Robust High Performance Preconditioning Techniques for Solving General Sparse Linear Systems

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

Ruipeng Li

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Yousef Saad

June, 2015

Acknowledgements

I wish to express my deepest appreciation to my advisor Yousef Saad, who introduced me to the realm of numerical linear algebra, for his patience and support of this work, without whom this thesis would not have been at all possible. I feel myself very fortunate to have been his PhD student and have had his expert guidance all these years. It has been a privilege to me to have worked with him. His way of thinking has definitely influenced me.

My gratitude extends to my Master's advisor, Prof. Hai Jiang at Department of Computer Science, Arkansas State University, who inspired my interest in computer science in the first place. He encouraged me to pursuit the doctoral degree at University of Minnesota and kept a watchful eye on me.

I would like to extend my appreciation to Prof. Stergios Roumeliotis who opened a window for me to another fascinating field—robotics, for a series of introductory lectures on robotics and for the opportunity to work in his group for a year. I am indebted to Prof. Daniel Boley for many useful discussions in the group meetings. I also want to thank Prof. David Yuen at the Department of Earth Sciences for many conversations about "the GPU computing era" and the beers, too. I would also like to thank Yousef Saad, Stergios Roumeliotis, Daniel Boley and Sachin Sapatnekar for reviewing this thesis and serving on my doctoral final examination committee.

I am grateful to ConocoPhillips, NVIDIA and Stone Ridge Technology, who provided me four summers' worth of time as student interns. Our GPU-accelerated iterative solver package has been motivated by the work in these internships and has benefited from the knowledge I learned there.

My thanks goes to the colleagues in the Scientific Computing lab, Pierre Carrier, Jie Chen, Haw-ren Fang, Da Gao, Vasileios Kalantzis, Daniel Osei-Kuffuor, Thanh Ngo, Jok Tang, Shashanka Ubaru, Eugene Vecharynski, and Yuanzhe Xi, for the fruitful discussions, their valuable suggestions over the years, and also for the friendship. I especially would like to thank Daniel Osei-Kuffuor who never tired of answering my questions during the early years of my PhD. I also want to thank Yuanzhe Xi for the collaboration that contributes to Chapter 6 of this thesis.

Thanks are also due to the members of the Multiple Autonomous Robotic Systems (MARS) lab led by Prof. Roumeliotis, in particular, Ahmed Ahmed, Ryan Dutoit, Georgios Georgiou, Chao Guo, Dimitrios Kottas, Esha Nerurkar, John O'Leary, and Kejian Wu.

This work was supported by the Department of Energy under grant DOE/DE-FG-08ER25841, by the National Science Foundation under grant NSF/DMS-1216366, and by the Minnesota Supercomputing Institute.

— R.L.

Dedication

To my parents.

Abstract

The solution of large and sparse linear systems is often required by numerical simulations in many fields of science and engineering. Solving these linear systems is usually the major bottleneck for large-scale applications as it represents the most time-consuming part of the computations. For problems formulated in 2-D geometries, the state-of-the-art direct methods can efficiently solve fairly large sparse linear systems. On the other hand, for 3-D problems, the use of sparse direct methods has become prohibitive in terms of both the memory requirement and the computational complexity. For such problems, iterative methods have thus become a more attractive choice. Among these methods, Krylov subspace methods combined with incomplete LU (ILU) type preconditioners are among the most reliable general-purpose iterative solvers, which have been successfully employed in many applications. In spite of this, there are still two main drawbacks of the ILU-type preconditioners. In the first place is the robustness of these preconditioners. When the matrix is highly ill-conditioned or indefinite, ILU preconditioners are unlikely to work. Secondly, the construction and the application of these preconditioners represent a serial bottleneck, which leads to severe degradation of performance on modern parallel processors.

This thesis proposes several preconditioning methods with the considerations of both the robustness for indefinite problems and the efficiency on modern parallel computing architectures. First, we discuss the acceleration techniques by the current many-core processors for several preconditioning approaches. Next, we present a class of new preconditioning techniques based on low-rank approximations by exploiting decay properties of eigenvalues. These preconditioning methods are proposed primarily as means to bypass the difficulties mentioned above that are encountered by standard ILU preconditioners. Implementations of these methods and the performance comparisons with standard preconditioners are also discussed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis considers the problem of solving the linear system

$$Ax = b, \tag{1.1}$$

where the matrix $A \in \mathbb{R}^{n \times n}$, called the coefficient matrix, is large and sparse, $b \in \mathbb{R}^n$ is the right-hand-side vector, and $x \in \mathbb{R}^n$ is the solution vector. So far the biggest source of sparse linear systems is the numerical solution methods of partial differential equations (PDEs), which include finite difference methods, finite element methods, and finite volume methods. A common feature of these methods is that the solution of a PDE is discretized to a finite number of unknowns that are involved in a large set of linear equations. Typically, these discretizations are very sparse in the sense that an unknown is only coupled with a few other unknowns that are physically close to it. Thus, matrices arising from these discretizations are sparse, i.e., they have very few nonzero elements per row. Sparse matrix problems can also come from applications that are not governed by PDEs, including circuit simulation, economics modeling, optimization problems, and the PageRank problem [1], to name just a few.

The methods for computing the solution x of (1.1) fall into two main categories: *sparse direct methods* and *iterative methods*. Direct methods for sparse linear systems [2] perform a form of the LU factorization of the matrix A, which usually consists of 4 stages: 1) reordering the matrix A to reduce “fill-ins”, i.e., the nonzero entries in the LU factors that were initially zero in A, 2) a symbolic factorization to determine the nonzero pattern of the factorization, 3) a numerical factorization that computes the actual LU factors, and 4) a solve step that performs the forward and backward substitutions with the factors and the right-hand side b. In contrast with iterative methods, direct methods can always, when ignoring rounding errors, produce the solution to the working precision in a finite and fixed amount of computations. Therefore, their robustness lends themselves to be regarded as "black-box solvers". The books by Duff, Erisman and Reid [3] and Davis [2] can serve as excellent references for sparse direct methods. The state-of-the-art sparse direct solvers can efficiently solve fairly large sparse linear systems especially for those from problems formulated in 2-D geometries. On the other hand, for 3-D problems, the use of direct methods becomes prohibitive. For instance, consider Poisson's equation

$$-\Delta u = f, \text{ in domain } \Omega, \tag{1.2}$$

with proper boundary condition. For a 2-D case with n unknowns, the computational complexity of modern multifrontal direct solvers is of the order of $\mathcal{O}(n^{3/2})$ and the memory requirement is $\mathcal{O}(nlogn)$. For a 3-D case, this solver takes $\mathcal{O}(n^2)$ computational cost and $\mathcal{O}(n^{4/3})$ memory space, which will be very expensive when n is large. Combining these with the fact that direct methods are in general difficult to be parallelized, iterative methods would be left as the only option for solving such large 3-D problems.

Unlike sparse direct methods, iterative methods will require much less memory and much fewer operations for 3-D problems: the ratios between the operation counts and the storage requirement of these two types of methods can be of order n [4]. Essentially, there are two broad types of iterative methods: *specialized* and *general-purpose*. Specialized solvers are the methods designed for a certain type of problems and they often utilize information from the original PDE and the underlying mesh. For example, specialized solvers for solving the problem (1.2) on regular meshes include the fast Fourier transform, the block cyclic reduction method, and multigrid that is, in particular, asymptotically optimal. On the other hand, general-purpose solvers take the matrix A and the vector b and try to solve for the solution(s) x. These methods do not use any other information and make no assumption on the original problem. Apparently, direct methods belong to this type of solvers. General-purpose iterative methods include the basic iterative schemes such as the Jacobi iteration, the Gauss-Seidel iteration, successive overrelaxation (SOR), and the state-of-the-art Krylov subspace methods, e.g., the conjugate gradient (CG) method, the generalized minimal residual (GMRES) method, and the biconjugate gradient stabilized (BiCGSTAB) method.

Another line of general-purpose iterative methods is the algebraic multigrid (AMG) method [5, 6], which is a generalization of traditional multigrid. It tries to recover the properties of the underlying PDE and mesh merely from the matrix A. Therefore, the overall success of AMG is still somewhat restricted to certain types of problems. Moreover, it is not surprising that when AMG works, it can be extremely efficient, far more so than Krylov subspace methods [7]. However, the dividing line of these two types of methods, preconditioned Krylov subspace methods and AMG, is becoming blurred and combinations of these two methods are often seen. For instance, AMG is often used as a preconditioning method in conjunction with Krylov subspace methods. Also, the idea of splitting fine nodes and coarse nodes from the standard multigrid algorithm has been used to develop ILU-type preconditioners for Krylov subspace methods [8–10]. On the other hand, Krylov subspace methods have been used in AMG either as a smoother or as a multigrid cycle that is known as the "K-cycle" [11]. Moreover, multilevel ILU preconditioners can be reformulated to a form of AMG, or the AMG framework can be realized via multilevel ILU methods [12, §13.6.4]. A brief discussion on their connections will be given in Section 2.3.5.

In most cases, Krylov subspace methods, also referred to as *accelerators*, by themselves are not enough to lead to convergence. Some sort of preconditioning is often necessary. It is well known that the rate of convergence of iterative methods strongly depends on the spectral properties of the matrix A. Generally speaking, preconditioning amounts to transforming the original linear system in order to improve these properties. In modern iterative solution techniques, accelerators and preconditioners are the two essential ingredients. As Krylov subspace methods are reaching their maturity, preconditioning has become *the most critical ingredient* in developing robust and efficient iterative methods for solving linear systems [12, 13]. An excerpt from the book [14] by Trefethen and Bau reads:

"Nothing will be more central to computational science in the next century [21st century] than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly. For Krylov subspace matrix iterations, this is preconditioning." (p. 319).

In the past several decades, a tremendous amount of work has been carried out on the development of effective preconditioning approaches, and a variety of methods have been proposed, derived from different perspectives and with different considerations, such as using relaxation-type methods for preconditioning, incomplete LU (ILU) factorizations, and approximate inverse methods. Saad [12, p. 297] remarked that *"there are virtually no limits to available options for obtaining good preconditioners,"* and he characterized preconditioning techniques as *"a combination of art and science"*.

The comprehensive treatise by Saad [12] and the book by van der Vorst [4] are excellent references for Krylov subspace methods and preconditioning techniques. For a good overview of the history of the development of iterative methods, Saad and van der Vorst [7] is highly recommended. The survey paper by Benzi [13] gives an overview of the state of the art of preconditioning techniques.

ILU factorizations are among the most reliable preconditioners in a general setting and they have gained success in many applications. However, there are situations where these methods will not perform well. In the first place is the robustness of the preconditioner. When the matrix A is highly ill-conditioned or indefinite, ILU preconditioners are unlikely to work, either because the construction of the factors will not complete or because the resulting factors are unstable and in some cases quite dense. The second consideration is with regard to their efficiency on parallel machines. A drawback of ILU preconditioners is that the sparse triangular solves as the preconditioning operation represent a sequential bottleneck, which leads to severe degradation of performance for parallel processing. This actually motivated the development of a class of approximate inverse preconditioners in the 1990s as alternatives to ILU preconditioners [15–18]. More recently, the advent of the many-core architectures, such as Graphics Processing Units (GPUs), made this performance problem even more significant. The performance of sparse triangular solves was found exceedingly poor on GPUs [19].

This thesis proposes several new preconditioning techniques primarily as means to bypass the issues just mentioned for ILU-type preconditioning methods with the considerations of both the robustness for indefinite problems and the efficiency on modern parallel computing architectures.

1.1 Summary of thesis

The thesis is organized as follows. Chapter 2 gives some background of preconditioned Krylov subspace methods and preconditioning techniques, and discusses several ILU-type preconditioners in particular. Standard techniques in domain decomposition (DD) methods such as graph partitionings, sparse distributed linear systems, and Schur complement approaches are also introduced.

With Chapter 3 begins the discussions of the main topics of this thesis. This chapter presents several standard parallel preconditioners with GPU-acceleration. These methods themselves are by no means novel; rather, many of them can be found in classic books, for instance Saad [12]. Nevertheless, it is the adaptation of these methods to the new many-core platform and putting them together as a GPU-accelerated solver package that are the novel contributions by the author. Moreover, the methodology used to leverage GPUs to obtain the accelerations might be more important than the methods themselves, since it can provide important insights and guidelines for the further development of more advanced and complicated GPU-appropriate preconditioners.

Chapters 4, 5 and 6 are devoted to another main topic of this thesis—the low-rank approximation based preconditioners. In Chapter 4, we present a divide-and-conquer approach to compute preconditioners for symmetric matrices, which is referred to as the multilevel low-rank (MLR) preconditioner. In a nutshell, the idea is based on the observation that if a domain is divided into two subdomains, then one can get the inverse of the matrix associated with the whole domain by the inverses of the matrices associated with both subdomains plus a low-rank correction. The divide step can be performed via standard DD methods. In the conquer step, the inverse of the original matrix can be expressed by the Sherman-Morrison-Woodbury (SMW) formula, in which we apply the low-rank approximations to obtain the preconditioning matrix.

In Chapter 5, another preconditioning method for symmetric matrices is introduced. This method extends the idea of the MLR preconditioner to the general framework of distributed sparse linear systems via standard DD approaches. Again, an approximate inverse preconditioner is obtained by exploiting a low-rank property and the SMW formula. We refer to a preconditioner obtained by this approach as a DD based low-rank (DDLR) preconditioner. A difference between these two preconditioners is that the DDLR preconditioner is not recursive. As a result, this method is much easier to implement.

Finally, in Chapter 6, we use low-rank approximations with the classical Schur complement techniques and propose a Schur complement based low-rank correction preconditioner that is referred to as the SLR preconditioner for solving general sparse linear systems. In essence, the idea of this method is that if the difference between the inverse of the Schur complement and the inverse of the interface matrix, i.e., the matrix associated with the interface unknowns resulting from DD, can be well approximated by a low-rank matrix, we can directly obtain an approximate inverse of the Schur complement by the inverse of the interface matrix with a low-rank correction.

Experiment results indicated that the presented preconditioning methods appear to be more robust than the traditional ILU-based methods for indefinite problems and more efficient to apply in the iteration phase.

At the end of this chapter, it is worth to point out that even though these low-rank approximations based preconditioners were developed with highly parallel platforms in mind, especially the ones equipped with GPUs, we have not implemented and tested these methods in such environments. Nevertheless, the potentials can be seen from their performance tested on multi-core CPUs.

Chapter 2

Background

2.1 Krylov subspace methods

We consider the problem of solving the linear system

$$Ax = b,$$

where $A \in \mathbb{R}^{n\times n}$ is the coefficient matrix, $b \in \mathbb{R}^n$ is the right-hand-side vector, and $x \in \mathbb{R}^n$ is the solution vector. Given a nonsingular matrix M and an initial guess x_0, the splitting $A = M - N$ defines the basic iteration

$$\begin{aligned} Mx_m &= Nx_{m-1} + b \\ x_m &= M^{-1}(Nx_{m-1} + b) = x_{m-1} + M^{-1}(b - Ax_{m-1}). \end{aligned} \tag{2.1}$$

For $M = I$, we obtain the Richardson iteration

$$x_m = x_{m-1} + (b - Ax_{m-1}) = x_{m-1} + r_{m-1}. \tag{2.2}$$

On the other hand, when $M \neq I$, (2.1) can be viewed as the Richardson iteration for the equivalent "preconditioned system", $\tilde{A}x = \tilde{b}$ with $\tilde{A} = M^{-1}A$ and $\tilde{b} = M^{-1}b$. The preconditioned iterations will be discussed in the next section. Here we will only consider the case with $M = I$. From (2.2), it follows that

$$r_m = b - Ax_m = (I - A)r_{m-1} = (I - A)^m r_0, \tag{2.3}$$

and thus

$$\begin{aligned} x_m &= x_0 + r_0 + r_1 + \ldots + r_{m-1} \\ &= x_0 + \sum_{i=0}^{m-1} (I - A)^i r_0 \\ &\in x_0 + \operatorname{span}\{r_0, Ar_0, \ldots, A^{m-1} r_0\} \equiv x_0 + \mathcal{K}_m(A, r_0), \end{aligned}$$

where $\mathcal{K}_m(A, r_0)$ denotes the *Krylov subspace*

$$\mathcal{K}_m(A, r_0) = \operatorname{span}\{r_0, Ar_0, \ldots, A^{m-1} r_0\}.$$

In the methods of this type, at step m, the approximate solution x_m is sought in the affine space $x_0 + \mathcal{K}_m(A, r_0)$, which can be written in the form

$$x_m = x_0 + P_{m-1}(A) r_0,$$

where P_{m-1} is a polynomial of degree $m - 1$, and the corresponding residual is

$$r_m = (I - AP_{m-1}(A)) r_0 \equiv Q_m(A) r_0,$$

with Q_m is a polynomial of degree m and $Q_m(0) = 1$. For (2.3), $Q_m(A) = (I - A)^m$.

Better approximate solutions can be extracted from the affine space $x_0 + \mathcal{K}_m(A, r_0)$ by applying *projection methods* for solving linear systems, which are referred to as *Krylov subspace methods.* Let $\mathcal{K}$ and $\mathcal{L}$ be two m-dimensional subspaces. The projection method onto the subspace $\mathcal{K}$ and orthogonal to $\mathcal{L}$ seeks a solution $\tilde{x}$ such that

$$\tilde{x} \in x_0 + \mathcal{K} \quad \text{and} \quad b - A\tilde{x} \perp \mathcal{L},$$

the general scheme of which is represented in the following algorithm [12, §5.1.2].

```
1: repeat
2:     Select a pair of subspaces $\mathcal{K}$ and $\mathcal{L}$
3:     Choose bases $V = [v_1, \ldots, v_m]$ and $W = [w_1, \ldots, w_m]$ for $\mathcal{K}$ and $\mathcal{L}$
4:     $r = b - Ax$
5:     solve: $(W^T AV)y = W^T r$
6:     $x := x + Vy$
7: until convergence
```

For the Krylov subspace methods, we have subspace $\mathcal{K} = \mathcal{K}_m(A, r_0)$. These methods will fall into different categories based on the choices of the subspace $\mathcal{L}$, three of which are listed as the following.

1. $\mathcal{L} = \mathcal{K}_m(A, r_0)$. When A is SPD, the methods of this type minimize the A-norm of the error, i.e., $\|\tilde{x} - x^*\|_A = (A(\tilde{x} - x^*), \tilde{x} - x^*)^{1/2}$ where x^* denotes the exact solution. Examples include the conjugate gradient (CG) method, the Lanczos method and the full orthogonalization (FOM) method.

2. $\mathcal{L} = A\mathcal{K}_m(A, r_0)$. These methods minimize the residual norm $\|b - A\tilde{x}\|_2$. Examples are the generalized minimal residual (GMRES) method and the minimal residual (MINRES) method for symmetric cases.

3. $\mathcal{L} = \mathcal{K}_m(A^T, s_0)$ with $(r_0, s_0) \neq 0$. These methods are based on the Lanczos biorthogonalization procedure, examples of which are the biconjugate gradient (BCG) method and the quasi-minimal residual (QMR) method.

Let the columns of $V_m = [v_1, \ldots, v_m]$ form an orthonormal basis of the Krylov subspace $\mathcal{K}_m(A, r_0)$ as computed by the Arnoldi procedure [20] with r_0 as the initial vector. On the return of the Arnoldi procedure, we have $AV_m = V_{m+1}\bar{H}_m$, where $\bar{H}_m$ is an $(m+1) \times m$ upper Hessenberg matrix, and $V_m^T A V_m = H_m$, where $H_m \in \mathbb{R}^{m\times m}$ is the matrix obtained by deleting the last row of $\bar{H}_m$. Moreover, we have $\beta v_1 = r_0$ with $\beta = \|r_0\|_2$. In the FOM method, the orthogonality condition of the projection method leads to solving the system of equations

$$V_m^T A V_m y = V_m^T r_0 \quad \Leftrightarrow \quad H_m y = \beta e_1,$$

where e_1 denotes the first vector of the canonical basis, while in the GMRES method, the orthogonality condition leads to

$$(AV_m)^T A V_m y = (AV_m)^T r_0,$$

which solves the least-squares problem

$$\min_y \|AV_m y - r_0\|_2 \quad \Leftrightarrow \quad \min_y \|\bar{H}_m y - \beta e_1\|_2.$$

A question that has not been addressed is, *how fast do the Krylov subspace methods converge, and what does the convergence rate depend on?* When A is a normal matrix and thus is unitarily diagonalizable, let $A = X\Lambda X^H$, where $\Lambda = \mathrm{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ is the diagonal matrix of the eigenvalues and X is the unitary matrix of the eigenvectors. At the m-th step of a residual minimization method (e.g., GMRES), we can write

$$\|r_m\|_2 \le \|Q(A)r_0\|_2 = \|Q(\Lambda)r_0\|_2 \le \max_i |Q(\lambda_i)| \|r_0\|_2, \quad \forall Q \in \mathbb{P}_m \text{ and } Q(0) = 1,$$

where Q is any polynomial of degree not exceeding m which satisfies $Q(0) = 1$. Moreover, in a simple case when all the λ_i's are real and positive, it can be shown [21] that

$$\|r_m\|_2 \le 2\left(1 - \frac{2}{\sqrt{\kappa}+1}\right)^m \|r_0\|_2. \tag{2.4}$$

where $\kappa = \lambda_1/\lambda_n$ is the ratio of the largest eigenvalue to the smallest. See [12, §6.11.4] for more general results. Therefore, when GMRES is applied to a normal matrix, the convergence merely depends on *the distribution of the eigenvalues of* A. The convergence is in general slow when the spectral condition number κ is large, and the placement of the eigenvalues also has effect on the convergence. Clustered eigenvalues away from zero are generally more favorable.

