transactions on Mathematical Software*, 46(4):1–31, December 2020.

[108] Hong Zhang, Barry Smith, Michael Sternberg, and Peter Zapol. SIPs: Shift-and-invert parallel spectral transformations. *ACM Transactions on Mathematical Software*, 33(2):9, June 2007.

[109] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, September 2005.

[110] Vasileios Kalantzis. *Domain decomposition algorithms for the solution of sparse symmetric generalized eigenvalue problems.* PhD Thesis, University of Minnesota, 2018.

[111] Weiguo Gao, Xiaoye S. Li, Chao Yang, and Zhaojun Bai. An implementation and evaluation of the AMLS method for sparse eigenvalue problems. *ACM Transactions on Mathematical Software*, 34(4):1–28, July 2008.

[112] Jin Hwan Ko and Zhaojun Bai. High-frequency response analysis via algebraic substructuring. *International Journal for Numerical Methods in Engineering*, 76(3):295–313, October 2008.

[113] Chao Yang, Weiguo Gao, Zhaojun Bai, Xiaoye S. Li, Lie-Quan Lee, Parry Husbands, and Esmond Ng. An algebraic substructuring method for large-scale eigenvalue calculation. *SIAM Journal on Scientific Computing*, 27(3):873–892, January 2005.

[114] Constantine Bekas and Yousef Saad. Computation of smallest eigenvalues using spectral Schur complements. *SIAM Journal on Scientific Computing*, 27(2):458–481, January 2005.

[115] Vassilis Kalantzis. A spectral Newton-Schur algorithm for the solution of symmetric generalized eigenvalue problems. *ETNA - Electronic Transactions on Numerical Analysis*, 52:132–153, 2020.

[116] Vassilis Kalantzis, Ruipeng Li, and Yousef Saad. Spectral Schur complement techniques for symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 45:305–329, 2016.

[117] Tosio Kato. *Perturbation Theory for Linear Operators*, volume 132 of *Classics in Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[118] S.H. Lui. Kron's method for symmetric eigenvalue problems. *Journal of Computational and Applied Mathematics*, 98(1):35–48, October 1998.

[119] Yangfeng Su, Tianyi Lu, and Zhaojun Bai. 2D eigenvalue problem I: Existence and number of solutions, September 2022. arXiv:1911.08109 [cs, math].

[120] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, January 2019.

[121] Christopher A. Beattie, Mark Embree, and D. C. Sorensen. Convergence of polynomial restart Krylov methods for eigenvalue computations. *SIAM Review*, 47(3):492–515, January 2005.

[122] G. W. Stewart and Ji-Guang Sun. *Matrix Perturbation Theory*. Academic Press, Boston, MA, 1990.

[123] James R Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of computation*, 31(137):163–179, 1977.

[124] Yuanzhe Xi, Ruipeng Li, and Yousef Saad. Fast computation of spectral densities for generalized eigenvalue problems. *SIAM Journal on Scientific Computing*, 40(4):A2749–A2773, January 2018.

[125] Daniela Calvetti, Lothar Reichel, and Danny Chris Sorensen. An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(1):21, 1994.

[126] Kesheng Wu and Horst Simon. Thick-restart Lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(2):602–616, January 2000.

[127] C. Bekas and A. Curioni. Very large scale wavefunction orthogonalization in Density Functional Theory electronic structure calculations. *Computer Physics Communications*, 181(6):1057–1068, June 2010.

[128] Erin Carson, Kathryn Lund, Miroslav Rozloník, and Stephen Thomas. Block Gram-Schmidt algorithms and their stability properties. *Linear Algebra and its Applications*, 638:150–195, April 2022.

[129] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and D Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, January 1999.

[130] Robert D. Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *Computational Science - ICCS 2002*, Lecture Notes in Computer Science, pages 632–641, Berlin, Heidelberg, 2002. Springer.

[131] Takeo Hoshi, Hiroto Imachi, Akiyoshi Kuwata, Kohsuke Kakuda, Takatoshi Fujita, and Hiroyuki Matsui. Numerical aspect of large-scale electronic state calculation for flexible device material. *Japan Journal of Industrial and Applied Mathematics*, 36(2):685–698, July 2019.

[132] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):1–25, 2011.

[133] Patrick R. Amestoy, Iain S. Duff, Jean-Yves LExcellent, and Jacko Koster. MUMPS: A general purpose distributed memory sparse solver. In *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, volume 1947, pages 121–130. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[134] Xiaoye S. Li and James W. Demmel. SuperLU_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, June 2003.

[135] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* JHU press, 2013.

[136] Charles Van Loan. A block QR factorization scheme for loosely coupled systems of array processors. In *Numerical Algorithms for Modern Parallel Computer Architectures*, volume 13, pages 217–232. Springer US, New York, NY, 1988.

[137] Andrew Kerr, Dan Campbell, and Mark Richards. QR decomposition on GPUs. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78, Washington D.C. USA, March 2009. ACM.

[138] Ahmad Abdelfattah, Stan Tomov, and Jack Dongarra. Batch QR factorization on GPUs: Design, optimization, and tuning. In *Computational Science ICCS 2022*, Lecture Notes in Computer Science, pages 60–74, Cham, 2022. Springer International Publishing.

[139] Marc Hofmann and Erricos John Kontoghiorghes. Pipeline Givens sequences for computing the QR decomposition on a EREW PRAM. *Parallel Computing*, 32(3):222–230, March 2006.

[140] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the ACM*, 25(1):81–91, January 1978.

[141] Jieping Ye, Qi Li, Hui Xiong, H. Park, R. Janardan, and V. Kumar. IDR/QR: an incremental dimension reduction algorithm via QR decomposition. *IEEE Transactions on Knowledge and Data Engineering*, 17(9):1208–1222, September 2005.

[142] K. Yoo and H. Park. Accurate downdating of a modified Gram-Schmidt QR decomposition. *BIT Numerical Mathematics*, 36(1):166–181, March 1996.