For a diagonalizable matrix $A = X\Lambda X^{-1}$, the convergence will also depend on the condition of the eigenvectors,

$$\|r_m\|_2 \le \|Q(A)r_0\|_2 = \|XQ(\Lambda)X^{-1}r_0\|_2 \le \kappa_2(X) \max_i |Q(\lambda_i)| \|r_0\|_2,$$

where $\kappa_2(X) = \|X\|_2 \|X^{-1}\|_2$ denotes the 2-norm condition number of X, and a similar result to (2.4) is

$$\|r_m\|_2 \le 2\kappa_2(X)\left(1 - \frac{2}{\sqrt{\kappa}+1}\right)^m \|r_0\|_2.$$

When A is nearly normal, in which case $\kappa_2(X)$ is not too large, it is still reasonable to say that the convergence of GMRES is determined by the eigenvalues of A. When A is far from being normal or not diagonalizable, the convergence properties will be much more complicated [22].

2.2 Preconditioned iterations

In practical use of the Krylov subspace methods, it is almost always necessary to combine some sort of preconditioning to achieve convergence. Roughly speaking, preconditioning is a modification to the original linear system such that the modified linear system has the same solution and is easier to solve with iterative methods.

For a linear system $Ax = b$, we can write the *preconditioned linear system* as

$$M^{-1}Ax = M^{-1}b, \tag{2.5}$$

where M is called the *preconditioning matrix*, and $M^{-1}A$ is called the *preconditioned matrix*. From a practical point of view, M should satisfy the following requirements:

1. The solve with M is inexpensive,

2. M is close to A in some sense such that $M^{-1}A$ is better conditioned.

When applying Krylov subspace methods to (2.5), $M^{-1}A$ is required only for the matrix-vector product of the form $u = M^{-1}Av$ at each step of the iterations where a solve with M is required. Therefore, we neither need to form $M^{-1}A$ nor do we need to invert M. All that is needed is solving linear systems with M. Furthermore, in some cases, M may not be explicitly defined. Examples are multilevel preconditioning algorithms where M is often recursively defined, and the case where another iterative method is used as the preconditioner, which is often referred to as the *inner-outer-iteration* scheme. With a preconditioning matrix M, three forms of preconditioning exist.

1. Left preconditioning as shown in (2.5).

2. Right preconditioning, which leads to the preconditioned system

$$AM^{-1}y = b, \quad x = M^{-1}y. \tag{2.6}$$

3. Split preconditioning. For examples, when $M = LL^T$ as obtained from an incomplete Cholesky factorization,

$$L^{-1}AL^{-T}y = L^{-1}b, \quad x = L^{-T}y. \tag{2.7}$$

The splitting preconditioning might be useful in the symmetric cases. Note that when A is SPD and $M = LL^T$, we cannot apply the CG method to the left- and right-preconditioned system, (2.5) and (2.6), since $M^{-1}A$ and AM^{-1} are not symmetric, whereas CG is directly applicable to the split-preconditioned system (2.7) where the symmetry is preserved. However, left- and right-preconditioning can still be used by preserving the symmetry in alternative ways. Specifically, when A and M are both SPD, $M^{-1}A$ is self-adjoint with respect to the M-inner product, which is

$$(x, y)_M = (Mx, y) = (x, My)$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M^{-1}Ay)_M.$$

Similarly, AM^{-1} is self-adjoint with respect to the M^{-1}-inner product. Therefore, the CG method can be rewritten with the M- and the M^{-1}-inner products for the left- and the right-preconditioning respectively. Also, it can be shown that *all the three preconditioned CG methods are mathematically equivalent* [12]. Note that for left- and right-preconditioning, M is not required to be in a factored form.

2.3 Incomplete LU preconditioners

In a nutshell, the main idea of incomplete LU (ILU) factorizations is to drop part of the nonzeros in the LU factors in the course of Gaussian elimination. Let G be the set of the positions where the nonzeros will be dropped, which excludes the diagonal,

$$G \subseteq \{(i, j) \mid 1 \leq i, j \leq n \;\text{ and }\; i \neq j\}. \tag{2.8}$$

For an ILU factorization with respect to G, which can be represented by

$$A = LU - R, \tag{2.9}$$

we have

$$l_{i,j} = u_{i,j} = 0, \;\text{ if } (i, j) \in G \quad\text{and}\quad R_{ij} = 0, \;\text{ if } (i, j) \notin G.$$

In practical implementation, when the matrix A is stored in a sparse row format, e.g., in the CSR format, it will be efficient to compute Gaussian elimination in a row-wise version, where the rows of L and U are computed one at a time. This is often referred to as the *IKJ* variant, the “up-looking” scheme, or sometimes the Tinny-Walker algorithm [3]. The ILU factorization based on the *IKJ* variant is shown as follows where A will be overwritten by L and U after the factorization.

Algorithm 1 ILU factorization, IKJ variant

```
1: for $i = 2, \ldots, n$ do
2:   for $k = 1, \ldots, i-1$ and if $(i,k) \notin G$ do
3:     $a_{i,k} = a_{i,k}/a_{k,k}$
4:     for $j = k+1, \ldots, n$ and if $(i,j) \notin G$ do
5:       $a_{i,j} = a_{i,j} - a_{i,k}a_{k,j}$
6:     end for
7:   end for
8: end for
```

2.3.1 Dropping based on the level of fill

When we let the zero pattern G be exactly the zero pattern of A, i.e.,

$$G = \{(i,j) \mid a_{ij} = 0, i \neq j\}$$

we obtain the ILU factorization *with no fill-in* or ILU(0), where nonzeros are only allowed in the LU factors at the positions where the corresponding entries in A are nonzero. The accuracy of the ILU(0) preconditioning may not be sufficient in many cases. A more accurate ILU(k) factorization allows more fill-ins by using the notion of the *level of fill*, which is defined as follows. Initially, the level of fill of each position is

$$\begin{cases} lev_{i,j} = 0 & a_{i,j} \neq 0 \quad \text{or} \quad i = j \\ lev_{i,j} = \infty & \text{otherwise} \end{cases} . \tag{2.10}$$

In Algorithm 1, when $a_{i,j}$ is modified (line 5), the corresponding $lev_{i,j}$ is updated by

$$lev_{i,j} = \min\left(lev_{i,j}, lev_{i,k} + lev_{k,j} + 1\right). \tag{2.11}$$

With the above definition of the level of fill, the ILU(k) factorization is then obtained by dropping the nonzeros with the levels greater than k. Apparently, the ILU(0) factorization is a special case of the ILU(k) factorization with $k = 0$.

In a practical implementation of the ILU(k) factorization, it was found more efficient to first perform a *symbolic* factorization separated from the numerical factorization. In the symbolic factorization, the level of fill of each entry is computed so that the structure of the LU factors will be fully determined. Therefore, the numerical factorization can be computed efficiently with a static nonzero pattern.

There is another way to interpret the level of fill and the ILU(k) factorization based on the graph model of the Gaussian elimination process. Let $\mathcal{G}(A)$ denote the adjacency graph of A and $\mathcal{G}(L+U)$ be the graph of $L+U$, which is called the *filled graph* of A. The following results to determine the locations of the fill entries by examining $\mathcal{G}(A)$ are well-known in sparse direct methods [2, 23].

Definition 2.3.1 (fill path) *A fill-path is a path in $\mathcal{G}(A)$ between vertex i and vertex j such that each vertex k in this path excluding i and j is numbered smaller than both i and j, i.e., $k < \min(i, j)$.*

Theorem 2.3.1 (fill path theorem) *For an LU factorization $A = LU$ and neglecting numerical cancellation, the (i, j)th entry of the filled matrix $L + U$ is nonzero if and only if there exists a fill path between i and j in $\mathcal{G}(A)$.*

The analogous fill path theorem for the level of fill defined in (2.10) and (2.11) is stated below. As it turns out, a nonzero entry in the filled matrix with the level of fill p corresponds to a shortest fill path of length $p+1$ in $\mathcal{G}(A)$.

Theorem 2.3.2 (level fill path theorem) *For an LU factorization of $A = LU$ and neglecting numerical cancellation, with the level of fill defined in (2.10) and (2.11), a nonzero (i, j)th entry of the filled matrix $L + U$ has a level of fill p if and only if there exists a shortest fill path of length $p + 1$ between i and j in $\mathcal{G}(A)$.*

A practical use of Theorem 2.3.2 is to compute the structure of an ILU(k) factorization [24]. Note that the standard symbolic factorization for ILU(k) requires a procedure akin to Algorithm 1, which only computes the levels of fill and thereby determines the structure of the LU factors. In a row-wise algorithm, computing the nonzero pattern of row i will require all the previously computed rows, 1 to $i-1$, so that this algorithm is inherently serial. On the other hand, symbolic factorization based on the level fill path requires only $\mathcal{G}(A)$. Therefore, the structure of each row can be determined independently, which gives this algorithm an appealing fine-grain parallelism property.

An algorithm based on a breadth first search (BFS) [25] for finding the structure of the U part was presented in [24]. For the completeness of the discussion, we give the full algorithm for finding the structure of both the L part and the U part in Algorithm 2. We first clarify some notations which will be used in this algorithm. The BFS is implemented via a queue, Q, where Enqueue and Dequeue are standard queue operations (see, e.g., [25] for details). $i \rightsquigarrow u$ denotes a path in $\mathcal{G}(A)$ starting from i and ending at u. $\max(i \rightsquigarrow u \setminus i)$ denotes the node with the largest index in this path excluding i, and $|i \rightsquigarrow u|$ is the length of this path. visit(v) indicates if v has been seen or not in the BFS, and mark(v) tells if a shortest fill path from i to v has been found. adj(v) denotes the adjacent nodes of v in $\mathcal{G}(A)$, i.e., the indices of the nonzeros in row v of A except v itself. lower(i) and upper(i) are two sets of the indices of the nonzero entries in the strict lower and upper part of row i of $L + U$ respectively.

For row i, the algorithm starts with Q containing only node i, which is the source of the BFS. As in the standard BFS, we always remove the head of Q, which is the node u in line 4. Then, we find the node with the largest index, μ, in the path $i \rightsquigarrow u \setminus i$ and the length, λ, of this path (when $i \rightsquigarrow u \setminus i$ is empty, we set $\mu = -1$ and $\lambda = 0$). If $\lambda > k$, this indicates the finishing of the entire BFS of depth at most $k + 1$. Otherwise, we examine each v in adj(u) except i. If it is the first time that we see v, visit(v) will be set. In this case, if v is numbered larger than i, a shortest fill path to v is found and v should appear in upper(i). Here, we do not need to add v to Q because it is not possible that a longer fill path can be found which containing the fill path from i to v (since $v > i$). If $v < i$, we test if v is labeled larger than the largest index in $i \rightsquigarrow u \setminus i$, which is μ, in line 16. If so, a shortest fill path to v is found and v should appear in lower(i). In either case, we need to insert v into Q. On the other hand, if v is a node which has been seen before (i.e., visit(v) = true) and a shortest fill path to v has not been found (mark(v) = false), in this case we must have $v < i$ and v should be insert to Q. If $v > \mu$, v is added to lower(i).

It is worth to point out that we do not need to consider the case when v has been visited and also marked. When the first time v was visited and marked via $i \rightsquigarrow_1 v$, we must have $\max(i \rightsquigarrow_1 v \setminus \{i, v\}) < v$, since $i \rightsquigarrow_1 v$ is a fill path. When v is revisited via another path $i \rightsquigarrow_2 v$, from the order of the vertices seen in BFS, we have $|i \rightsquigarrow_2 v| \geq |i \rightsquigarrow_1 v|$. Consider a fill path from i to some node k, $i \rightsquigarrow_2 v \rightsquigarrow k$, which contains $i \rightsquigarrow_2 v$. Clearly, we have $v < k$. Since $\max(i \rightsquigarrow_1 v \setminus \{i, v\}) < v$, it follows that $\max(i \rightsquigarrow_1 v \setminus i) < k$. This actually tells us that $i \rightsquigarrow_1 v \rightsquigarrow k$ is also a fill path and is shorter than $i \rightsquigarrow_2 v \rightsquigarrow k$. Since we look for a shortest fill path to k, $i \rightsquigarrow_2 v$ can be safely ignored.

Algorithm 2 A BFS algorithm for computing the structure of row i of ILU(k)

```
Enqueue($Q$, $i$)
lower($i$) = upper($i$) = $\emptyset$
while $Q \neq \emptyset$ do
  $u$ = Dequeue($Q$)
  $\mu = \max(i \rightsquigarrow u \setminus i), \quad \lambda = |i \rightsquigarrow u| - 1$
  if $\lambda > k$ then
    break                                          {stop the while-loop}
  end if
  for each $v \in$ adj($u$) and $v \neq i$ do
    if visit($v$) = false then
      visit($v$) = true
      if $v > i$ then
        add $v$ to upper($i$)
        mark($v$) = true
      else
        if $v > \mu$ then
          add $v$ to lower($i$)
          mark($v$) = true
        end if
        Enqueue($Q$, $v$)
      end if
    else                                           {visit($v$) = true}
      if mark($v$) = false then
        if $v > \mu$ then
          add $v$ to lower($i$)
          mark($v$) = true
        end if
        Enqueue($Q$, $v$)
      end if
    end if
  end for
end while
```

In Table 2.1, we report the performance of the standard symbolic factorization algorithm for ILU(k) and the one based on the BFS algorithm, and the performance of solving the linear systems with the GMRES method preconditioned by ILU(k). In the header of this table, 'N' is the size of the matrix, 'NNZ' is the number of the nonzeros, 'fill' is the fill-ratio of the factorization, i.e., the ratio of the number of the nonzeros of the factors to the number of the nonzeros of the original matrix, 'nits' stands for the number of iterations required, and T_{sym} and T_{num} are the times (in seconds) for the symbolic and numerical factorizations respectively. The test cases include a 2D 5-point Laplacian matrix and a 3D 7-point Laplacian matrix, namely Lap2D and Lap3D, and three matrices from University of Florida sparse matrix collection [26]. The experiments were conducted on a machine equipped with two quad-core CPUs. The BFS-based symbolic factorization was parallelized by OpenMP [27].

For each matrix, the symbolic factorization algorithms were tested with two different k values. As shown by the results, the BFS-based algorithm (denoted by 'bfs') achieved higher efficiency than the standard one (denoted by 'std') in this parallel environment except for the 2D Laplacian matrix, in which case, however, the symbolic factorization of ILU(k) is inexpensive with a moderate k.

2.3.2 Dropping based on threshold

The motivation behind the dropping strategy based on the level of fill is that for diagonally dominant matrices, an element of a higher level of fill tends to be smaller in magnitude. Therefore, dropping these elements can keep the entries in the residual R in (2.9) small. However, for matrices that are far from being diagonal dominant, this dropping criterion may not be appropriate. Instead, the ILU factorization with threshold dropping (often referred to as ILUT) uses an alternative strategy to drop fill-ins, where the entries are dropped if their magnitudes are smaller than a given threshold. Practical implementations of ILUT often use the drop tolerance τ in a relative manner: in a row-wise algorithm, an entry will be dropped if its magnitude is less than $\tau\|a_i\|$, where a_i denotes the i-th row of A. Moreover, there is a second parameter p often included in addition to τ, which determines the largest number of nonzeros allowed per row. After computing a row of the factorization with the application of the drop tolerance τ, the second dropping rule is applied by only keeping the p largest elements in magnitude in

Table 2.1: Performance of the BFS-based ILU(k) symbolic factorization.

Matrix	N	NNZ	ILU(k)		fill	nits	T_{sym}	T_{num}
Lap2D	1,000,000	4,996,000	$k=3$	std	2.6	475	0.20	0.17
				bfs			0.30	
			$k=5$	std	4.2	221	0.44	0.45
				bfs			0.45	
Lap3D	1,000,000	6,940,000	$k=3$	std	6.0	44	1.56	2.87
				bfs			0.80	
			$k=5$	std	14.8	31	8.18	14.6
				bfs			2.85	
af23560	23,560	484,256	$k=3$	std	7.6	17	0.38	0.49
				bfs			0.10	
			$k=5$	std	12.6	11	0.97	1.13
				bfs			0.21	
ship_003	121,728	8,086,034	$k=3$	std	4.2	255	6.10	7.12
				bfs			1.85	
			$k=5$	std	7.6	164	19.1	22.5
				bfs			6.20	
Poisson3D	85,623	2,374,949	$k=1$	std	8.0	36	3.71	7.85
				bfs			0.84	
			$k=2$	std	29.0	22	35.8	86.58
				bfs			2.89	

order to have a direct control of the memory usage. This approach is referred to as the dual threshold ILU factorization [28]. Implementation details of this approach can be found in [12, 28].

2.3.3 Inverse-based dropping

In previous approaches to ILU factorizations, the basic idea was to minimize the error of the factorization, or in other words, to keep the entries of R in (2.9) small. However, it turned out that it is often more important to monitor the errors in the inverses of the factors [29, 30], because these errors have a direct impact on the preconditioned matrix. For an ILU factorization, $A \approx \tilde{L}\tilde{U}$, we write the inverses of the factors as

$$\tilde{L}^{-1} = L^{-1} + X, \quad \text{and} \quad \tilde{U}^{-1} = U^{-1} + Y,$$

where $A = LU$ is the (exact) LU factorization. Then, the preconditioned matrix reads

$$\tilde{L}^{-1}A\tilde{U}^{-1} = (L^{-1} + X)A(U^{-1} + Y) = I + UY + XL + XAY,$$

from which it is clear that the entries of X and Y need to be small in order to make the preconditioned matrix close to the identity matrix. However, *dropping small elements in L and U may yield arbitrarily large terms in X and Y* [30], especially in the cases when L and U are ill-conditioned.

The inverse-based dropping strategy given in [29, 30] is a greedy algorithm that considers the dropping of one element at a time. Suppose that in an ILU factorization, L is computed column by column, while U is computed row by row. Therefore, at the finish of the k-th step of the factorization, the first k columns of L and the first k rows of U have already been computed, and the last $n-k$ columns of L and the last $n-k$ rows of U are the corresponding columns and rows of the identity matrix. We denote by L_k and U_k the L and U matrices obtained after the k-th step before applying any dropping. Then, consider the situation when a term l_{jk} ($j > k$) is replaced with zero. The perturbed matrix can be written as

$$\tilde{L}_k = L_k - l_{jk}e_je_k^T = L_k(I - l_{jk}e_je_k^T),$$

where e_j and e_k are the j-th and k-th canonical basis vectors respectively, and we use the fact that $L_ke_j = e_j$. Noticing that $(I - l_{jk}e_je_k^T)^{-1} = (I + l_{jk}e_je_k^T)$, we see that the inverse of L_k is perturbed by

$$\tilde{L}_k^{-1} = (I + l_{jk}e_je_k^T)L_k^{-1} = L_k^{-1} + l_{jk}e_je_k^TL_k^{-1}.$$

This tells us that dropping l_{jk} from L_k will result in a perturbation to the j-th row of L_k^{-1} by $l_{jk}e_k^TL_k^{-1}$. Hence, a better dropping strategy for l_{jk} is comparing $|l_{jk}|\|e_k^TL_k^{-1}\|$ against the drop tolerance than looking at $|l_{jk}|$ only. Here, the k-th row of L_k^{-1}, $e_k^TL_k^{-1}$, is required, which is computable since it only requires the leading $k \times k$ submatrix of L_k. Recall that the first k columns of L_k are assumed to be available at step k. In the MatLab notation, vector $z_k^T = e_k^TL_k^{-1}$ can be computed by

$$\mathtt{z_k(1:k) = e_k(1:k)' \,/\, L_k(1:k,1:k);} \qquad \mathtt{z_k(k+1:n) = 0;} \tag{2.12}$$

which is a $k \times k$ sparse triangular solve with a sparse right-hand side. Similarly, the dropping criterion for u_{kj} ($k < j$) is checking $|u_{kj}|\|U_k^{-1}e_k\|$ against the drop tolerance, where the k-th column of U_k^{-1} is required. Instead of computing $e_k^T L_k^{-1}$, techniques used by condition number estimators were adapted in [29, 30] for just estimating the infinity norm, $\|e_k^T L_k^{-1}\|_\infty$ by

$$\|e_k^T L_k^{-1}\|_\infty \approx \frac{|e_k^T L^{-1} b|}{\|b\|_\infty},$$

where b is a vector of ± 1 which is dynamically built to make the k-th component of $L^{-1}b$ large in magnitude. Details of this method can be found in [30]. Compared with computing $e_k^T L_k^{-1}$ in (2.12), this method is much less expensive, which almost adds no cost per step.

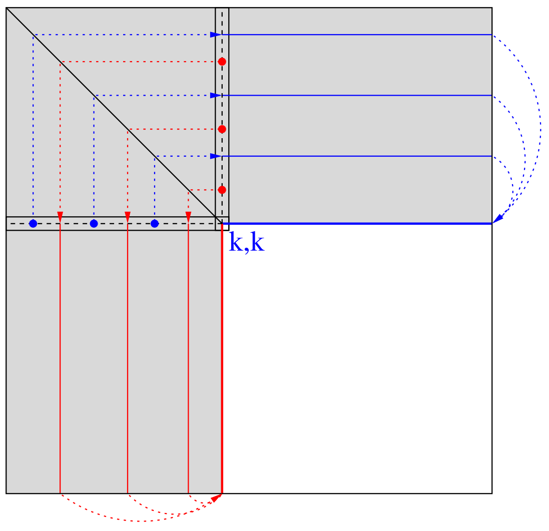
An appropriate ILU factorization algorithm to incorporate the inverse-based dropping is the one based on the Crout version of Gaussian elimination [30] shown in Algorithm 3, where the factorization is computed in the order required by the inverse-based dropping, i.e., at a given step k of the factorization, the first k columns of L and the first k rows of U are available.

Algorithm 3 Crout variant of LU factorization [30]

```
1: for $k = 1, \dots, n$ do
2:     for $i = 1, \dots, k-1$ and if $a_{k,i} \neq 0$ do                {row $k$ of $U$}
3:         $a_{k,k:n} = a_{k,k:n} - a_{k,i} a_{i,k:n}$
4:     end for
5:     for $i = 1, \dots, k-1$ and if $a_{i,k} \neq 0$ do                {column $k$ of $L$}
6:         $a_{k+1:n,k} = a_{k+1:n,k} - a_{i,k} a_{k+1:n,i}$
7:     end for
8:     $a_{k+1:n,k} = a_{k+1:n,k} / a_{k,k}$
9: end for
```

At step k, the k-th column of L and the k-th row of U are computed. Column k of L is a combination of the lower triangular part of column k of A and column j of L with $u_{j,k} \neq 0$, $1 \leq j < k$. Analogously, row k of U is computed by combining the upper triangular part of row k of A and row j of U with $l_{k,j} \neq 0$, $1 \leq j < k$. This step is represented schematically in Figure 2.1. A Crout version of ILU factorization, termed ILUC, can be obtained by applying certain dropping strategy in Algorithm 3. The details for efficient implementation of ILUC can be found in [30]. The Crout version of Gaussian elimination was also used to develop incomplete Cholesky (IC) factorization or ILU factorization for symmetric matrices [31–33].

Figure 2.1: Step k of the Crout LU that computes column k of L and row k of U.



2.3.4 Existence and stability

ILU factorizations of A may breakdown due to zero pivots even when A admits an LU factorization without pivoting [13]. When SPD preconditioners are needed, IC factorizations will also fail when nonpositive pivots are encountered. The existence and numerical stability of ILU factorizations for M-matrices and IC factorizations for symmetric M-matrices were proved in [34] for *any zero pattern* G in (2.8). Moreover, it was also shown that the splitting, $A = LU - R$ is a regular splitting such that the corresponding basic iteration,

$$LUx_{i+1} = Rx_i + b, \tag{2.13}$$

will converge to the solution of $Ax = b$, for every choice of x_0. This implies that when using $M = LU$ as a preconditioner, the triangular solve will be stable. These results were extended for H-matrices [35–37]. However, IC factorizations may fail for general SPD matrices [13, 38]. Remedies to this issue include the following strategies, which can guarantee the existence of IC factorizations for SPD matrices. The first approach is to use local diagonal perturbations [38], in which an encountered nonpositive diagonal entry is replaced with some positive number. A second approach uses a global diagonal shift, which often works better than the local ones. The shifted IC factorization proposed in [35] used a shift of the form $\hat{A} = A + \alpha D$, where D is the matrix that consists of the diagonal of A, and thereafter an IC factorization was performed on $\hat{A}$. The scalar α was selected by a trial-and-error strategy to find the smallest α such that the IC factorization exists. Similar approaches were also used in [31,32,39,40]. The diagonally compensated reduction scheme [41], where positive off-diagonal entries are zeroed and added to the diagonal, also falls into this category of approaches. A third approach proposed in [36, 42] adjusts the diagonal entries that are associated with the dropped terms. Specifically, for any dropped term, namely, $r_{i,j}$, a quantity $|r_{i,j}|$ is added to both the i-th and the j-th diagonal entries. Therefore, in this way, any positive definite matrix admits an incomplete factorization $A = LU - R$ for every zero pattern, where $U_{i,i} > 0$ for all i and R is positive semidefinite. Furthermore, for H-matrices or SPD matrices, the basic iteration (2.13) with the LU factors from this approach is convergent [36], so that the triangular solve is stable.

On the other hand, the improved robustness by these strategies comes with a price in the sense that the quality of the resulting preconditioners may deteriorate, giving a performance that is often worse than that of the standard ILU preconditioners for the cases where the standard ILU factorizations do not breakdown (c.f., the performance comparison shown in [13]).

When applying ILU factorizations to indefinite matrices, the issues of breakdown and numerical stability are generally more severe and common. Furthermore, there may be another problem which has not yet been addressed: the LU factors may be much more ill-conditioned compared with A, so that even when the factorization process itself does not fail and has no severe instability issue (without the presence of near-zero pivots), the unstable triangular solves can also make the preconditioner ineffective [40,43,44]. Common difficulties that are often encountered in ILU factorizations for indefinite problems are summarized below [44].

1. *Inaccuracy and instability due to very small pivots.*

2. *Unstable triangular solves.*

3. *Inaccuracy due to dropping.*

4. *Zero pivots.*

Strategies discussed above to stabilize IC factorizations for SPD cases can also be applied to general matrices. Other treatments include (combinations of) the following techniques.

1. *Row, column, or diagonal scaling.*

2. *Symmetric reordering.* Fill-reducing reordering methods that are popular in sparse direct methods can also be used to improve the accuracy and stability of ILU factorizations. Reordering methods that are "local" in the sense that adjacent nodes are not numbered far apart, such as the bandwidth-reducing orderings were often found helpful [45]. Examples are the reverse Cuthill-McKee (RCM) ordering and the Fiedler ordering [46]. The minimal-degree (MD) type methods and the nested dissection (ND) method are generally not advocated for ILU factorizations with few fill-ins, since they often lead to worse rates of convergence [45, 47, 48]. However, when more fill-ins are allowed, these reorderings can be helpful, which can improve the convergence at a lesser cost of memory [12, §10.6]. A more recent reordering technique based on multilevel graph-coarsening was introduced in [10] to improve the quality of ILU. Unlike the methods mentioned above, this method finds the ordering not only based on the structure of the matrix but also exploiting the algebraic properties.

3. *Nonsymmetric reordering.* By and large, the stability issues in an ILU factorization process and the triangular solve are due to the lack of diagonal dominance of A. Nonsymmetric reordering algorithms which can permute large entries to the diagonal [49, 50] can be used to enhance the diagonal dominance. These methods are often used in combination with a symmetric reordering to reduce fill-in.

4. *Pivoting.* Another technique to handle small pivots in ILU factorizations is to use pivoting. In a row-wise ILU factorization, column (partial) pivoting can be performed efficiently by selecting the largest entry in magnitude in a given row to be the diagonal entry. In practice, we do not need to actually permute the columns. Instead, only column indices need to be exchanged in a permutation vector. ILUT with partial pivoting is referred to as ILUTP (for details, see [51]).

Techniques 1-3 are preprocessing steps before computing an ILU preconditioner, which can be also helpful for other types of preconditioning methods, e.g., the approximate inverse preconditioners. A drawback of techniques 3-4 is that for structured matrices, they may completely destroy the structure, which might results a much higher fill-in.

2.3.5 Multilevel ILU

In essence, a multilevel variant of ILU factorizations is an approach based on matrix reordering and partial Gaussian elimination. Compared with standard ILU preconditioners, convergence rates of multilevel ILU methods are often superior, especially for hard problems. Another advantage of these methods is the parallel efficiency, which makes them more scalable than the standard ILUs for problems of large scales. A number of methods based on multilevel ILUs have been developed, among which we cite [52–64].

In a two-level method, the unknowns are split into two sets and correspondingly the linear system is reordered into the block form

$$\begin{pmatrix} B & F \\ E^T & C \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{2.14}$$

from which, a block LU factorization follows

$$\begin{pmatrix} B & F \\ E^T & C \end{pmatrix} = \begin{pmatrix} I & \\ E^T B^{-1} & I \end{pmatrix} \begin{pmatrix} B & F \\ & S \end{pmatrix}, \tag{2.15}$$

where S is the Schur complement $S = C - E^T B^{-1} F$. From (2.15), it is clear that solving the linear system will require two solves with B and one solve with S. For the splitting (2.14), the goal is to let B have some "nice" properties in terms of the LU factorization. When being used as a preconditioner, an ILU factorization, $B \approx L_B U_B$, is often used to reduce the computational cost and the memory requirement, and thus S can be computed by $S \approx C - (E^T U_B^{-1})(L_B^{-1} F)$, where S can be kept sparse by dropping small terms in the process of the computation. A multilevel ILU method can be obtained by recursively applying the two-level method to S. When the number of levels is p, for each level i with $i < p$, we have

$$P_i A_i P_i^T = \begin{pmatrix} B_i & F_i \\ E_i^T & C_i \end{pmatrix} \approx \begin{pmatrix} L_{B_i} & \\ E_i^T U_{B_i}^{-1} & I \end{pmatrix} \begin{pmatrix} U_{B_i} & L_{B_i}^{-1} F_i \\ 0 & A_{i+1} \end{pmatrix}, \tag{2.16}$$

where P_i is the permutation matrix, and A_{i+1} is an approximate Schur complement. At level p, an ILU factorization is used for A_p which is assumed to be inexpensive.

The preprocessing stage of a multilevel ILU factorization consists of conducting the following steps at each level but the last one: (1) finding the reordering, (2) permuting the matrix, (3) factoring B, and (4) computing the next level matrix, and at the last level, factoring $P_p A_p P_p^T$. Suppose that the original linear system $Ax = b$ is reordered by applying all the permutations, $P_1, \ldots, P_p$. Then, the solution method of the multilevel ILU factorization is given by function $x = \texttt{MILUSolve}(1, b)$ shown in Algorithm 4.

Algorithm 4 p-level ILU solve: $x_i = \texttt{MILUSolve}(i, b_i)$

```
1: if $i = p$ then
2:     Solve $A_i x_i = b_i$
3: else
4:     Compute $g_i' = g_i - E_i^T U_B^{-1} L_B^{-1} f_i$
5:     Recursive call: $v_i = \texttt{MILUSolve}(i+1,\, g_i')$
6:     Compute $u_i = U_B^{-1} L_B^{-1} (f_i - F_i v_i)$
7:     Return $x_i = (u_i^T, v_i^T)^T$
8: end if
```

Different strategies to select the B matrix have been developed, some of which are discussed as follows. In the ILUM method [60] and the MRILU method [54], B was selected as a diagonal matrix, which corresponds to an independent set (IS) of the vertices in the adjacency graph of A. This approach was extended in the BILUM method [59], which allows dense block independent sets (BIS) of size 2, and was further extended into BILUTM [61] and the ARMS method [62] with general BIS. The BIS ordering of B can be obtained by nested dissection (ND) [65]. ND-based multilevel ILU was discussed in [62]. The issue of the stability in the factorization of B was considered in [62] by rejecting the rows which lack diagonal dominance in B. A reordering algorithm to select B based on the diagonal dominance was developed in [66] and an extension to nonsymmetric orderings can be found in [67, 68]. Another type of methods to select C and B is to use the idea of the C/F splitting [8, 9], i.e., the splitting of coarse nodes and fine nodes, used in the algebraic multigrid (AMG) method, which is based on the interpolation or the aggregation between the coarse nodes (the nodes in C) and the fine nodes (the nodes in B). Indeed, multilevel ILU methods can be reformulated as a form of AMG. Connections between these two types of methods have been established, see, e.g. [8, 12, 69–73]. In the following, we simply show their connections in the two-level (two-grid) case. Let A_h, I_h^H and I_H^h denote the fine-grid operator, the restriction operator and the prolongation operator respectively with

$$A_h = \begin{pmatrix} B & F \\ E^T & C \end{pmatrix}, \quad I_h^H = \begin{pmatrix} -E^T B^{-1} & I \end{pmatrix} \quad \text{and} \quad I_H^h = \begin{pmatrix} -B^{-1}F \\ I \end{pmatrix}. \tag{2.17}$$

It will be easy to see that the coarse-grid operator A_H, which is computed by the *Galerkin condition*, is the Schur complement S, i.e.,

$$S = A_H \equiv I_h^H A_h I_H^h. \tag{2.18}$$

Moreover, the following proposition shows that with the above definitions, the solution method with the block factorization (2.15) is equivalent to an AMG-like two-grid cycle, for solving linear system $A_h x_h = b_h$.

Proposition 2.3.1 *With A_h, A_H, I_h^H and I_H^h defined in* (2.17) *and* (2.18)*, the two-grid cycle shown below solves the system $A_h x_h = b_h$.*

1: On the fine grid, use the initial guess $x_h = 0$, so the residual $r_h = b_h$;
2: Restrict r_h to the coarse grid by $r_H = I_h^H r_h$;
3: Solve $A_H e_H = r_H$ to obtain the coarse-grid error e_H;
4: Interpolate e_H by $e_h = I_H^h e_H$ and correct x_h by $x_h := x_h + e_h = e_h$;
5: Compute residual $r_h = b_h - A_h x_h$, solve $A_h e_h = r_h$, and correct x_h by $x_h := x_h + e_h$.

Moreover, this approach is equivalent to the following approach which uses the block factorization (2.15) *with $x_h = \left(u_h^T, v_h^T\right)^T$ and $b_h = \left(f_h^T, g_h^T\right)^T$ that are partitioned conformingly.*

a: $g_h' = g_h - E^T B^{-1} f_h$;
b: solve $S v_h = g_h'$;
c: $u_h = B^{-1}(f_h - F v_h)$.

Proof. Steps 1 and 2 in the first approach correspond to step (a) in the second approach and step 3 corresponds to step (b). Step 4 in the first approach actually computes $u_h = -B^{-1} F v_h$. The system needed to be solved at step 5 is

$$\begin{pmatrix} B & F \\ E^T & C \end{pmatrix} \begin{pmatrix} \delta u_h \\ \delta v_h \end{pmatrix} = \begin{pmatrix} f_h \\ E^T B^{-1} f_h \end{pmatrix},$$

and the solution is $\delta u_h = B^{-1} f_h$ and $\delta v_h = 0$. Therefore, with $u_h := u_h + \delta u_h$, steps 4 and 5 correspond to step (c) in the second approach. □

2.4 Domain decomposition

The Domain Decomposition (DD) approach gives rise to efficient methods for solving linear systems arising from the discretization of 2-D/3-D partial differential equations (PDEs) for parallel computers. In these methods, the problem for the entire domain is divided into smaller subdomain problems, from the solutions of which the solution of the original problem, or an approximate solution as in the case of a preconditioning method, can be produced. When dealing with non-PDE problems or PDE problems without an underlying mesh, DD methods can be realized by using graph partitioning algorithms [74–79], where the graph is the adjacency graph of the coefficient matrix. The metrics for the quality of partitioning a graph, for example, maximizing the intra-subdomain volumes or minimizing the inter-subdomain couplings, are usually also appropriate for DD methods for solving linear systems. Therefore, a good graph partitioner will be, in general, a prerequisite for the success of parallel preconditioners based in DD methods. In this chapter, we consider DD methods for general matrices from a purely algebraic point of view, based on the concept of general graph partitionings.

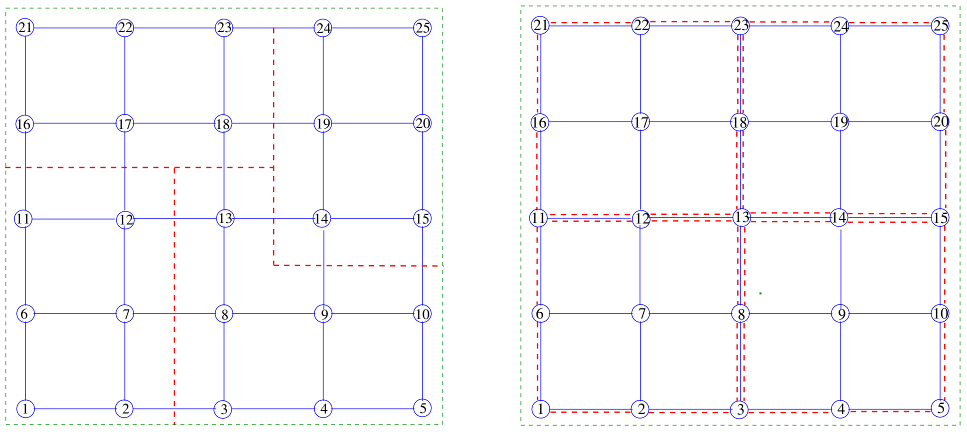
2.4.1 Two types of graph partitionings

Figure 2.2 shows two standard ways of partitioning a graph. On the left side is a *vertex-based* partitioning which is common in the sparse matrix computation community where it is also referred to as graph partitioning by *edge-separators*. A vertex is an equation-unknown pair and the vertex set is subdivided into p partitions, i.e., p non-overlapping subsets whose union is equal to the original vertex set. On the right side is an *edge-based* partitioning, which, in contrast, consists of assigning edges to subdomains. This is also called graph partitioning by *vertex separators* in the graph theory community.

From the perspective of a subdomain, the local unknowns are those belong to this subdomain in contrast with the external unknowns. In a vertex-based partitioning, one can distinguish the following 3 types of unknowns:

1. interior unknowns that are coupled only with local unknowns,

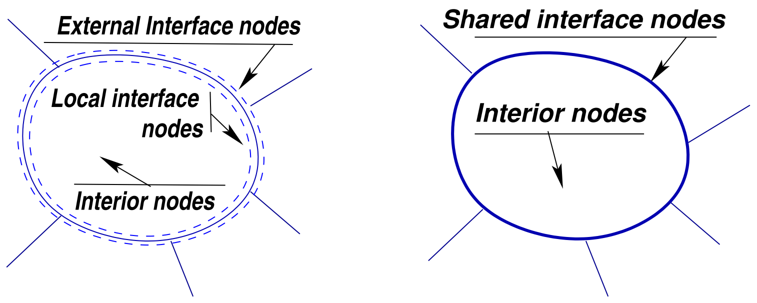
Figure 2.2: Two classical ways of partitioning a graph, vertex-based partitioning (left) and edge-based partitioning (right).



2. local interface unknowns that are coupled with both external and local unknowns,
3. external interface unknowns that belong to other subdomains and are coupled with local interface unknowns.

In an edge-based partitioning, the local and external interface unknowns are merged into one set consisting all the nodes that are *shared* by a subdomain and its neighbors while the interior nodes are those nodes that are *not shared*. This is illustrated in Figure 2.3.

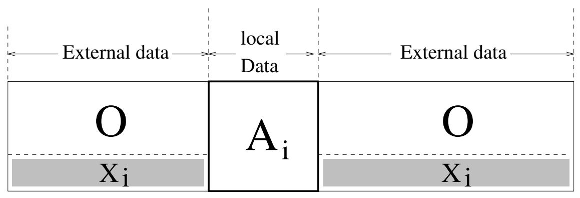
Figure 2.3: A local view of a partitioned domain: vertex-based partitioning (left), edge-based partitioning (right).



2.4.2 Local matrix representation

For both types of partitionings, a set of equations (rows of the linear system) are assigned to each subdomain. If equation i is assigned to a given subdomain, then it is common to also assign unknown i to the same subdomain. Thus, each domain holds a set of equation-unknown pairs. This viewpoint is prevalent when taking a purely algebraic viewpoint for solving systems of equations that arise from PDEs or general unstructured sparse matrices. The rows of the matrix assigned to subdomain i can be split into two parts as shown in Figure 2.4: a local matrix A_i which acts on the local unknowns and an interface matrix X_i which acts on the external interface unknowns for vertex-based partitioning, or the external shared unknowns for edge-based partitioning. The zero-blocks indicate that the interior nodes of different subdomains are decoupled.

Figure 2.4: A local view of a partitioned sparse matrix.



Local unknowns in each subdomain are reordered so that the interface unknowns are listed after the interior ones. Thus, vector of the local unknowns x_i is split into two parts: a subvector u_i of the interior unknowns followed by a subvector y_i of the interface unknowns. Right-hand side b_i is conformingly split into subvectors f_i and g_i. Partitioned according to this splitting, the local system of equations can be written as

$$\begin{pmatrix} B_i & E_i \\ F_i^T & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}. \tag{2.19}$$

Here N_i is the set of the indices of the subdomains that are neighboring to i. The term $E_{ij}y_j$ is a part of the product which reflects the contribution to the local equations from the neighboring subdomain j. The result of this multiplication affects only the local interface equations, which is indicated by the zero in the top part of the second term of the left-hand side of (2.19).

2.4.3 Alternating Schwarz preconditioners

The simplest DD-based preconditioning method might be the Schwarz alternating procedures, where the solution for the entire domain is obtained by alternatingly solving the problems restricted to the partitioned subdomains. The *multiplicative Schwarz method* can be regarded as a generalization of the block Gauss-Seidel method, while analogously a generalization of the block Jacobi approach is the *additive Schwarz method*, which is more amenable to parallel computing [80].

Consider a non-overlapping decomposition of domain Ω such that $\Omega = \bigcup_i \Omega_i$ and $\Omega_i \cap \Omega_j = \emptyset$ for $i \neq j$. Let R_i be the restriction operator from Ω to Ω_i, which consists of the rows of the identity matrix corresponding to the nodes in Ω_i. In practice, the presence of a certain level of overlapping between subdomains is known to improve the overall convergence rate at a higher cost of solving the problem in each subdomain [81–83]. The δ-level overlapping subdomain, Ω_i^δ, is defined recursively as follows. Ω_i^δ is expended from $\Omega_i^{\delta-1}$ by including all the immediate neighboring nodes of $\Omega_i^{\delta-1}$, and $\Omega_i^0 \equiv \Omega_i$. Let $R_{i,\delta}$ denote the restriction operator from Ω to Ω_i^δ (with $R_{i,0} \equiv R_i$). The local solver in the alternating Schwarz methods can be expressed as

$$M_{i,\delta}^{-1} = R_{i,\delta}^T (R_{i,\delta} A R_{i,\delta}^T)^{-1} R_{i,\delta}. \tag{2.20}$$

When apply $M_{i,\delta}^{-1}$ to a vector, it restricts the vector to Ω_i^δ, solve the local problem, and extends the local solution to Ω. Thus, we can write the multiplicative Schwarz method for solving linear system $Ax = b$ as

$$\text{for each subdomain } i\ , \quad x \leftarrow x + M_{i,\delta}^{-1}(b - Ax),$$

and write the additive Schwarz (AS) method as

$$x \leftarrow x + \sum_i M_{i,\delta}^{-1}(b - Ax).$$

A simple improvement to the classical overlapping AS method is the restricted additive Schwarz (RAS) method proposed in [83], where R_i^T is used in the local solver on one side for the *non-overlapping* prolongation, i.e.,

$$M_{RAS(i,\delta)}^{-1} = R_i^T (R_{i,\delta} A R_{i,\delta}^T)^{-1} R_{i,\delta}.$$

Clearly, compared with the classical AS method, the communication required due to the overlapping subdomains is halved, and the RAS method often converges more rapidly.

Two-level and multilevel algorithms

The above alternating Schwarz methods are often referred to as *one-level* methods [81]. A well-known drawback of these methods is that they usually require more iterations to converge as the number of subdomains increases, so that the benefits of the increased parallelism may be outweighed by the increased number of iterations. A remedy to this issue is to use so called *two-level* algorithms. In these algorithms, besides the local solvers, another term that captures the global information is introduced. For example, when combined with the AS method, the two-level algorithm can be written as

$$x \leftarrow x + \left(R_0^T A_C^{-1} R_0 + \sum_i M_{i,\delta}^{-1} \right) (b - Ax),$$

where A_C represents a *coarse grid operator* and R_0 is the corresponding restriction operator. A scheme often used is the *Galerkin coarse grid correction* with $A_C = R_0 A R_0^T$. Therefore, the coarse grid part of the preconditioner is $R_0^T(R_0 A R_0^T)^{-1} R_0$, which has the same form as the local solver (2.20). For matrices obtained from discretized PDEs, the matrix associated with the coarse level of discretization can be used for A_C, while for general matrices, the matrix corresponding to the coarsened graph can be used. When recursively applying this two-level algorithm to A_C, a *multi-level* scheme is then obtained. The intertwining connections between the multigrid methods and the multilevel DD algorithms were demonstrated in [81, §3].

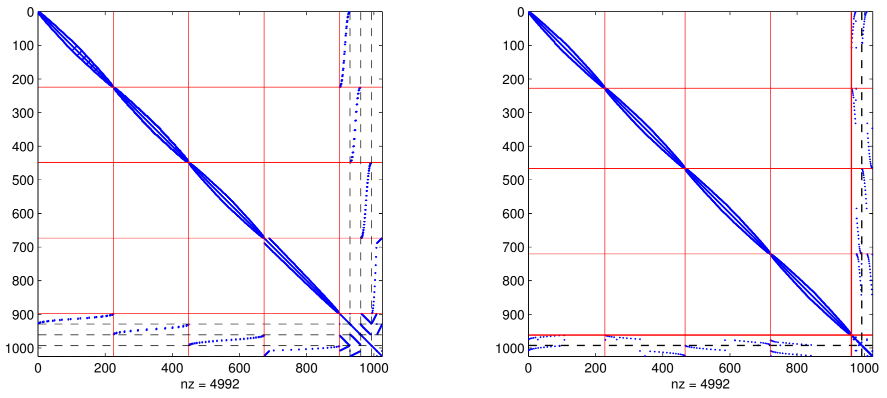
2.4.4 Schur complement approaches

If $\mathcal{Y}_i$ denotes the set of the local interface unknowns of subdomain i, then the global interface $\mathcal{Y}$ is given by $\mathcal{Y} = \bigcup_{i=1}^p \mathcal{Y}_i$. Note that in the case of the vertex-based partitioning, we have $\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset$, for $i \neq j$ such that $y^T = [y_1^T, y_2^T, \cdots, y_p^T]$ and $g^T = [g_1^T, g_2^T, \cdots, g_p^T]$, where y and g are the subvectors of x and b corresponding to $\mathcal{Y}$. If we stack all the interior unknowns $u_1, u_2, \ldots, u_p$ into vector u in this order, and reorder the equations so that u is listed first followed by y, a global system of the following form is obtained,

$$\left(\begin{array}{cccc|c} B_1 & & & & \hat{E}_1 \\ & B_2 & & & \hat{E}_2 \\ & & \ddots & & \vdots \\ & & & B_p & \hat{E}_p \\ \hline \hat{F}_1^T & \hat{F}_2^T & \dots & \hat{F}_p^T & C \end{array}\right) \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_p \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_p \\ g \end{pmatrix}, \tag{2.21}$$

where C is the matrix representation of the global interface $\mathcal{Y}$, and $\hat{E}_i$ and $\hat{F}_i$ define the couplings between subdomain i and $\mathcal{Y}$, which are expanded from E_i and F_i in (2.19) by adding zero columns. An illustration is shown in Figure 2.5 for the vertex-based and the edge-based partitionings of 4 subdomains for a 2-D Laplacian matrix.

Figure 2.5: An example of a 2-D Laplacian matrix which is partitioned into 4 subdomains with edge separators (left) and vertex separators (right).



A popular way of solving a system of equations put into the form of (2.21) is to exploit the Schur complement techniques that eliminate the interior unknowns first and then focus on (approximately) solving in some way for the system which only involves the interface unknowns. This system is referred to as the *reduced system*, of which the coefficient matrix is the *global Schur complement*,

$$S = C - \sum_{i=1}^{p} \hat{F}_i^T B_i^{-1} \hat{E}_i.$$

When the interface unknowns are solved, the interior unknowns can then be easily recovered by back substitution. This defines a preconditioning operation for the global system. This method is also referred to as the *iterative substructuring method* in the structure analysis community.

Vertex-based partitioning

Consider the local system of equations (2.19) obtained in a vertex-based partitioning. Assuming that B_i is nonsingular, u_i can be eliminated by means of the first equation, $u_i = B_i^{-1}(f_i - E_i y_i)$, which yields, upon substitution, the system for the local interface unknowns

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - F_i^T B_i^{-1} f_i \equiv g_i', \tag{2.22}$$

in which S_i is the *local* Schur complement with $S_i = C_i - F_i^T B_i^{-1} E_i$. When written for each subdomain i, (2.22) yields the *global* reduced system, $Sy = g'$, that involves only the interface unknown vectors y_i and has a natural block structure,

$$\begin{pmatrix} S_1 & E_{12} & \dots & E_{1p} \\ E_{21} & S_2 & \dots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p,2} & \dots & S_p \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} g_1' \\ g_2' \\ \vdots \\ g_p' \end{pmatrix}. \tag{2.23}$$

Each of the diagonal blocks in this system is a local Schur complement, which is dense in general. The off-diagonal blocks E_{ij} are identical with those of (2.19) which are sparse. Notice that the global interface matrix C has the same block structure,

$$C = \begin{pmatrix} C_1 & E_{12} & \dots & E_{1p} \\ E_{21} & C_2 & \dots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p,2} & \dots & C_p \end{pmatrix}. \tag{2.24}$$

Parallel and distributed ILU

Parallel and distributed ILU factorization algorithms have been developed by several researchers [84–87]. In these algorithms, DD is applied at the first step. Then the interior unknowns of each subdomain are eliminated independently from those of the other subdomains. The last and the most difficult step is to eliminate the interface nodes, in other words, to factorize the global Schur complement, S. In [12, §12.6] and [84], the distributed ILU(0) of S was discussed, where multi-coloring of the *subdomain graph* was used to process the interface nodes in parallel. In the PILU preconditioner [85], a more advanced parallel ILU(k) algorithm was developed, where fill-ins within and between the subdomains are allowed with certain constrains imposed on the fill-ins between the subdomains. Results in [85] indicated that it is crucial to allow adequate fill-ins between subdomains in order to keep fast convergence rate and also a high degree of parallelism for scalability. In the parallel ILUT preconditioner proposed in [86], the interface nodes are eliminated by iteratively finding a maximal independent set (MIS) in each partially filled graph and processing the nodes in the MIS in parallel.

A more general framework of these methods was established in the parallel algebraic recursive multilevel solver (pARMS) [88]. In pARMS, the solver for B_i is referred to as the *local preconditioner*, which can be ILU or the ARMS method. The reduced system (2.23) is solved approximately by a small number of *inner iterations* of a preconditioned Krylov subspace method, or by just applying the preconditioner. This preconditioner is referred to as the *global preconditioner*. Several global preconditioners have been developed which include block Jacobi [89], block SSOR [90], distributed ILU(0) [88] and the RAS method [91] which is termed the "Schur-RAS" approach. Experimental results for different combinations of these choices can be found in [88].

Chapter 3

Preconditioning techniques for massively parallel SIMD processing

3.1 Introduction

In this chapter, we overview our experience in developing high-performance iterative linear solvers with the acceleration by Graphics Processing Units (GPUs) [19]. Our goal is to illustrate the advantages and difficulties encountered when deploying GPUs to perform sparse matrix computations. The main focus of this chapter will be GPU-efficient preconditioning techniques for iterative methods for solving sparse linear systems. We will begin with two important sparse matrix kernels, the sparse matrix-vector product and the sparse triangular solve, two important building blocks of iterative methods. Later, we will discuss the parallel preconditioners which we have employed on GPUs.

3.2 GPU architecture and CUDA programming

A GPU is built as a scalable array of multi-threaded *Streaming Multiprocessors* (SMs), each of which consists of multiple *Scalar Processor* (SP) cores. Threads are organized in *warps*. A warp is defined as a group of 32 threads of consecutive thread IDs, and a *half-warp* is either the first or second half of a warp. To manage hundreds of threads, the multiprocessors employ a *Single Instruction Multiple Threads* (SIMT) model. Each thread is mapped into one SP core and executes independently with its own instruction address and register state [92]. The *compute capability* of a device is version number that identifies the GPU hardware features. Currently, the highest compute capability is 5.x. For details on the features supported by each compute capability, see [92].

The NVIDIA GPU platform has various memory hierarchies. The types of memory can be classified as follows: (1) off-chip global memory, (2) off-chip local memory, (3) on-chip shared memory, (4) read-only cached off-chip constant memory and texture memory and (5) registers. The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory has a much higher latency (about 400 to 800 clock cycles for devices of compute capability 1.x and 2.x) compared with the on-chip shared memory (32 bits per 2 clock cycle if there are no *bank conflicts*).

In order to improve the overall memory throughput, we should minimize the data transfer from the global memory by maximizing the use of the on-chip memory: the shared memory and the caches. The shared memory can be used as a user-managed cache, which is much faster than the global memory but we also need to pay attention to the problem of *bank conflict*. Cached constant memory and texture memory are also beneficial for accessing reused data and data with 2D spatial locality.

The most efficient way to use the global memory bandwidth is to coalesce simultaneous memory accesses by the threads in a warp into a single or several memory transactions by following the optimal access pattern. For devices of compute capability 1.2 and 1.3, memory access by a half-warp is coalesced as soon as the words accessed by all the threads lie in the same aligned segment of size equal to 128 bytes if all the threads access 32-bit or 64-bit words. Local memory accesses are always coalesced. The alignment and coalescing techniques are explained in detail in [92]. For devices of compute capability 2.0 and higher, the global memory is cached such that the impact of coalescing is reduced by exploiting the data locality.

Programming NVIDIA GPUs for general-purpose computation is supported by the CUDA (Compute Unified Device Architecture) environment. CUDA programs on the host (CPU) invoke a *kernel grid* running on the device (GPU). The same kernel is executed by many threads. The threads are organized into thread blocks, which are distributed to SMs and split into warps scheduled by SIMT units. All the threads in a thread block share the same shared memory and can synchronize themselves by a barrier. Threads in a warp execute one common instruction at a time. This is referred to as the *warp-level synchronization* [92]. Full efficiency is achieved when all the threads in a warp follow the same execution path. Branch divergence causes serial execution.

3.3 GPU-accelerated sparse kernels

The potential of GPUs for sparse matrix computations was recognized early on when GPU programming still required shader languages, see, e.g., [93, 94]. After the advent of CUDA, GPUs have drawn much more attention for sparse matrix computations and solving sparse linear systems. Compared with dense matrix computations, sparse matrix computations are unable to reach a significant percentage of the peak floating point throughput of a machine. This is mainly due to the indirect and irregular memory accesses in the sparse mode.

3.3.1 Sparse matrix-vector product

The first sparse kernel we consider is the sparse matrix-vector product (SpMV) of the form, $y = Ax$, where A is a sparse matrix, and x and y are dense vectors. This kernel is one of the major components of sparse matrix computations accounting for a big part of the cost of sparse iterative linear solvers as an example. In recent years, quite a few GPU-accelerated SpMV algorithms with carefully designed sparse matrix formats have been developed, see [95–105] and the references therein. In this section, we will discuss the implementations of GPU SpMV kernels in several well-known non-blocking formats. This can provide important insights for understanding the algorithmic and implementation principles to achieve high performance in GPU computing.

SpMV in the DIA/ELL format

The DIA (DIAgonal) format is a special format for diagonally structured matrices in which nonzero values are restricted to lie in a small number of diagonals. This format is efficient for memory bandwidth because there is no double indexing as the case in a more general sparse format. Specifically, the column index of an element can be calculated by its row index plus the offset value of that diagonal which is stored in `IOFF`. On the other hand, this format may potentially waste storage since zeros are padded for the non-full diagonals. Ellpack-Itpack (ELL) format is a more general scheme. The assumption in this format is that there are at most d nonzero elements per row, where d is typically small. Then, two arrays of dimension $n \times d$ are required to store the nonzeros elements and the column indices respectively, where n is the size of the matrix. In terms of the storage efficiency, the ELL format will also require space to store more elements than the actual nonzeros unless all the rows have exactly the same number of nonzeros.

As an example, consider the matrix A defined by

$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}. \tag{3.1}$$

Then, the DIA format of matrix A is as follows

`DIAG` =

*	1.	2.
3.	4.	5.
6.	7.	8.
9.	10.	*
11.	12.	*

, `IOFF` =

-1	0	2

and the corresponding ELL format is

`COEF` =

1.	2.	0.
3.	4.	5.
6.	7.	8.
9.	10.	0.
11.	12.	0.

, `JCOEF` =

1	3	1
1	2	4
2	3	5
3	4	4
4	5	5

where the 2-D arrays, `DIAG`, `COEF` and `JCOEF` are assumed to be stored in the column-major order. The entries that are not actually used (i.e., the padded nonzeros) are denoted by the stars. For the DIA kernel, one thread is used for each row and thus consecutive threads can access contiguous memory addresses. The ELL kernel works similarly except that the column indices are loaded from array `JCOEF`. We note here that the DIA and ELL formats make specific assumptions about the nonzero patterns of sparse matrices, which, however, may not be satisfied by general sparse matrices.

SpMV in the CSR format

The compressed sparse row (CSR) format is a general sparse matrix storage format, which does not make any assumption about the nonzero pattern, and the number of elements stored is exactly equal to the number of the nonzeros. A standard CSR format for a sparse matrix that has n rows and m nonzeros can be specified by three arrays: a real/complex data array of length m, namely `AA`, which contains the nonzero elements row by row; an index (integer) array of length m, namely `JA`, which contains the column indices of the nonzeros; and an index (integer) array of length $n+1$, namely `IA`, which contains the starting location of the nonzero entries of each row, i.e., $\texttt{IA}(i)$ points to the first nonzero of row i (if there is any) stored in the arrays `AA` and `JA`, and for the last entry of `IA` we have $\texttt{IA}(n+1) = \texttt{IA}(1) + m$. In this way, the CSR format stores the nonzero entries of each row in contiguous memory. Therefore, it is easy to access the rows of a CSR format matrix, while the column-wise accessing will not be efficient.

For the matrix in (3.1) which has 5 rows and 12 nonzeros, the three arrays in the CSR format are as follows.

`AA` :	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
`JA` :	1	3	1	2	4	2	3	5	3	4	4	5
`IA` :	1	3	6	9	11	13						

To parallelize the CSR format SpMV, we partition the rows among the threads. The computation for each row, which is the dot product between a sparse row vector and the dense vector x, is completely independent. For the CSR SpMV on GPUs, a simple scheme is to assign a thread to each row, which is referred to as the *scalar* CSR kernel [96], in which case the computations of all the threads are independent. The main drawback of this scheme is that threads with consecutive IDs will access the matrix at noncontiguous memory addresses, which are apart from each other by the spacing indicated by two adjacent row pointers in `IA`. As a result, the memory accessing pattern in this scheme will not be efficient since the chance of memory transaction coalescing is significantly reduced.

A better approach, termed *vector* CSR kernel, introduced in [95, 96] is to assign a group of threads (e.g., a half-warp, or a warp) instead of only one thread to each row.

In this approach, since all the threads in a group access the nonzero elements of the same row, which are more likely to belong to the same memory segment, the chance of coalescing the memory transactions will be much higher. In this approach, an essential computation is the *parallel vector dot product* (the dot product between a sparse matrix row and the vector x) per group of threads. This can be accomplished by a *parallel reduction* scheme. The vector CSR kernel using a half-warp per row is shown as follows.

```
__global__
void csr_v_k(int n, int *d_ia, int *d_ja, REAL *d_a, REAL *d_y) {
  // num of half-warps
  int nhw = gridDim.x * BLOCKDIM / HALFWARP;
  // half warp id
  int hwid = (blockIdx.x * BLOCKDIM + threadIdx.x) / HALFWARP;
  // thread lane in each half warp
  int lane = threadIdx.x & (HALFWARP-1);
  // half warp lane in each block
  int hwlane = threadIdx.x / HALFWARP;
  // shared memory for partial result
  volatile __shared__ REAL r[BLOCKDIM+8];
  volatile __shared__ int startend[BLOCKDIM/HALFWARP][2];
  for (int row = hwid; row < n; row += nhw) {
    // row start and end point
    if (lane < 2)  startend[hwlane][lane] = d_ia[row+lane];
    int p = startend[hwlane][0];
    int q = startend[hwlane][1];
    REAL sum = 0.0;
    for (int i=p+lane; i<q; i+=HALFWARP)
      sum += d_a[i-1] * x[d_ja[i-1]-1];
    // parallel reduction
    r[threadIdx.x] = sum;
    r[threadIdx.x] = sum = sum + r[threadIdx.x+8];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+4];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+2];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+1];
    if (lane == 0)  d_y[row] = r[threadIdx.x];
  }
}
```

In this kernel, the shared memory (`r`, line 12) is used for performing the parallel reduction, and storing the starting and the ending positions of each row (`startend`, line 13). In line 16, the first two threads of each half-warp load the two row pointers of the associated row into `startend` in the shared memory from `IA` in the global memory. Then, all the other threads in the same half-warp can read these two values from the shared memory. By the for-loop in lines 20-21, each thread computes its partial sum. In lines 23-27, we first put the partial sum into `r` and then perform the parallel reduction of `r` in the shared memory to sum up all the partial results. Finally, the first thread of each half-warp saves the result to vector y. Moreover, note that there is no synchronization point required in this kernel. This relies on the *warp-level synchronization* [92]: "a warp executes one common instruction at a time, threads within a warp are implicitly synchronized", so that the synchronization in the parallel reduction can be omitted for better performance. Generally speaking, the full efficiency of the above half-warp vector CSR kernel requires at least half the warp size (which is 16) nonzeros per row, while in [96] a full warp is assigned to each row. The optimal number of threads to use per row will depend on the number of nonzeros per row of the matrix.

SpMV in the JAD format

The JAD (JAgged Diagonal) format can be viewed as a generalization of the ELL format which removes the assumption on the fixed-length rows [12]. To build the JAD structure, we first sort the rows by a non-increasing order of the number of the nonzeros per row. Then the first JAD consists of the first element of each row; the second JAD consists of the second element, etc. So, the number of the JADs is the largest number of the nonzeros per row. Consider the matrix in (3.1) and the permuted matrix is then

$$PA = \begin{pmatrix} 3. & 4. & 0 & 5. & 0 \\ 0 & 6. & 7. & 0 & 8. \\ 1. & 0 & 2. & 0 & 0 \\ 0 & 0 & 9. & 10. & 0 \\ 0 & 0 & 0 & 11. & 12. \end{pmatrix}.$$

Then the JAD format of PA is shown as follows.

AA	3	6	1	9	11	4	7	2	10	12	5	8
JA	1	2	1	3	4	2	3	3	4	5	4	5
IA	1	6	11	13								

The JAD format also consists of three arrays in analogy to the CSR format: one real/complex data array `AA` contains the nonzero elements; one index array `JA` contains the column index of each nonzero; and one index array `IA` contains the beginning position of each JAD in `AA` and `JA`. For the SpMV in the JAD format, the computation is partitioned row-wise and only one thread is assigned to each row such that fine-grained parallelism is achieved. Note that since `AA` and `JA` are stored in the JAD order, the JAD kernel will not suffer from the performance drawback as seen in the scalar CSR kernel, which means that consecutive threads can access contiguous memory, which follows the suggested memory access pattern and can improve the memory bandwidth by coalescing the memory transactions. The JAD SpMV kernel is shown as follows.

```
__global__
void jad_k(int n, int njad, int *d_ia, int *d_ja, REAL *d_a, REAL *d_y) {
  int i,j,p,q;  REAL r;
  int row = blockIdx.x*blockDim.x+threadIdx.x;
  int nthreads = gridDim.x * blockDim.x;
  __shared__ int shia[BLOCKDIM];
  if (threadIdx.x <= njad)  shia[threadIdx.x] = d_ia[threadIdx.x];
  __syncthreads();
  while (row < n) {
    r = 0.0;  i = 0;
    p = shia[0];  q = shia[1];
    while ( ((p+row) < q) && (i < njad) ) {
      j = p+row;
      r += d_a[j-1] * x[d_ja[j-1]-1];
      if (i++ < njad) {
        p = q;  q = shia[i+1];
      }
    }
    d_y[row] = r;
    row += nthreads;
  }
}
```

In this kernel, we assume that the number of the threads in each thread block (the dimension of each thread block), indicated by `BLOCKDIM`, is not smaller than the number of the JADs, `njad`. On the other hand, when `BLOCKDIM` $<$ `njad`, we can compute the whole SpMV by a sequence of the JAD kernels, each of which individually handles at most `BLOCKDIM` JADs. In this kernel, one thread is assigned for each row (line 4). For the nonzero entry at the i-th row and the j-th JAD, the value and the column index can be obtained by $\texttt{AA}(\texttt{IA}(j)+i)$ and $\texttt{JA}(\texttt{IA}(j)+i)$. In contrast with the CSR format, the entire `IA` array for all the JADs is required by each thread for the computations with each row. Therefore, within each thread block, we prefetch `IA` from the global memory into the array `shia` in the shared memory (line 7), such that afterwards the data in `IA` can be retrieved with a low latency. Note that loading `IA` into `shia` is performed in parallel. Therefore, a block-level synchronization is needed (line 8) to guarantee that writing the data into `shia` is complete before using them.

SpMV in the HYB format

The HYB format is a hybrid format proposed in [96]. It achieves the best performance among all the formats in the cuSPARSE [106] and the Cusp libraries [107]. In this format, the nonzeros of a matrix are split into two parts and stored separately in the SIMD-efficient ELL format and a general sparse format, e.g., the CSR format or the coordinate (COO) format where a triplet of the row number, the column number and the value is saved for each nonzero. Obviously, the ideal case is that all the nonzeros can be saved in the ELL part without introducing many artificial nonzeros. Recall that the efficiency of the ELL format depends on if the matrix has roughly the same number of nonzeros per row, and the performance of the ELL kernel will deteriorate if this number varies significantly. Thus, the main idea is to find a *typical* number of nonzeros among all the rows, store these entries in the ELL part and put all the remaining ones into the CSR/COO part. As long as most of the nonzeros can be stored in the ELL part, the HYB kernel is expected to have a high computational throughput close to the ELL kernel. With regard to the number of nonzeros per row used in the ELL part, a simple and effective strategy is to balance the ratio between the number of the unused entries and the number of all the nonzeros stored. For instance, we can put as many as possible of nonzeros in the ELL part while having this ratio lower than a given threshold.

Vector x

For a device of compute capability lower than 2.0, where the global memory is not cached, we can place the vector x in the cached texture memory and access it by texture fetching (for details, see [92]) since in the SpMV the data in x will be reused and are read-only. As reported in [96], texture memory caching can improve the performance by an average of 30% and 25% for single and double precisions respectively.

3.3.2 Sparse triangular solve

The second GPU sparse kernel we consider is the sparse triangular solve (SpTrsv) with a dense right-hand side, which is of the form $x = L^{-1}b$ or $x = U^{-1}b$ where L and U are sparse lower and upper triangular respectively, and x and y are dense. Assume that L is a unit lower triangular matrix stored in the CSR format in arrays (il, jl, l) without the diagonal elements, which are all ones. The forward substitution for solving $Lx = b$ is show as follows, where x is initialized as b.

```
1: for $i = 1, 2, \ldots, n$ do
2:     for $j = il(i), \ldots, il(i+1) - 1$ do
3:         $x(i) \leftarrow x(i) - l(j) \times x(jl(j))$
4:     end for
5: end for
```

The procedure of the backward substitution for solving a unit upper triangular system $Ux = b$ is similar except that the outer loop will be performed in the reverse order instead, i.e., **for** $i = n, n-1 \ldots, 1$ **do**. In the forward/backward substitution process, the outer loop for each unknown is sequential, while the computations in the inner loop can be split and parallelized. In the case when the triangular matrix is stored row-wise, the inner loop computes the dot product of each sparse row of the triangular matrix and the dense vector x. On the other hand, when the matrix is stored in a sparse column format, the inner loop will be the AXPY operation of each sparse column and x. However, for both cases, since the number of the nonzeros involved in the inner loop is typically small, this parallelization approach will not be efficient.

Level scheduling

Better parallelism can be achieved by analyzing the dependency of the unknowns. Unknown i can be immediately determined once all the other ones involved in equation i become available. The underlying graph of a triangular matrix is a directed acyclic graph (DAG). There is an edge $j \rightarrow i$ in the DAG if the (i, j) position of the matrix is nonzero, which indicates that the solution of x_i depends on that of x_j. Therefore, the idea is to group the unknowns into different levels so that all the unknowns within the same level can be computed simultaneously. This level information can be obtained by exploiting the topological sort of the DAG. This approach is referred to as *level scheduling* [12]. For the forward substitution, the level of unknown x_i which is denoted by $lev(i)$ is first initialized by $lev(i) = 0$ for all i, and then can be computed by

$$lev\,(i) = 1 + \max_{j} \{lev\,(j)\}\,, \text{ for all } j \text{ such that } L_{ij} \neq 0, \tag{3.2}$$

for $i = 1, 2, \ldots, n$. For the backward substitution, $lev(i)$ can be computed by performing (3.2) for $i = n, n-1, \ldots, 1$ with L_{ij} replaced with U_{ij}.

Suppose that $nlev$ is the number of levels, q lists the indices of the unknowns by a nondecreasing order of their levels, and $p\,(i)$ gives the position in q of the first unknown of the i-th level. Then, the forward substitution with level scheduling for a CSR format unit lower triangular matrix (il, jl, l) is shown as follows. The second for-loop (line 2) is for computing the unknowns of level m, so that it can be performed in parallel.

```
1: for $m = 1, \ldots, nlev$ do
2:    for $k = p(m), \ldots, p(m+1) - 1$ do {parallel for-loop}
3:       $i = q\,(k)$
4:       for $j = il\,(i)\,, \ldots, il\,(i+1) - 1$ do
5:          $x(i) \leftarrow x(i) - l(j) \times x(jl(j))$
6:       end for
7:    end for
8: end for
```

The analogous backward substitution can be handled similarly. Clearly, we have $1 \leq nlev \leq n$, where n is the size of the system. Thus, the degree of parallelism on average is $n/nlev$. The best case when $nlev = 1$ corresponds to the situation when all the unknowns can be computed simultaneously (e.g., a diagonal matrix). In the worst case, when $nlev = n$, each unknown will be of a different level, such that the forward/backward substitution will be completely sequential. So, the performance of the level scheduling scheme will significantly depend on $nlev$. For the triangular matrices which are Cholesky factors, we have $nlev = h_T$, where h_T is the height of the elimination tree [108], while for incomplete Cholesky factorizations, we have $nlev \leq h_T$. Minimum degree type reordering algorithms, e.g., the Multiple Minimal Degree (MMD) ordering [109] or the Approximate Minimum Degree (AMD) ordering [2,110,111], which can give short and bushy elimination trees, are often helpful in reducing the number of levels.

GPU implementation

We implemented the CSR format level-scheduling SpTrsv kernel on GPUs. The equations are first grouped into different levels. Starting from the first level, each half-warp solves one unknown of the current level in which the vector dot product is computed by the parallel reduction scheme as was done in the vector CSR format SpMV kernel. Between two levels, synchronization across the half-warps is required. If the kernel is launched with only one thread block, synchronization is supported by the function `__syncthreads()`. On the other hand, when there are more than one thread blocks used, for the synchronization we have to launch another kernel for the next level after the one for the current level terminates.

3.4 GPU-accelerated preconditioned iterative methods

For preconditioned Krylov subspace methods for solving linear systems, the following components can be accelerated by GPUs: (1) SpMV, (2) preconditioning operation and (3) level-1 BLAS operations, where for (1) we can use the GPU SpMV kernels, and for (3) cuBLAS [112] can be used. The overall efficiency of these methods will be determined by both the convergence rate and the computational cost per iteration. The convergence rate mainly depends on the quality of the preconditioner. On the other hand, when considering the computational cost, not only the computational complexities (measured by flop counts) but also the efficiency on the underlying computing platform should be taken into account. In this section, some standard SIMD-type parallel preconditioning techniques along with their GPU implementations will be discussed.

3.4.1 ILU preconditioners

The preconditioning operation of an ILU preconditioner is a forward substitution followed by a backward one, i.e., $u = U^{-1}L^{-1}v$. As mentioned in Section 3.3.2, the performance of the parallel triangular solves with level scheduling will degrade when the number of the levels is large, which typically increases with the fill-ins in ILU factorizations. The number of the levels can be reduced if the original matrix is preordered by some minimal-degree type method, which is, however, not advocated for ILU preconditioners with few fill-ins [45, 47, 48], [12, §10.6]. Thus, when a triangular solve cannot be performed efficiently on GPUs, we may consider performing it on the CPU side. As a result, this requires transferring data between CPU and GPU at each iteration.

3.4.2 Block Jacobi preconditioners

For a block Jacobi preconditioner, we compute an ILU factorization individually for each diagonal block. One thread block is assigned to each diagonal block and threads within each thread block can cooperate to solve the local triangular system in parallel with level scheduling and save the local results to the global output vector. There is no global synchronization or communication needed in this preconditioner

3.4.3 Multi-color SSOR/ILU(0) preconditioners

Another strategy to enhance parallelism is to exploit graph coloring. A graph $\mathcal{G}$ is p-colorable if its vertices can be colored in p colors such that no adjacent vertices have the same color. Finding the smallest p, which is called the chromatic number, of a general graph is an NP-complete problem [113]. However, a simple greedy algorithm known as the greedy coloring algorithm can provide an adequate coloring in a number of colors upper bounded by $\Delta(\mathcal{G})+1$, where $\Delta(\mathcal{G})$ denotes the maximum vertex degree. Suppose p colors are found and the vertices are ordered by the colors. The corresponding adjacency matrix has the following structure:

$$\begin{bmatrix} D_1 & & & & \\ & D_2 & & -F & \\ & & \ddots & & \\ & -E & & D_{p-1} & \\ & & & & D_p \end{bmatrix}, \tag{3.3}$$

where the p diagonal blocks are diagonal matrices since vertices of the same color are not adjacent. If $ptr\,(i)$ points to the first row associated with the i-th color, the Multi-Color Successive Over Relaxation (MC-SOR) sweep will take the procedure as follows.

```
1: for $i = 1, \ldots, p$ do
2:     $n_1 = ptr(i)$
3:     $n_2 = ptr(i+1) - 1$
4:     $r(n_1 : n_2) = \omega * (b\,(n_1 : n_2) - A(n_1 : n_2, :) * x)$
5:     $x(n_1 : n_2) = x(n_1 : n_2) + D_i^{-1} * r(n_1 : n_2)$
6: end for
```

The for-loop in line 1 for the p colors is sequential. For each color i, a scaling by the diagonal matrix D_i and a matrix-vector product $A(n_1 : n_2, :) * x$ are performed, both of which can be easily vectorized. The Symmetric SOR (SSOR) sweep consists of the SOR sweep followed by another backward one, i.e, from the last color to the first one. When k-step SSOR iterations are used as a preconditioning method, we denote it by the MC-SSOR(k) preconditioner.

Another method to utilize parallelism from multi-coloring is to compute an ILU(0) factorization of matrix (3.3). This preconditioner has the same parallelism as that of MC-SSOR, which is denoted by the MC-ILU(0) preconditioner. Compared with the ILU(0) factorization with the original ordering, MC-ILU(0) usually drops more fill-in elements. Therefore, the resulting preconditioner in general will have lower accuracy and the number of the iterations required will be higher. For both methods, the degree of the parallelism is n/p. When p is large, a possible remedy is to "sparsify" the graph, which can be carried out as follows:

1. Find a coloring with p colors from the greedy algorithm, where $p \leq \Delta\,(\mathcal{G}) + 1$,

2. Find all the nodes whose degrees are larger than or equal to $p-1$ and remove the edges with small weights (i.e., zero out the matrix entries of small magnitudes),

Usually, repeating these steps a few times can quickly reduce the number of colors. Then, we obtain an *approximate multi-coloring* of the original graph, where two vertices of the same color are either not connected or connected by a "weak" edge. Correspondingly, in the adjacency matrix (3.3), D_i will not be exactly a diagonal matrix anymore but will have some off-diagonal entries of small magnitudes. By dropping these off-diagonal entries, the same procedure of MC-SSOR/ILU(0) can be performed.

3.4.4 Least-squares polynomial preconditioners

A polynomial preconditioner M is defined by

$$M^{-1} = s(A),$$

where s is some polynomial, typically of a low degree. Note that there is no need to form the preconditioning matrix $s(A)$ or the preconditioned matrix $As(A)$ explicitly since for an arbitrary vector v, the preconditioning operation $u = s(A)v$ can be performed by a sequence of SpMVs. With a high-performance SpMV kernel, polynomial preconditioners can be efficient on SIMD parallel machines. Popular choices for polynomial preconditioners include simple Neumann polynomials, Chebyshev polynomials and least-squares polynomials. Here we consider the least-squares polynomials for symmetric matrices. Discussions on nonsymmetric cases can be found in [12, §12.3.4].

Let $\mathbb{P}_k$ be the space of polynomials of degree not exceeding k. Consider the inner product on $\mathbb{P}_k$:

$$\langle p, q \rangle = \int_{\alpha}^{\beta} p(\lambda)q(\lambda)\omega(\lambda)\, d\lambda, \tag{3.4}$$

where $p, q \in \mathbb{P}_k$ and ω is a non-negative weight function on interval $[\alpha, \beta]$. Then, the corresponding norm $\langle p, p\rangle^{1/2}$ is called ω-norm, denoted by $\|p\|_{\omega}$. Our aim is to find the polynomial $s_{k-1}^{\star}(\lambda)$ that minimizes

$$\|1 - \lambda s_{k-1}\|_{\omega} \tag{3.5}$$

over all the polynomials of degree not exceeding $k-1$. $s_{k-1}^{\star}$ is called the *least-squares polynomial*, and $r_k = 1 - \lambda s_{k-1}^{\star}$ is termed the *least-squares residual polynomial.*

An algorithm for computing the least-squares polynomial

As known from the orthogonal polynomial theory, there is an orthonormal sequence $\{p_n\}$ of polynomials with respect to the inner product (3.4), where $\{p_n\}$ satisfies the following 3-term recurrence:

$$\beta_{n+1}p_{n+1}(\lambda) = (\lambda - \alpha_n)p_n(\lambda) - \beta_n p_{n-1}(\lambda) \tag{3.6}$$

with

$$\alpha_n = \langle \lambda p_n, p_n \rangle\,, \quad \beta_{n+1} = \|(\lambda - \alpha_n)p_n(\lambda) - \beta_n p_{n-1}(\lambda)\|_\omega. \tag{3.7}$$

From (3.6) and (3.7), starting with a certain polynomial $\tilde{p}_0(\lambda)$, the sequence of orthonormal polynomials $\{p_n(\lambda)\}$ can be computed by the following algorithm, which is known as the Stieltjes procedure in the literature [114, 115].

Algorithm 5 Stieltjes

```
1: $\beta_0 = \|\tilde{p}_0\|_w$, $p_0 = \tilde{p}_0/\beta_0$, $p_{-1} \equiv 0$
2: for $n = 0, 1, \ldots$ do
3:     $\alpha_n = \langle \lambda p_n, p_n \rangle$
4:     $\tilde{p}_{n+1} = (\lambda - \alpha_n)p_n - \beta_n p_{n-1}$
5:     $\beta_{n+1} = \|\tilde{p}_{n+1}\|_\omega = \langle \tilde{p}_{n+1}, \tilde{p}_{n+1} \rangle^{1/2}$
6:     $p_{n+1} = \tilde{p}_{n+1}/\beta_{n+1}$
7: end for
```

Let $\mathbb{P}_{k,1}$ denote the space of all the polynomials of the form $\lambda s(\lambda)$ with $s(\lambda) \in \mathbb{P}_{k-1}$. For the least-squares problem (3.5), it is clear that the residual polynomial r_k must be orthogonal to the space $\mathbb{P}_{k,1}$. In other words, $s^\star_{k-1}$ must satisfy

$$\left\langle 1 - \lambda s^\star_{k-1}, t \right\rangle = 0, \quad \forall t \in \mathbb{P}_{k,1}. \tag{3.8}$$

Consider a sequence of polynomials $\{p_n\}_{0 \le n \le k-1}$ with $p_n \in \mathbb{P}_{k-1}$, such that $\{\lambda p_n\}$ is orthonormal with respect to the inner product (3.4). Denote $q_n \equiv \lambda p_n$ and clearly we have $q_n \in \mathbb{P}_{k,1}$. The orthonormal sequence $\{q_n\}$ can be computed by the Stieltjes procedure with the starting polynomial $\tilde{q}_0 = \lambda$. Therefore, if we expand $\lambda s^\star_{k-1}$ in the orthonormal basis $\{q_n\}$

$$\lambda s^\star_{k-1} = \sum_{n=0}^{k-1} \gamma_n q_n = \sum_{n=0}^{k-1} \gamma_n \lambda p_n,$$

from (3.8), it follows that

$$\gamma_n = \langle 1, \lambda p_n \rangle \,. \tag{3.9}$$

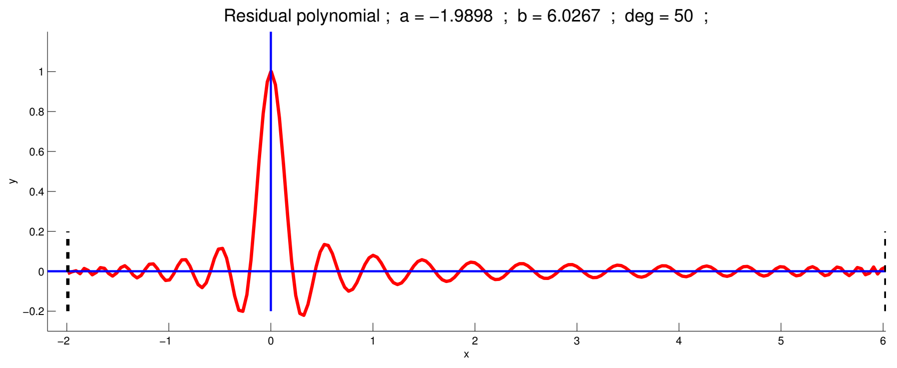
Then, the desired least-squares polynomial can be obtained by

$$s_{k-1}^{\star} = \sum_{n=0}^{k-1} \gamma_n p_n, \tag{3.10}$$

where $\{p_n\}$ satisfies the 3-term recurrence (3.6) and $p_0 = 1/\beta_0$.

An example of the least-squares residual polynomial $r_k = 1 - \lambda s_{k-1}^{\star}$ of degree 50 on interval $(-1.99, 6.03)$ is shown in Figure 3.1.

Figure 3.1: An example of the least-squares residual polynomial of degree 50.



An important issue to consider is how to compute the inner products which appear in the Stieltjes procedure and (3.9). It has been realized that when the polynomials are expressed in a basis of Chebyshev polynomials with some suitable weight function, these inner products can be computed efficiently without resorting to numerical integration [116, 117]. For completeness, we give the details of these computations in the Stieltjes procedure and (3.9).

Let $c = (\alpha+\beta)/2$ and $d = (\beta-\alpha)/2$ be the center and the half width of the interval (α, β). For the inner product (3.4), consider the weight function

$$\omega(\lambda) = \frac{2}{\pi}[d^2 - (\lambda - c)^2]^{1/2}, \quad \lambda \in (\alpha, \beta). \tag{3.11}$$

It follows that the polynomials

$$P_n(\lambda) \equiv T_n[(\lambda - c)/d], \quad \lambda \in (\alpha, \beta) \tag{3.12}$$

where T_n is the Chebyshev polynomial of the first kind of degree n, are orthogonal in (α, β) with respect to the weight function (3.11), since we have

$$\langle P_i, P_j \rangle = \begin{cases} 2 & i = j = 0 \\ 1 & i = j \neq 0 \\ 0 & i \neq j \end{cases} .$$

The following recurrence relation of P_n will be useful,

$$\lambda P_n = \frac{d}{2} P_{n+1} + c P_n + \frac{d}{2} P_{n-1}, \quad n \geq 1 \tag{3.13}$$
$$\lambda P_0 = d P_1 + c P_0$$

Let polynomial $p_n(\lambda)$ of degree not exceeding n be expressed in the basis (3.12) as

$$p_n(\lambda) = \sum_{i=0}^{n} \mu_i P_i(\lambda).$$

The following proposition gives the formulae for computing the coefficients α_n, β_{n+1} and the polynomial $p_{n+1}(\lambda)$ in the Stieltjes procedure.

Proposition 3.4.1 *Let $p(\lambda)$ be any polynomial having the following expansion*

$$p(\lambda) = \sum_{i=0}^{n} \mu_i P_i(\lambda),$$

and let

$$\sigma = 2\mu_0^2 + \sum_{i=1}^{n} \mu_i^2, \quad \tau = 2\mu_0\mu_1 + \sum_{i=1}^{n-1} \mu_i \mu_{i+1}$$

Then,

$$\langle p, p \rangle = \sigma \tag{3.14}$$
$$\langle \lambda p, p \rangle = d\tau + c\sigma \tag{3.15}$$
$$\lambda p(\lambda) = \sum_{i=0}^{n+1} \eta_i P_i(\lambda) \tag{3.16}$$

with

$$\eta_0 = \frac{d}{2}\mu_1 + c\mu_0$$
$$\eta_1 = \frac{d}{2}\mu_2 + d\mu_0 + c\mu_1$$
$$\eta_i = \frac{d}{2}(\mu_{i+1} + \mu_{i-1}) + c\mu_i, \quad i \geq 2. \tag{3.17}$$

Proof. Consider λp first. From (3.13), we have

$$\begin{aligned}\lambda p &= \sum_{i=0}^{n} \mu_i \lambda P_i = \mu_0 \lambda P_0 + \sum_{i=1}^{n} \mu_i \left(\frac{d}{2} P_{i+1} + c P_i + \frac{d}{2} P_{i-1} \right) \\ &= (\frac{d}{2}\mu_1 + c\mu_0) P_0 + (\frac{d}{2}\mu_2 + d\mu_0 + c\mu_1) P_1 + \sum_{i=2}^{n+1} \left[\frac{d}{2}(\mu_{i+1} + \mu_{i-1}) + c\mu_i \right] P_i.\end{aligned}$$

Then, for (3.15) we have

$$\begin{aligned}\langle \lambda p, p \rangle &= \left\langle \sum_{i=0}^{n+1} \left[\frac{d}{2}(\mu_{i+1} + \mu_{i-1}) + c\mu_i \right] P_i + \frac{d}{2}\mu_0 P_1, \sum_{i=0}^{n} \mu_i P_i \right\rangle \\ &= \left(\frac{d}{2}\mu_1 + c\mu_0 \right) \mu_0 \langle P_0, P_0 \rangle + \sum_{i=1}^{n} \left[\frac{d}{2}(\mu_{i+1} + \mu_{i-1}) + c\mu_i \right] \mu_i \langle P_i, P_i \rangle \\ &\quad + \frac{d}{2}\mu_0\mu_1 \langle P_1, P_1 \rangle \\ &= \left(d\mu_0\mu_1 + 2c\mu_0^2 \right) + \left(\frac{d}{2}\mu_0\mu_1 + d\sum_{i=1}^{n-1} \mu_i\mu_{i+1} + c\sum_{i=1}^{n} \mu_i^2 \right) + \frac{d}{2}\mu_0\mu_1 \\ &= d\left(2\mu_0\mu_1 + \sum_{i=1}^{n-1} \mu_i\mu_{i+1} \right) + c\left(2\mu_0^2 + \sum_{i=1}^{n} \mu_i^2 \right) = d\tau + c\sigma.\end{aligned}$$

Finally, for (3.14), we can write

$$\langle p, p \rangle = \left\langle \sum_{i=0}^{n} \mu_i P_i, \sum_{i=0}^{n} \mu_i P_i \right\rangle = 2\mu_0^2 + \sum_{i=1}^{n} \mu_i^2 = \sigma,$$

which completes the proof. □

In summary, in the Stieltjes procedure α_n is computed by (3.15), β_{n+1} is computed by (3.14) where the λp_{n+1} is obtained by (3.16)-(3.17). According to (3.9), we have the coefficients $\gamma_n = \langle 1, \lambda p_n \rangle$ which can be computed as follows

$$\gamma_n = \langle 1, \lambda p_n \rangle = \left\langle P_0, \sum_{i=0}^{n} \mu_i P_i(\lambda) \right\rangle = 2\mu_0.$$

Another issue which has not been addressed is the choice of the interval (α, β) where the inner product (3.4) is defined. Suppose that A is symmetric. The spectrum of A must be included in this interval. A simple approach is to use the Gershgorin circle theorem to find (α, β), which is, however, usually overestimated. Better estimation of the spectrum (not bounds) can be obtained inexpensively by using a few steps of the Lanczos method [118]. In order to have bounds of the spectrum, the safeguard terms used in [119] are included, see [120, §13.2] for the definitions of these terms.

Application of the least-squares polynomial as a preconditioner

Let us consider solving the linear system $Ax = b$ where A is symmetric with iterative methods and using the least-squares polynomial

$$s^{\star}_{k-1}(A) = \sum_{n=0}^{k-1} \gamma_n p_n(A)$$

as the preconditioner, from which it follows that an approximate solution of x can be obtained by

$$x = x_0 + A^{-1} r_0 \approx x_0 + s^{\star}_{k-1}(A) r_0 = x_0 + \sum_{n=0}^{k-1} \gamma_n p_n(A) r_0, \tag{3.18}$$

where x_0 is an initial guess and $r_0 = b - Ax_0$ is the corresponding residual. Let $v_n = p_n(A) r_0$ and we can rewrite (3.18) as

$$x \approx x_0 + \sum_{j=0}^{k-1} \gamma_n v_n. \tag{3.19}$$

From the 3-term recurrence (3.6) for the polynomials $\{p_n\}$, we have the following recurrence for the vectors $\{v_n\}$

$$\beta_{n+1} v_{n+1} = A v_n - \alpha_n v_n - \beta_n v_{n-1},$$

with the coefficients $\{\alpha_n\}$, $\{\beta_n\}$ and $\{\gamma_n\}$ obtained from (3.7) and (3.9). Therefore, the preconditioning operation (3.19) can be performed as follows.

```
1: $v_0 = r_0/\beta_0$, $v_{-1} \equiv 0$, $x_1 = x_0 + \gamma_0 v_0$
2: for $n = 1, \ldots, k-1$ do
3:     $v_n = (A v_{n-1} - \alpha_{n-1} v_{n-1} - \beta_{n-1} v_{n-2}) / \beta_n$
4:     $x_{n+1} = x_n + \gamma_n v_n$
5: end for
```

3.4.5 Other types of GPU-accelerated preconditioners

The fact that ILU preconditioners are not suitable for highly parallel machines motivated the development of a class of approximate inverse preconditioners in the 1990s as alternatives, that do not require sparse triangular solves. These methods are based on finding an approximate inverse of the original matrix or the approximate LU factors of the inverse, that are also kept sparse, see, e.g., [15–18, 121, 122] among others. Therefore, preconditioning operations of these preconditioners are SpMV, which makes them appropriate for the GPU computing. Compared with ILU preconditioners, the cost of computing these preconditioners and the number of iterations required for convergence are usually high. A well-known weakness of these preconditioners is that they do not work very well for highly indefinite matrices in general. GPU-accelerated approximate-inverse-type preconditioners were studied in [123–126]. Another important type of preconditioning methods, which has been accelerated by GPUs [127, 128], is the algebraic multigrid (AMG). The optimality of these methods in certain types of problems and the fact that fine-grained parallelism is much more prevalent in the AMG algorithm make these methods have an outstanding scalability at large processor core counts. However, the success of AMG is still somewhat restricted to certain types of problems.

3.5 Experiment results

The experiments were conducted on Cascade, a GPU cluster at the Minnesota Supercomputing Institute (MSI). Each node consists of dual Intel Xeon X5675 processors (12 MB Cache, 3.06 GHz, 6-core) and 96 GiB of main memory; and 4 NVIDIA TESLA M2070 GPU (448 cores, 1.15 GHz, 3GB memory). In the following tests, we compared the performance of the sparse kernels and the preconditioned Krylov subspace methods on a single CPU and a single GPU. The CUDA programs were compiled by the NVIDIA CUDA compiler (`nvcc`) with flag `-arch sm_20` and the CPU programs were compiled by g++ using the -O3 optimization level. The CUDA Toolkit 4.1 and Intel Math Kernel Library (MKL) 10.2 were used for programming. MKL is threaded using OpenMP [27] for the use of multicores. For the CUDA kernel configuration, we used one-dimensional thread block of size 256, and the maximum number of threads was hard coded to be the maximum number of resident threads on this device, which is 14×1536.

3.5.1 SpMV kernels

We first compare the performance of the CPU and the GPU SpMV kernels for double precision floating point arithmetic. On the CPU side, the CSR SpMV kernel is obtained from routine `mkl_dcsrgemv` from MKL. The GPU SpMV kernels tested include the two CSR variants, CSR-scalar and CSR-vector, the JAD kernel, the DIA kernel and the HYB kernel from the NVIDIA CUSPARSE library [106]. The test matrices were selected from the University of Florida sparse matrix collection [26]. The size (N), the number of the non-zeros (NNZ) and the average number of non-zeros per row (RNZ) of each matrix are tabulated in Table 3.1.

Table 3.1: Name, order (N), number of nonzeros (NNZ) and average number of nonzeros per row (RNZ) of the test matrices.

Matrix	N	NNZ	RNZ
sherman3	5,005	20,033	4.0
memplus	17,758	126,150	7.1
msc23052	23,052	1,154,814	50.1
bcsstk36	23,052	1,143,140	49.6
rma10	46,835	2,374,001	50.7
dubcova2	65,025	1,030,225	15.8
cfd1	70,656	1,828,364	25.9
poisson3Db	85,623	2,374,949	27.7
cfd2	123,440	3,087,898	25.0
boneS01	127,224	6,715,152	52.8
majorbasis	160,000	1,750,416	10.9
pwtk	217,918	11,634,424	53.4
af_shell8	504,855	17,588,875	34.8
parafem	525,825	3,674,625	7.0
ecology2	999,999	4,995,991	5.0
Lap7pt	1,000,000	6,940,000	6.9
spe10	1,094,421	7,598,799	6.9
thermal2	1,228,045	8,580,313	7.0
atmosmodd	1,270,432	8,814,880	6.9
atmosmodl	1,489,752	10,319,760	6.9

The performance of the CPU and the GPU kernels is reported in Table 3.2 measured in GFLOPS. According to the experimental results, the CSR-vector kernel (CSRv) performs, in general, better than the CSR-scalar kernel (CSRs), especially for the matrices which have large numbers of nonzeros per row (cf. e.g., `msc23052`, `boneS01`). The JAD kernel yields higher performance than the two CSR kernels, and for most cases, the JAD kernel can be competitive or can even outperform the HYB kernel. Compared with the CPU kernel, the speedup of the GPU kernels can reach a factor of 8. For diagonally structured matrices, the DIA kernel is the most efficient.

Table 3.2: Performance of the CPU/GPU SpMV kernels.

Matrix	MKL	CSRs	CSRv	JAD	HYB	DIA
sherman3	3.47	3.67	1.76	4.29	1.03	5.28
memplus	2.55	0.95	2.89	0.43	4.19	-
msc23052	8.02	1.63	9.32	8.05	9.73	-
bcsstk36	9.82	1.63	9.36	8.03	8.67	-
rma10	3.80	1.72	10.19	12.61	8.48	-
dubcova2	4.88	1.97	5.97	11.36	8.61	-
cfd1	4.70	1.73	8.61	12.91	12.33	-
poisson3Db	1.97	1.28	5.70	6.57	5.88	-
cfd2	2.88	1.76	8.52	11.95	12.18	-
boneS01	3.16	1.82	10.77	8.69	10.58	-
majorbasis	2.92	2.67	4.81	11.70	11.54	13.06
pwtk	3.08	1.69	10.33	13.80	13.24	-
af_shell8	3.13	1.54	10.34	14.56	14.27	-
parafem	2.30	3.15	4.42	8.01	7.78	-
ecology2	2.06	8.37	6.72	12.04	12.11	13.33
Lap7pt	2.59	3.92	4.66	11.58	12.44	18.70
spe10	2.49	4.06	4.63	8.66	10.96	-
thermal2	1.76	3.05	3.55	5.47	7.33	-
atmosmodd	2.09	3.78	4.69	10.89	10.97	16.03
atmosmodl	2.25	3.74	4.71	10.77	10.61	14.65

3.5.2 SpTrsv kernels

In Table 3.3, we report the performance of the sparse triangular solution kernels (measured in MFLOPS) on the CPU and the GPU. The triangular matrices are the ILU(0) factors of the test matrices. The CPU kernel is routine `mkl_dcsrtrsv` from MKL.

The GPU kernels include two simple row and column kernels where only the dot products or the AXPY operations are parallelized, and a row version with level-scheduling. As shown, the performance of the two simple kernels is extremely low, only up to 30 MFLOPS whereas hundreds of MFLOPS were reached by the CPU kernel. The performance of the kernel with level-scheduling is significantly enhanced when the number of levels (indicated by the number in parenthesis in the table) is moderate. In a few cases, namely `poisson3Db`, `parafem`, `Lap7pt`, `thermal2`, `atmosmodd` and `atmosmodd`, the performance can exceed that of the CPU kernel. MD-type orderings can significantly reduce the number of levels and thereafter raise the performance. The last two columns of the table show the performance with the MMD ordering.

Table 3.3: Performance of the CPU/GPU SpTrsv kernels.

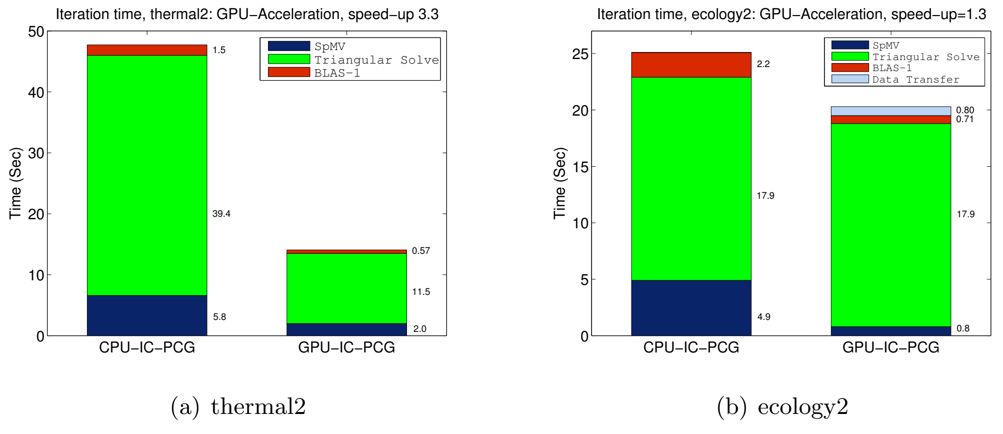
Matrix	MKL	Row	Col	Lev		MMD-Lev	
sherman3	439.2	3.5	3.6	79.3	(50)	338.7	(9)
memplus	699.1	5.4	5.9	77.7	(174)	84.2	(147)
msc23052	953.9	21.3	25.2	649.8	(192)	704.8	(186)
bcsstk36	1001.9	19.9	23.1	41.6	(4,457)	461.3	(274)
rma10	838.3	21.6	29.2	47.0	(7,785)	324.6	(765)
dubcova2	755.7	10.1	14.2	103.0	(1634)	1519.3	(44)
cfd1	819.4	15.4	18.5	80.0	(4,340)	365.2	(637)
poisson3Db	455.9	13.1	16.6	945.1	(136)	992.5	(191)
cfd2	841.2	15.2	21.1	128.4	(4,357)	703.8	(494)
boneS01	887.0	21.9	28.2	997.8	(813)	1207.2	(480)
majorbasis	718.0	7.78	6.8	463.2	(600)	573.6	(436)
pwtk	946.5	20.4	26.1	14.6	(142,168)	1615.7	(559)
af_shell8	919.6	17.1	21.8	635.4	(3,725)	1699.6	(550)
parafem	562.1	5.2	6.1	1333.2	(7)	1166.9	(20)
ecology2	450.8	3.6	5.3	354.8	(1,999)	1086.3	(5)
Lap7pt	556.9	4.9	3.9	659.4	(298)	1407.3	(6)
spe10	574.0	4.9	3.8	484.6	(622)	675.1	(349)
thermal2	361.1	5.1	6.7	587.1	(1,239)	939.3	(32)
atmosmodd	564.2	4.9	5.5	668.0	(352)	1403.0	(6)
atmosmodl	548.8	4.9	5.8	661.8	(432)	1425.9	(6)

In the following experiments, we will report the performance of Krylov subspace methods with the parallel preconditioning techniques discussed in Section 3.4.

3.5.3 ILU preconditioners

First, we present the performance of ILU preconditioners. In Figure 3.2, we show the time breakdown of the CG method with the incomplete Cholesky (IC) preconditioner for two SPD cases, namely `thermal2` and `ecology2`, with and without GPU-acceleration. As shown, for both cases the running time of the SpMVs and the BLAS-1 operations was reduced. For `thermal2`, the SpTrsv was performed on the GPU, which was also accelerated. So, for this case the overall speedup of the iteration time is a factor of 3.3. On the other hand, for `ecology2`, the SpTrsv resided on the CPU side due to the poor performance of the GPU kernel. As a result, extra cost of data transferring between the host and the device was required. The overall speedup for this case is only a factor of 1.3, which reflects Amdahl's law, which describes the performance improvement when only a fraction of computations is parallelized.

Figure 3.2: Time breakdown of the IC-CG method.



(a) thermal2 (b) ecology2

The performance of the IC-CG method for the SPD cases is reported in Table 3.4, where the time for computing the preconditioner (p-t), the fill-ratios (fill), i.e., the ratio of the number of nonzeros in the preconditioner to the number of nonzeros in the original matrix, the numbers of iterations (its) required to reduce residual norm by 6 orders of magnitude, the iteration time (i-t) and the speedup of the overall time compared with the CPU counterpart are tabulated. The second column indicates whether the SpTrsv was performed on the CPU or the GPU. All timings are in seconds.

Table 3.4: Performance of the IC-CG method.

Matrix	sptrsv	p-t	nz	its	i-t	speedup
msc23052	CPU	0.35	2.1	277	1.38	1.2
bcsstk36	CPU	0.44	2.7	130	0.79	1.2
dubcova2	CPU	0.34	2.6	29	0.19	1.2
cfd1	CPU	0.44	2.2	245	2.12	1.3
cfd2	CPU	0.75	2.4	172	2.69	1.4
boneS01	CPU	1.93	2.2	252	7.89	1.4
af_shell8	CPU	3.73	2.4	105	9.03	1.5
parafem	GPU	1.35	2.7	188	5.28	1.6
ecology2	CPU	0.48	2.3	561	20.34	1.3
thermal2	GPU	3.80	2.3	329	14.19	2.9

For the symmetric indefinite and the nonsymmetric cases, GMRES with a restart dimension of 40, denoted by GMRES(40), was used combined with the ILUT preconditioner. Experiments were carried out to compare the performance of the ILUT-GMRES method with and without GPU acceleration. The results are shown in Table 3.5.

Table 3.5: Performance of the ILUT-GMRES method.

Matrix	sptrsv	p-t	nz	its	i-t	speedup
sherman3	CPU	0.01	2.1	62	0.19	0.3
memplus	CPU	0.02	1.1	65	0.22	1.3
poisson3Db	GPU	1.31	2.3	44	0.98	1.5
majorbasis	CPU	0.22	1.9	4	0.04	1.3
spe10	CPU	0.69	1.8	44	2.16	1.6
atmosmodd	GPU	0.95	1.6	119	5.05	2.5
atmosmodl	GPU	1.15	1.7	55	2.64	3.5

3.5.4 Block Jacobi preconditioners

A graph partitioner from the METIS package [76] was used to partition the adjacency graph. ILU factorizations were used on the diagonal blocks. The preconditioning operation is the block SpTrsv, which was always performed on the GPU. In Table 3.6, we report the results of the block-Jacobi-GMRES method with different numbers of blocks (nb) for case `SPE10`, and the performance (in MFLOPS) of the block SpTrsv (bsptrsv). From the results, we can see that the performance of the block SpTrsv increases with the number of blocks, since more parallelism is available in this computation. However, the number of iterations required also increases since the preconditioner becomes weaker. For this case, the best number of blocks in terms of the iteration time is 32.

Table 3.6: SPE10: Performance of the block-Jacobi-GMRES method.

nb	bsptrsv	p-t	its	i-t
4	334	0.86	81	7.71
8	546	0.86	102	6.34
16	623	0.84	121	6.33
32	727	0.81	119	5.88
64	750	0.80	122	6.18
128	755	0.79	155	7.97

3.5.5 Multi-color SSOR/ILU(0) preconditioners

Experimental results of MC-SSOR/ILU(0) are shown in Table 3.7, where ‘nc’ stands for the number of colors and k is the optimal number of the SSOR steps in terms of the overall iteration time. Compared with the standard ILU(0), with MC-ILU(0), GMRES required more iterations for the convergence. However, an overall performance gain was still achieved in terms of the reduced iteration time.

For two cases in Table 3.7, `memplus` and `spe10`, a large number of colors were found by the coloring algorithm, which drags down the performance. The heuristic sparsification strategy discussed at the end of Section 3.4.3 was applied, which turned out to be able to reduce the numbers of colors effectively so that the efficiency of the MC-SSOR/ILU(0) preconditioning can be significantly improved. The performance and the numbers of colors before and after the sparsification are shown in Table 3.8.

Table 3.7: Performance of the MC-SSOR/ILU(0)-GMRES method.

Matrix	MC-SSOR					MC-ILU(0)			ILU(0)		
	p-t	its	i-t	nc	k	p-t	its	i-t	p-t	its	i-t
sherman3	0.00	40	0.12	4	15	0.00	312	0.99	0.00	92	0.31
memplus	0.00	60	0.94	97	5	0.02	146	0.77	0.02	138	0.87
poisson3Db	0.01	39	0.66	17	5	0.26	107	0.60	0.39	106	0.97
majorbasis	0.00	7	0.04	6	2	0.05	12	0.04	0.07	9	0.10
spe10	0.02	93	11.03	340	5	0.19	274	7.50	0.31	157	7.03
atmosmodd	0.02	69	3.89	2	5	0.17	210	4.38	0.19	137	5.52
atmosmodl	0.02	33	2.01	2	5	0.20	105	2.46	0.31	63	3.04

Table 3.8: Performance of MC-SSOR/ILU(0) without and with sparsification.

Matrix	without sparsification					with sparsification				
	nc	SSOR(5)		ILU(0)		nc	SSOR(5)		ILU(0)	
		its	i-t	its	i-t		its	i-t	its	i-t
memplus	97	60	0.94	146	0.77	3	47	0.19	202	0.58
spe10	340	93	11.03	274	7.50	7	93	5.37	247	5.09

3.5.6 Least-squares polynomial preconditioners

In the next set of experiments, we tested the performance of the least-squares polynomial preconditioner for the symmetric cases. The time for building the preconditioner is typically small, including the time for computing the polynomial and the time for performing a few steps of the Lanczos algorithm for the eigenvalue bounds. The results are shown in Table 3.9, where ‘deg’ is the degree of the polynomial used. We compared the performance of this preconditioner with that of the IC preconditioner presented in Table 3.4 by showing the speedup in terms of the total solution time, which is shown in the last column of Table 3.9. A speedup by a factor up to 4.1 was achieved.

Finally, at the end of this section, we present the performance of the preconditioned CG method for solving the Poisson equation on a 2-D 1000×1000 grid and a 3-D $100 \times 100 \times 100$ grid. For each preconditioner, the total solution time is reported in Table 3.10. The first row of this table gives the performance of the IC-CG method on the CPU and the performance of its GPU-accelerated counterpart. From rows 2 to 5,

Table 3.9: Performance of CG with the least-squares polynomial preconditioner.

Matrix	p-t	its	i-t	deg	speedup
msc23052	0.02	98	1.39	40	1.2
bcsstk36	0.02	66	1.08	60	1.1
dubcova2	0.07	17	0.06	10	4.1
cfd1	0.05	58	0.77	30	3.1
cfd2	0.09	20	1.25	80	2.6
boneS01	0.11	235	4.70	10	2.1
af_shell8	0.34	68	4.78	20	2.5
parafem	0.31	41	2.19	30	2.7
ecology2	0.57	323	16.40	20	1.3
thermal2	0.77	147	15.41	20	1.1

we show the results of the four GPU-appropriate preconditioners discussed. For the 2D case, the least-squares polynomial preconditioner of degree 20 yielded the best number of iterations and also the best total solution time, while for the 3D case this method of degree 15 converged with the fewest iterations but the winner in terms of the running time was the multi-coloring IC(0) preconditioner.

Table 3.10: Performance comparison of all the preconditioning methods.

Precon	Lap2d			Lap3d		
	its	CPU	GPU	its	CPU	GPU
IC	120	9.12	5.85	47	3.40	2.12
BJ(8)	326	17.01	10.33	96	5.12	3.39
MC-SSOR(3)	407	26.69	9.01	44	5.86	1.81
MC-IC(0)	738	20.31	5.56	101	5.11	0.94
LS-Poly	91	11.47	3.23	16	3.57	1.08

3.6 Summary and discussion

In this chapter, we have presented approaches for developing sparse matrix-vector product kernels and sparse triangular solution kernels on the current many-core platforms, and use these kernels to construct efficient iterative methods for solving sparse linear systems in conjunction with different preconditioning techniques. Among these preconditioners, block Jacobi is the simplest but it usually requires many iterations to converge. For the matrices which can be colored with a few colors, multi-color SSOR/ILU(0) can yield good performance due to the inherent high degree of parallelism. The least-squares polynomial preconditioner was also found successful on GPUs for solving sparse symmetric linear systems. A software package, CUDA_ITSOL, consisting of these preconditioning techniques as an iterative solver package for GPUs was made publicly available, see `http://www-users.cs.umn.edu/~saad/software/`.

Based on the experimental results reported here and elsewhere, we observed that, when used as general purpose many-core processors, the current GPUs will provide a much lower performance advantage for irregular (sparse) computations than they can for more regular (dense) computations. However, when used carefully, GPUs can still be beneficial as co-processors to CPUs to speedup some portions of complex computations. We highlighted a few alternative preconditioning methods to the standard ones. A further step is to adapt the current methods to a multi-CPU/GPU environment to solve larger scale problems.

Chapter 4

Divide and conquer low-rank preconditioners for symmetric matrices

Krylov subspace methods preconditioned with a form of incomplete LU (ILU) factorization can be quite effective for solving large sparse linear systems but there are situations where these preconditioners will not perform well. For instance, as discussed in Section 2.3.4, when the matrix is highly ill-conditioned or indefinite, ILU-type preconditioners are unlikely to work, either because the construction of the factors will not complete or because the resulting factors are unstable and in some cases quite dense. Another situation is related to the architecture under which the system is being solved. Building and using an ILU factorization is a highly scalar process. Blocking, which is a highly effective strategy utilized by sparse direct solvers to boost performance, is rarely exploited in the realm of iterative solution techniques. As seen in the previous chapter, ILU preconditioners can yield exceedingly poor performance on massively parallel processors like GPUs. In this chapter, we will present a preconditioning method based on low-rank approximations primarily as a means to bypass the issues just mentioned.

4.1 Introduction

A line of work that emerged in recent years as a means to compute preconditioners, is that of *rank-structured matrices*. The starting point is the work by W. Hackbusch and co-workers who introduced the notion of the hierarchical matrices ($\mathcal{H}$-matrices) in the 1990s [129–133]. These were based on some interesting rank-structure observed on matrices arising from the use of the fast multipole methods or the inverses of some partial differential operators. A similar rank-structure was also exploited by others in the so-called Hierarchically Semi-Separable (HSS) matrix format which represents certain off-diagonal blocks by low-rank matrices [134–137].

In this chapter, we present a divide-and-conquer approach to compute preconditioners for *symmetric* matrices, which is referred to as the Multilevel Low-Rank (MLR) preconditioner [138]. In a nutshell, the idea is based on the observation that if a domain is divided into two subdomains, then one can get the inverse of the matrix associated with the whole domain by the inverses of the matrices associated with both subdomains plus a low-rank correction. The divide step can be performed via standard domain decomposition (DD) methods. In the conquer step, the inverse of the original matrix can be expressed by the Sherman-Morrison-Woodbury (SMW) formula, in which we apply the low-rank approximations to obtain the preconditioning matrix.

Extensions to the nonsymmetric case are possible and will be explored in our future work. It should also be emphasized that even though these preconditioners are developed with highly parallel platforms in mind, especially the ones equipped with GPUs, we have not implemented and tested these methods in such environments. Nevertheless, the potentials can be seen from the experiment results on multi-core CPUs.

4.2 A divide-and-conquer algorithm

We begin with a model problem where the coefficient matrix A is a symmetric matrix derived from a 5-point stencil discretization of the Laplacian operator on domain Ω which is a regular $n_x \times n_y$ grid with Dirichlet boundary conditions. In this case, $A \in$

$\mathbb{R}^{n\times n}$ has the following form:

$$A = \left(\begin{array}{cccc|cccc} A_1 & D_2 & & & & & & \\ D_2 & A_2 & D_3 & & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & D_\alpha & A_\alpha & D_{\alpha+1} & & & \\ \hline & & & D_{\alpha+1} & A_{\alpha+1} & \ddots & & \\ & & & & \ddots & \ddots & D_{n_y} & \\ & & & & & D_{n_y} & A_{n_y} & \end{array}\right), \tag{4.1}$$

where A_j, $j = 1, \ldots, n_y$ is a tridiagonal matrix of dimension n_x and matrix $D_j \in \mathbb{R}^{n_x \times n_x}$ is diagonal for $j = 2, \ldots, n_y$, so that we have $n = n_x n_y$. In (4.1), the matrix A has the splitting of the following form

$$A \equiv \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & \\ & A_{22} \end{pmatrix} + \begin{pmatrix} & A_{12} \\ A_{21} & \end{pmatrix}, \tag{4.2}$$

with $A_{11} \in \mathbb{R}^{m\times m}$ and $A_{22} \in \mathbb{R}^{(n-m)\times(n-m)}$, where we assume that $0 < m < n$ and m is a multiple of n_x, i.e., $m = \alpha n_x$ with $\alpha \in \{1, 2, \ldots, n_y - 1\}$. The matrix splitting in (4.2) corresponds to the mesh bipartitioning as shown in Figure 4.1, where A_{11} and A_{22} are associated with subdomains Ω_1 and Ω_2, while A_{12} and A_{21} represent the couplings between Ω_1 and Ω_2. Then, an interesting observation is that $A_{12} = A_{21}^T \in \mathbb{R}^{m\times(n-m)}$ has rank n_x because

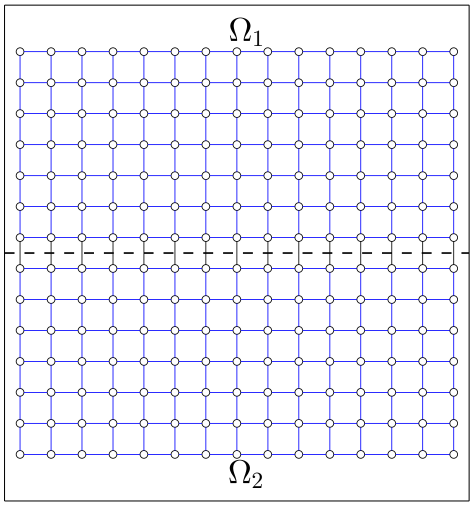
$$A_{12} = A_{21}^T = \begin{pmatrix} 0 & 0 \\ D_{\alpha+1} & 0 \end{pmatrix} = -E_1 E_2^T, \tag{4.3}$$

with

$$E_1^T = \begin{pmatrix} 0 & D_{E_1} \end{pmatrix} \in \mathbb{R}^{n_x\times m} \quad \text{and} \quad E_2^T = \begin{pmatrix} D_{E_2} & 0 \end{pmatrix} \in \mathbb{R}^{n_x\times(n-m)},$$

where D_{E_1} and D_{E_2} are diagonal matrices of dimension n_x such that $D_{E_1} D_{E_2} = -D_{\alpha+1}$. For example, in the common case when $D_{\alpha+1} = -I$, we can take $D_{E_1} = D_{E_2} = I_{n_x}$. From the viewpoint of the underlying mesh, this rank n_x can be interpreted as the size of the *edge-separator*, i.e., the number of the edges (the edges cut by the dashed line in Figure 4.1) connecting Ω_1 and Ω_2.

Figure 4.1: 2D mesh partitioning $\Omega = (\Omega_1, \Omega_2)$. Edges cut by the dashed line are the edge-separator.



Therefore, by substituting (4.3) in (4.2), we can rewrite the splitting of A as

$$A = \begin{pmatrix} A_{11} + E_1E_1^T & \\ & A_{22} + E_2E_2^T \end{pmatrix} - \begin{pmatrix} E_1E_1^T & E_1E_2^T \\ E_2E_1^T & E_2E_2^T \end{pmatrix}.$$

Thus, we have

$$A = B - EE^T, \quad B = \begin{pmatrix} B_1 & \\ & B_2 \end{pmatrix} \in \mathbb{R}^{n\times n}, \quad E = \begin{pmatrix} E_1 \\ E_2 \end{pmatrix} \in \mathbb{R}^{n\times n_x}, \tag{4.4}$$

with

$$B_1 = A_{11} + E_1E_1^T \in \mathbb{R}^{m\times m}, \quad B_2 = A_{22} + E_2E_2^T \in \mathbb{R}^{(n-m)\times(n-m)}.$$

Note that the diagonal matrix $E_1E_1^T$ perturbs the last n_x diagonal entries of A_{11}, while the diagonal matrix $E_2E_2^T$ perturbs the first n_x diagonal entries of A_{22}. These perturbed entries correspond to the interface nodes, i.e., the nodes that connect to nodes in other subdomains, in Ω_1 and Ω_2.

Consider the relation (4.2) again for a symmetric matrix and note that we have rewritten this in the form of a correction shown in (4.4). From (4.4) and the SMW formula we can derive the equation

$$A^{-1} = B^{-1} + B^{-1}E(\underbrace{I - E^T B^{-1} E}_{X})^{-1} E^T B^{-1} \equiv B^{-1} + B^{-1} E X^{-1} E^T B^{-1}, \tag{4.5}$$

with B and E defined in (4.4). A formalism similar to the one described above was exploited in [139] for the problem of determining the diagonal of the inverse of a matrix.

A first thought for exploiting (4.5) as a preconditioning method in a recursive divide-and-conquer framework, one that will turn out to be impractical, is to approximate $X \in \mathbb{R}^{n_x \times n_x}$ by some nonsingular matrix $\tilde{X}$. Then, the preconditioning operation applied to a vector v is given by

$$u = B^{-1}[v + E\tilde{X}^{-1} E^T B^{-1} v]. \tag{4.6}$$

Thus, each application of a preconditioner based on an approach of this type will require two solves with B, one solve with $\tilde{X}$ and the products with E and E^T. The solution with B, it is assumed, can be obtained by applying (4.5) recursively to the 2 diagonal blocks of B. However, in a multilevel scheme of this sort, it can be verified that applying the preconditioner in (4.6) that needs two solves with B will indeed require 4^{nlev-1} solves at the last level, where $nlev$ denotes the number of levels. Suppose that we can solve the systems at the last level with a linear cost in terms of their dimensions. Since there are 2^{nlev-1} matrices at the last level and their sizes sum up to n, the entire operation count of these solves will increase as $\mathcal{O}\left(n/2^{nlev-1} \cdot 4^{nlev-1}\right) = \mathcal{O}\left(2^{nlev-1} \cdot n\right)$, whereby we can see that the computational cost will explode for a moderately large number of levels. Thus, this scheme will not work and we will need to be careful about ways to exploit equality (4.5).

Practical implementations of the recursive scheme just sketched will be based on various approximations to $B^{-1}E$ and the related matrix X. One possibility is to compute a sparse approximation to $B^{-1}E$ using the ideas from approximate inverses and sparse-sparse techniques, see, e.g., [122]. Sparse approximate inverse methods have been used in a context that is somewhat related in [18]. We expect this approach not to work very well for highly indefinite matrices, as this is a well-known weakness of approximate inverse methods in general. Instead, we consider an alternative, described next, which relies on low-rank approximations.

4.3 Preconditioning methods with low-rank corrections

Our starting point is the relation (4.5). Assume that we have the best 2-norm rank-k approximation to $B^{-1}E$ as obtained from the SVD in the form

$$B^{-1}E \approx U_k V_k^T, \quad V_k^T V_k = I \tag{4.7}$$

where $U_k \in \mathbb{R}^{n \times k}$ and $V_k \in \mathbb{R}^{n_x \times k}$. Then, (4.5) yields several possible approximations.

First, one can just substitute an approximation $\tilde{X}$ for X, as in (4.6). So by replacing $U_k V_k^T$ for $B^{-1}E$ in $X = I - (E^T B^{-1})E$, we let

$$G_k = I - V_k U_k^T E, \tag{4.8}$$

which is an approximation to X. The matrix G_k is of size $n_x \times n_x$ and systems with it can be solved once it is factored at the outset. As was seen in the previous section, a preconditioner based on a recursive application of (4.6), with $\tilde{X}$ replaced by G_k, will see its cost explode with the number of levels, and so this option is avoided.

A computationally viable alternative is to replace every $B^{-1}E$ by its approximation based on (4.7). This leads to a preconditioner of the form

$$M^{-1} = B^{-1} + U_k V_k^T G_k^{-1} V_k U_k^T, \tag{4.9}$$

which means that we can build approximate inverses based on low-rank approximations of the form

$$M^{-1} = B^{-1} + U_k H_k U_k^T, \quad \text{with} \quad H_k = V_k^T G_k^{-1} V_k. \tag{4.10}$$

It turns out that matrix H_k can be computed in a simpler way than by the expression above. Specifically, it will be shown in Section 4.9 that

$$H_k = (I - U_k^T E V_k)^{-1},$$

and H_k is *symmetric*. The alternative expression (4.10) avoids the use of V_k explicitly. Hence, an application of the preconditioner requires one solve with B and a low-rank correction carried out by U_k and H_k. For now, we assume that a system with B can be solved in some unspecified manner.

Approximating $B^{-1}E$ and its transpose on both sides of X when deriving (4.9) from (4.5), may appear to be a poor choice as it may sacrifice accuracy. Instead, a middle ground approach which approximates $B^{-1}E$ on one side only of the expression, will lead to the following,

$$M^{-1} = B^{-1} + B^{-1}EG_k^{-1}V_kU_k^T = B^{-1}[I + EG_k^{-1}V_kU_k^T]. \tag{4.11}$$

However, we will show in Section 4.9 that the preconditioning matrices defined by (4.9) and (4.11) are equal. Therefore, in what follows, we will only consider the preconditioner based on the two-sided low-rank approximations defined by (4.10).

4.4 The multilevel framework

This section describes a framework for constructing the multilevel preconditioner based on the low-rank approximations. We start by defining the matrices at the first level by letting $A_0 \equiv A$, $E_0 \equiv E$ and $B_0 \equiv B$, where A, E and B are the matrices defined in (4.4). The index i will be used for the i-th node of the tree structure shown later in this section. Recall that B_i is a 2×2 block diagonal matrix. We label the two diagonal blocks of B_i as B_{i_1} and B_{i_2} respectively, and write

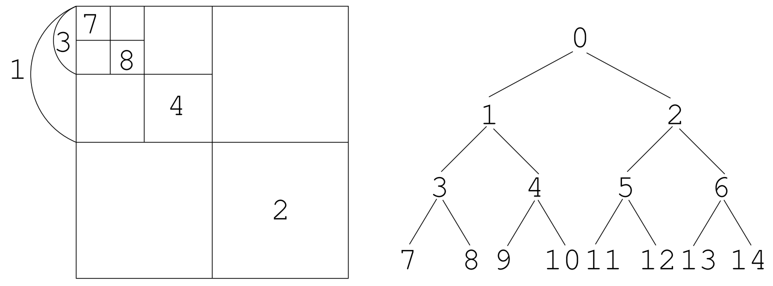
$$A_i = B_i - E_iE_i^T, \quad \text{with} \quad B_i = \text{diag}(B_{i_1}, B_{i_2}). \tag{4.12}$$

The next level is defined by having $A_{i_1} = B_{i_1}$ and $A_{i_2} = B_{i_2}$. The multilevel structure is obtained by putting A_{i_1} and A_{i_2} in the form of (4.12) and repeating the same process until a certain number of levels is reached. At the last level, systems with A_i are solved either exactly by direct methods or approximately by incomplete factorizations. If we view i_1 and i_2 as the children of i, this multilevel matrix splitting can be represented using a binary tree. An example of the 4-level structure is shown in Figure 4.2.

With this multilevel framework, we consider a recursive definition of the preconditioner in (4.10). For a non-leaf node i, suppose $(U_i)_k(V_i)_k^T$ is the best 2-norm rank-k approximation to $B_i^{-1}E_i$ and $(H_i)_k$ is the matrix defined in (4.10). For simplicity, we omit the subscript k from $(U_i)_k$, $(V_i)_k$ and $(H_i)_k$ in the remainder of this chapter without loss of clarity. Then we have the preconditioner M_i^{-1} of A_i as the following

$$M_i^{-1} = B_i^{-1} + U_iH_iU_i^T = \begin{pmatrix} A_{i_1}^{-1} & \\ & A_{i_2}^{-1} \end{pmatrix} + U_iH_iU_i^T \approx \begin{pmatrix} M_{i_1}^{-1} & \\ & M_{i_2}^{-1} \end{pmatrix} + U_iH_iU_i^T,$$

Figure 4.2: Matrix partitioning corresponding to the recursive domain bisection (left) and the obtained binary tree structure (right).



where $M_{i_1}^{-1}$ and $M_{i_2}^{-1}$ are the preconditioners of A_{i_1} and A_{i_2}. On the other hand, if node i is a leaf node, then M_i^{-1} is given by $M_i = L_i U_i$ assuming that an ILU factorization is used for A_i. Putting these together gives a recursive definition of M_i^{-1},

$$M_i^{-1} = \begin{cases} \begin{pmatrix} M_{i_1}^{-1} & \\ & M_{i_2}^{-1} \end{pmatrix} + U_i H_i U_i^T, & \text{if } i \text{ is not a leaf node} \\ \\ U_i^{-1} L_i^{-1}, & \text{otherwise} \end{cases} \tag{4.13}$$

where $M \equiv M_0$ is the MLR preconditioner of the original matrix A. The preconditioning operation of M_i, $x_i = M_i^{-1} b_i$, can be performed by the function `MLRSolve` shown Algorithm 6. In particular, $x = \texttt{MLRSolve}(0, b)$ performs the preconditioning operation of M. In lines 6 and 7 of Algorithm 6, b_{i_1} and b_{i_2} denote the two subvectors of b such that $b = \left(b_{i_1}^T, b_{i_2}^T\right)^T$, which is partitioned conformingly with the partitioning of node i into nodes i_1 and i_2.

4.5 Computing the low-rank approximations

In this section, we describe an approach to compute the low-rank approximations used in the MLR preconditioner. For each non-leaf node i, a rank-k approximation to matrix $B_i^{-1} E_i$ is required of the form

$$Q_i \equiv B_i^{-1} E_i \approx U_i V_i^T \quad \text{with} \quad V_i^T V_i = I. \tag{4.14}$$

Algorithm 6 Function $x_i = \texttt{MLRSolve}(i, b_i)$

```
 1: if $i$ is a leaf-node then
 2:     Solve $A_i x_i = b_i$
 3: else
 4:     $i_1 \leftarrow i$'s left child
 5:     $i_2 \leftarrow i$'s right child
 6:     $y_{i_1} = \texttt{MLRSolve}(i_1,\, b_{i_1})$
 7:     $y_{i_2} = \texttt{MLRSolve}(i_2,\, b_{i_2})$
 8:     $y_i^T = (y_{i_1}^T, y_{i_2}^T)$
 9:     $x_i = y_i + U_i H_i U_i^T b_i$
10: end if
```

Then for the 2-D model problem (4.1), we have $Q_i \in \mathbb{R}^{n_i \times n_x}$. Therefore, computing Q_i requires solves with B_i and the n_x columns of E_i, which will be inefficient if only a few large singular values and the associated singular vectors are wanted, as is the case when the rank $k \ll n_x$. However, the Lanczos bidiagonalization method [140, §10.4] (see also [120, 141]), can be invoked since it can approximate large singular values and the associated singular vectors without the requirement of forming the matrix explicitly.

As is well-known, in the presence of rounding errors, orthogonality in the Lanczos procedure is quickly lost and a form of reorthogonalization is needed in practice. In our approach, we simply use the full reorthogonalization scheme, in which the orthogonality of a new computed Lanczos vector against all previous ones is reinstated at each step. The cost of this step will not be an issue to the overall performance since we only perform a small number of Lanczos steps to approximate a few singular values. More efficient reorthogonalization schemes have been proposed, for details see, e.g., [142–144], but these will not be considered here.

To monitor the convergence of the computed singular values we adopt the approach used in [145]. Let $\theta_j^{(m-1)}$ and $\theta_j^{(m)}$ be the Ritz values, the singular values of the bidiagonal matrices, in two consecutive Lanczos steps, $m-1$ and m. Assume that we want to approximate the k largest singular values of A and $k < m$. Then with a preselected tolerance ϵ, the desired singular values are considered to have converged if

$$\left| \frac{\sigma_m - \sigma_{m-1}}{\sigma_{m-1}} \right| < \epsilon, \text{ where } \sigma_{m-1} = \sum_{j=1}^{k} \theta_j^{(m-1)} \text{ and } \sigma_m = \sum_{j=1}^{k} \theta_j^{(m)}. \tag{4.15}$$

The Lanczos bidiagonalization method requires the matrix only in the form of matrix-vector products and matrix-transpose-vector products. This is very appealing for our case, since the Q_i is implicitly available and recursively defined. Recall that $B_i = \text{diag}(A_{i_1}, A_{i_2})$ and $Q_i \equiv B_i^{-1}E_i$. We can see that applying the Lanczos algorithm to Q_i will require solving linear systems with A_{i_1} and A_{i_2}. Obviously, this can not be a practical approach since it will require solving linear systems with all the A_i's except A_0, even inexactly for instance via ILU factorizations of A_i. An alternative approach is to perform the Lanczos algorithm on a nearby matrix $\tilde{Q}_i \approx Q_i$ with an underlying assumption that its large singular values and the corresponding singular vectors are close to those of Q_i. Let $\tilde{B}_i = \text{diag}(M_{i_1}, M_{i_2})$, which is symmetric, with M_{i_1} and M_{i_2} defined in (4.13) that are the preconditioners of A_{i_1} and A_{i_2}. Then we define $\tilde{Q}_i$ as,

$$\tilde{Q}_i = \tilde{B}_i^{-1}E_i. \tag{4.16}$$

Hence, when performing the Lanczos algorithm on $\tilde{Q}_i$, the matrix-vector product and the matrix-transpose-vector product can be carried out by

$$y = \tilde{Q}_i x = \left(\tilde{B}_i^{-1}E_i\right)x = \tilde{B}_i^{-1}w, \quad \text{with} \quad w = E_i x, \tag{4.17}$$

and

$$y = \tilde{Q}_i^T x = \left(\tilde{B}_i^{-1}E_i\right)^T x = E_i^T z, \quad \text{with} \quad z = \tilde{B}_i^{-1}x. \tag{4.18}$$

With $w = (w_1^T, w_2^T)^T$ partitioned conformingly with $\tilde{B}_i$, the solve $y = \tilde{B}_i^{-1}w$ in (4.17) can be performed as

$$y = \tilde{B}_i^{-1}w = \begin{pmatrix} M_{i_1}^{-1}w_1 \\ M_{i_2}^{-1}w_2 \end{pmatrix} \equiv \begin{pmatrix} e \\ f \end{pmatrix},$$

where e and f can be computed by the recursive function in Algorithm 6 on nodes i_1 and i_2. Specifically, we have $e = \texttt{MLRSolve}(i_1, w_1)$ and $f = \texttt{MLRSolve}(i_2, w_2)$. Likewise, $z = \tilde{B}_i^{-1}x$ in (4.18) can be computed in the same way.

Note that in this approach, computing the low-rank approximation at i, which requires solving liner systems with the matrix $\tilde{B}_i$, will require solves on all the descendants nodes of i, namely, the nodes of the subtree rooted at i. This indicates that building the binary tree structure of an MLR preconditioner should be carried out in a *postorder traversal* (for details of the postorder traversal of a binary tree, see, e.g., [25]).

4.6 Computational cost and memory requirement

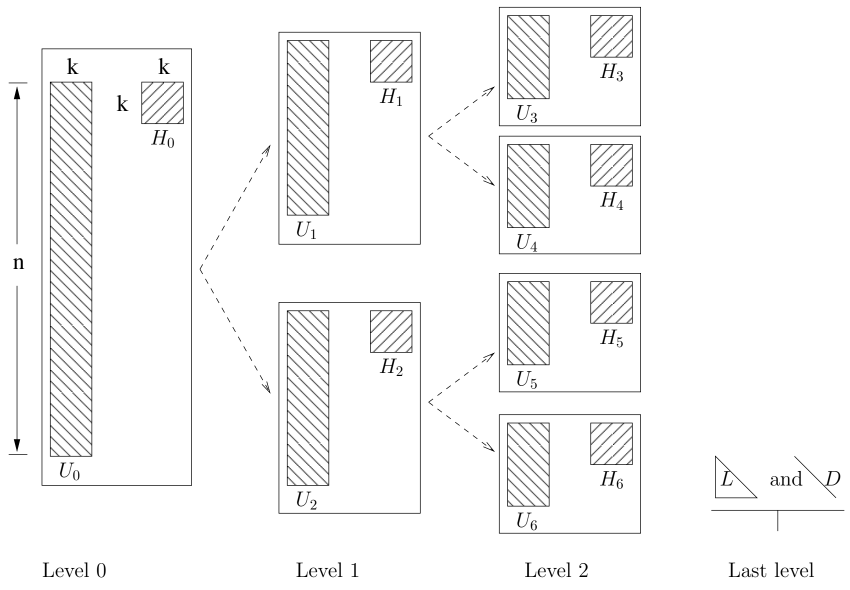
Next we analyze the computational cost of applying the MLR preconditioner, which mainly lies in the solves at the last level (line 2 of Algorithm 6) and the low-rank corrections in all the levels except the last one (line 9). Suppose that the MLR preconditioner of a matrix $A \in \mathbb{R}^{n\times n}$ has m levels. Therefore, there are 2^{m-1} matrices at the last level each of which is of size $n/2^{m-1}$. Applying this preconditioner requires one solve with each matrix. Assume that the cost of solving systems with these matrices is linear in term of their sizes, for example, when these matrices are factored by ILU factorizations and the average number of nonzeros per row in the factors is a constant, denoted by c. Then, the total operation count of these solves will be $\mathcal{O}\left(2^{m-1}\cdot cn/2^{m-1}\right) = \mathcal{O}(cn)$. On the other hand, the cost of the rank-k corrections is in the matrix-vector multiplications with U_i and H_i, which is of the order $\mathcal{O}\big(2(m-1)(kn+k^2)\big)$, where the first term kn and second term k^2 are associated with U_i and H_i respectively and typically the second term k^2 is negligible when k is small. Putting these together, applying the MLR preconditioner has a linear cost, which is $\mathcal{O}\big(\big(2k(m-1)+c\big)n\big)$.

For a non-leaf node i, U_i and H_i are stored for the low-rank correction. Therefore, the memory requirement for storing these matrices is $\mathcal{O}\big((m-1)kn + (2^{m-1}-1)\frac{k^2}{2}\big)$, where the first and second terms are corresponding to U_i and H_i respectively, and typically the second term is small with a moderate number of levels. For a leaf-node j, LU factors are stored when ILU factorizations are used. The matrices stored at all the levels of an MLR preconditioner are illustrated schematically in Figure 4.3.

4.7 Generalization via domain decomposition

To generalize the above scheme to unstructured matrices, we take a DD viewpoint. In the situation when a domain Ω is partitioned into Ω_1 and Ω_2, the unknowns are partitioned into 4 parts: the interior unknowns and the interface unknowns in Ω_1 and Ω_2. Denote by m_i the number of the interface unknowns in Ω_i, for $i = 1, 2$. If the

Figure 4.3: Matrices stored at all levels of an MLR preconditioner.



interface unknowns are listed last in each subdomain, $A \in \mathbb{R}^{n\times n}$ has the following form

$$A = \left(\begin{array}{cc|cc} \hat{B}_1 & \hat{F}_1 & & \\ \hat{F}_1^T & C_1 & & -W \\ \hline & & \hat{B}_2 & \hat{F}_2 \\ & -W^T & \hat{F}_2^T & C_2 \end{array}\right), \tag{4.19}$$

where $\hat{F}_i$ represents the connections between the interior and interface unknowns corresponding to $\hat{B}_i$ and C_i respectively, for Ω_i, $i = 1, 2$. In addition, $W \in \mathbb{R}^{m_1\times m_2}$ represents the coupling between the interface unknowns in Ω_1 and those in Ω_2, which is essentially the edge-separator. Consider the following matrix $E \in \mathbb{R}^{n\times m_1}$ defined with respect to the above partitioning of unknowns,

$$E^T = \begin{pmatrix} 0 & I & 0 & W \end{pmatrix}. \tag{4.20}$$

Then, we obtain the following generalization of (4.4):

$$A = \begin{pmatrix} B_1 & \\ & B_2 \end{pmatrix} - EE^T \quad \text{with} \quad B_i = \begin{pmatrix} \hat{B}_i & \hat{F}_i \\ \hat{F}_i^T & C_i + D_i \end{pmatrix} \text{ and } \begin{cases} D_1 = I \\ D_2 = W^T W \end{cases}. \tag{4.21}$$

For a model problem, e.g., the one discussed at the beginning of Section 4.2, with a careful ordering of the interface unknowns of Ω_1 and Ω_2, we have $W = I$, which yields $D_1 = D_2 = I$. Here, however, there could be a big imbalance between the expressions of B_1 and B_2. One way to mitigate the imbalance is by weighing the two nonzero terms in (4.20). The matrix E can then be redefined as

$$E_\alpha^T = \begin{pmatrix} 0 & \alpha I & 0 & \alpha^{-1} W \end{pmatrix}, \tag{4.22}$$

which yields $D_1 = \alpha^2 I$ and $D_2 = \alpha^{-2} W^T W$. In order to make the spectral radius of D_1 equal to that of D_2, we take α to be the square root of the largest singular value of W.

The above method can be generalized in a number of ways by considering any factorization of W, which can be a rectangular matrix. Consider any factorization $W = X_1 X_2$. Looking only at the relevant blocks of the matrix (4.19), we can write

$$\begin{pmatrix} 0 & W \\ W^T & 0 \end{pmatrix} = \begin{pmatrix} X_1 \\ X_2^T \end{pmatrix} \begin{pmatrix} X_1^T & X_2 \end{pmatrix} - \begin{pmatrix} X_1 X_1^T & 0 \\ 0 & X_2^T X_2 \end{pmatrix}.$$

This means that for any factorization $W = X_1 X_2$ we can replace E in (4.20) and D_1, D_2 in (4.21) with

$$E^T = \begin{pmatrix} 0 & X_1^T & 0 & X_2 \end{pmatrix}, \quad D_1 = X_1 X_1^T, \quad D_2 = X_2^T X_2. \tag{4.23}$$

With this, the next simplest generalization of (4.20), is to take X_1 to be a diagonal matrix, instead of αI. In this case, $X_2 = X_1^{-1} W$, and X_1 is some nonsingular diagonal matrix to be selected. We need to balance between X_1^2 and $X_2^T X_2 = W^T X_1^{-2} W$. Note that these two matrices can be of (slightly) different dimensions. Multiplying the matrix W to the left by X_1^{-1} corresponds to scaling the rows of W. We will *scale each row by the square root of its norm*, so

$$X_1 = \operatorname{diag}\{\sqrt{\omega_i}\}, \quad \omega_i = \|e_i^T W\|_2.$$

In this way,

$$X_1^2 = \operatorname{diag}\{\omega_i\} \quad \text{and} \quad X_2^T X_2 = W^T \operatorname{diag}\{\omega_i^{-1}\} W.$$

Although these two matrices can be of different dimensions, note that the norm of the i-th row of X_1 (which is $\sqrt{\omega_i}$) is the same as the norm of the i-th row of X_2. Of course,

$X_1 X_1^T$ is diagonal, and $X_2^T X_2$ is not. But in terms of their general magnitude these terms will be well balanced in most cases.

Finally, we also considered another balancing option based on the LU factorization $W = LU$, where $L \in \mathbb{R}^{m_1 \times l}$ and $U \in \mathbb{R}^{l \times m_2}$ with $l = \min(m_1, m_2)$. Note that W may not have a full rank. We then balance the LU factors by a diagonal matrix $\Lambda \in \mathbb{R}^{l \times l}$ as $W = (L\Lambda)\left(\Lambda^{-1}U\right) \equiv \tilde{L}\tilde{U}$. Here Λ is selected such that $\|\tilde{l}_i\|_2 = \|\tilde{u}_i^T\|_2$ for $i = 1, \ldots l$, where $\tilde{l}_i$ is the i-th column of $\tilde{L}$ and $\tilde{u}_i^T$ is the i-th row of $\tilde{U}$. Then formulas (4.23) are used with $X_1 = \tilde{L}, X_2 = \tilde{U}$. This strategy was explored but did not yield results as good as those of the other two methods discussed above.

4.8 Updating an MLR preconditioner

The MLR preconditioner has the appealing property that it can be easily updated. By this we mean that if the performance of the iterative procedure is not satisfactory, one can compute and use an improved version of the preconditioner without foregoing work previously performed. This is a quality shared by all approximate inverse preconditioners but the multilevel nature of MLR preconditioners requires particular considerations. With a rank-k MLR preconditioner already computed, an incremental rank-$(k+1)$ one can be obtained by resorting to a form of deflation. For each non-leaf node i, we perform the Lanczos algorithm on the *deflated matrix* of $\tilde{Q}_i$ in (4.16), $K_i \equiv \tilde{Q}_i - U_i V_i^T$, to compute the best 2-norm rank-1 approximation of K_i. Although V_i is not necessarily available, by assuming that $V_i = \tilde{Q}_i^T U_i$, we have $K_i = \tilde{Q}_i - U_i U_i^T \tilde{Q}_i = (I - U_i U_i^T)\tilde{Q}_i$, where U_i is available. In the Lanczos algorithm, the matrix-vector operation, $y = K_i x$, can be performed as $y = (I - U_i U_i^T)(\tilde{Q}_i x)$, where the term $\tilde{Q}_i x$ is computed as in (4.17), and $y = K_i^T x$ can be computed likewise. Suppose that the rank-1 approximation to K_i is of the form $K_i \approx uv^T$. Then, $\bar{U}_i$ in the rank-$(k+1)$ preconditioner is updated by appending the new vector, i.e., $\bar{U}_i = (U_i, u)$, and for $\bar{H}_i$, we have

$$\bar{H}_i^{-1} = I - \bar{U}_i^T E_i (V_i, v) = I - \begin{pmatrix} U_i^T E_i V_i & U_i^T E_i v \\ u^T E_i V_i & u^T E_i v \end{pmatrix} = \left(\begin{array}{c|c} H_i^{-1} & U_i^T E_i v \\ \hline u^T E_i V_i & 1 - u^T E_i v \end{array}\right). \quad (4.24)$$

Under a few assumptions, $\bar{H}_i$ is also symmetric. Specifically, we assume that $V_i^T v = 0$, $K_i v = u$ and $\tilde{Q}_i V_i = \tilde{B}_i^{-1} E_i V_i = U_i$. Then, the symmetry of $\bar{H}_i$ can be shown by

$$\begin{aligned} u^T E_i V_i &= v^T K_i^T E_i V_i = v^T (\tilde{Q}_i - U_i V_i^T)^T E_i V_i \\ &= v^T \tilde{Q}_i^T E_i V_i = v^T \tilde{Q}_i^T \tilde{B}_i U_i = v^T (E_i^T \tilde{B}_i^{-1}) \tilde{B}_i U_i = v^T E_i^T U_i. \end{aligned}$$

The symmetry of $\bar{H}_i$ is important because for the (2,1) entry, $u^T E_i V_i$, of the last term in (4.24), V_i is not necessarily available. This entry can be obtained by symmetry, i.e., as $v^T E_i^T U_i$. Note that the assumptions made above can be enforced, but they are satisfied approximately in practice and this should be enough to exploit the symmetry relation above since we are only building an approximate inverse for preconditioning. Finally, as in the process of building an MLR preconditioner, the incremental update should be also performed in a postorder traversal of the binary tree.

4.9 Analysis

This section will examine a few questions which have not been addressed so far, regarding the preconditioner defined in previous sections. First, we will explore a number of algebraic relationships already mentioned, which lead to a few simplifications of the algorithm. Second, we will attempt to explain on a model problem, why a small rank approximation is sufficient to capture the difference $A^{-1} - B^{-1}$ in (4.5).

4.9.1 Algebraic relationships and equivalences

We were surprised to find in our experiments that the preconditioners given by (4.9) and (4.11) are identical. In addition, we also noted that the matrix H_k (as well as G_k in some cases) is symmetric which is not immediately obvious. This section explores some of these issues. The main assumption we make in order to prove these results has to do with the form of the rank-k approximation $U_k V_k^T$, which will be referred to as assumption (H):

$$\boxed{\textbf{(H)}\text{: } U_k V_k^T \text{ is the best 2-norm rank-}k\text{ approximation to } B^{-1}E, \text{ and } V_k^T V_k = I.}$$

We begin with a simple lemma which provides an explicit formula for the inverse of the matrix G_k of (4.8).

Lemma 4.9.1 *Let G_k be defined by (4.8) and assume that the matrix $I - U_k^T E V_k$ is nonsingular and that we have $V_k^T V_k = I$. Then,*

$$G_k^{-1} = I + V_k \hat{H}_k U_k^T E \quad \textit{with} \quad \hat{H}_k = (I - U_k^T E V_k)^{-1}. \tag{4.25}$$

Furthermore, the following relation holds:

$$V_k^T G_k^{-1} V_k = \hat{H}_k, \tag{4.26}$$

i.e., the matrices H_k in (4.10) and the matrix $\hat{H}_k$ given in (4.25) are the same.

Proof. The expression for the inverse of G_k is provided by the SMW formula:

$$(I - V_k(U_k^T E))^{-1} = I + V_k \hat{H}_k U_k^T E \quad \text{with} \quad \hat{H}_k = (I - U_k^T E V_k)^{-1},$$

which is a valid expression under the assumption that $I - U_k^T E V_k$ is nonsingular.

Relation (4.26) follows from the observation that $\hat{H}_k^{-1} + U_k^T E V_k = I$, which yields

$$V_k^T G_k^{-1} V_k = V_k^T (I + V_k \hat{H}_k U_k^T E) V_k = I + \hat{H}_k U_k^T E V_k = \hat{H}_k [\hat{H}_k^{-1} + U_k^T E V_k] = \hat{H}_k.$$

This completes the proof. □

Since the matrices $\hat{H}_k$ and H_k are identical we will use the symbol H_k to denote them both in what follows.

Recall that $U_k \in \mathbb{R}^{n \times k}$, $V_k \in \mathbb{R}^{n_x \times k}$. Under the assumptions we have

$$B^{-1} E = U_k V_k^T + Z \quad \text{with} \quad Z V_k = 0 \tag{4.27}$$

where $Z \in \mathbb{R}^{n \times n_x}$. This is because if $B^{-1} E = U \Sigma V^T$ then the best rank-k approximation is given by $\sum_{i \le k} \sigma_i u_i v_i^T$ and so $Z = \sum_{i > k} \sigma_i u_i v_i^T$. A number of simple properties follow from this observation.

Proposition 4.9.1 *Assume that* **(H)** *is satisfied and that B is symmetric positive definite. Then we have $U_k^T E V_k = U_k^T B U_k$ and the matrix $U_k^T E V_k$ is therefore symmetric positive definite.*

Proof. We write,

$$U_k^T E V_k = U_k^T B (B^{-1} E) V_k = U_k^T B (U_k V_k^T + Z) V_k = U_k^T B U_k$$

Since B is SPD and U_k is of full rank then it follows that $U_k^T E V_k$ is SPD. □

Proposition 4.9.2 *Assume that* **(H)** *is satisfied and that the matrix* $I - U_k^T E V_k$ *is nonsingular. Then the two preconditioning matrices defined by (4.9) and (4.11), respectively, are equal.*

Proof. Comparing (4.9) and (4.11) we note that all we have to prove is that $B^{-1}EG_k^{-1}V_kU_k^T = U_kV_k^TG_k^{-1}V_kU_k^T$. The proof requires the expression for the inverse of G_k that is provided by Equation (4.25) of Lemma 4.9.1, a valid expression under the assumption that $I - U_k^T E V_k$ is nonsingular. From this it follows that:

$$\begin{aligned} B^{-1}EG_k^{-1}V_kU_k^T &= (U_kV_k^T + Z)G_k^{-1}V_kU_k^T \\ &= (U_kV_k^T + Z)V_k\left[I + H_kU_k^TEV_k\right]U_k^T \qquad (4.28) \\ &= (U_kV_k^T)V_k\left[I + H_kU_k^TEV_k\right]U_k^T \qquad (4.29) \\ &= (U_kV_k^T)\left[I + V_kH_kU_k^TE\right]V_kU_k^T \\ &= U_kV_k^TG_k^{-1}V_kU_k^T, \end{aligned}$$

where we have used the relation (4.27) in going from (4.28) to (4.29). □

Proposition 4.9.1 along with the expressions (4.10) of the preconditioner and (4.25) for H_k lead to yet another way of expressing the preconditioner.

Proposition 4.9.3 *Under the same assumptions as those of Proposition 4.9.2, the preconditioner* M *given by equation (4.10) satisfies the relation:*

$$M = B - BU_kU_k^TB. \qquad (4.30)$$

Proof. We need to invert the matrix M given in the above expression in order to compare it with (4.10). Using the SMW formula leads to the expression,

$$(B - (BU_k)(BU_k)^T)^{-1})^{-1}(BU)^TB^{-1} = B^{-1} + U_k(I - U_k^TBU_k)^{-1}U_k^T$$

Using the relation $U_kBU_k^T = U_k^TEV_k$ from Proposition 4.9.1 and the expression of H_k obtained from Lemma 4.9.1 leads to the same expression as (4.10) for the inverse of M. □

The expression of the preconditioner given by the above proposition provides some insight on the nature of the preconditioner. Consider the extreme situation when we use the full decomposition $B^{-1}E = U_kV_k^T$, so $k = n_x$ and $Z = 0$ in (4.27). Then,

$E = BU_kV_k^T$ and hence we will have $BU_k = EV_k$. Since V_k is now a square (unitary) matrix, therefore,

$$M = B - EV_kV_k^TE^T = B - EE^T,$$

which is the original matrix per (4.4). Not surprisingly, we do indeed obtain an exact preconditioner when an exact decomposition $B^{-1}E = U_kV_k^T$ is used. When an inexact decomposition $B^{-1}E \approx U_kV_k^T$ is used, $(k < n_x, Z \neq 0)$ then we have $B^{-1}E = U_kV_k^T + Z$ and we obtain $(E - BZ) = BU_kV_k^T$. We can now ask what approximation is the preconditioner making on the original matrix in this situation. Remarkably, the preconditioner used simply corresponds to a modification of (4.4) in which E is perturbed by $-BZ$.

Proposition 4.9.4 *Under the same assumptions as those of Proposition 4.9.2, the preconditioner M given by equation (4.10) satisfies the relation:*

$$M = B - (E - BZ)(E - BZ)^T,$$

where Z is given in (4.27).

Proof. The proof follows immediately from the above arguments, Proposition 4.9.3, and the equality

$$(E - BZ)(E - BZ)^T = BU_kV_k^TV_kU_k^TB = BU_kU_k^TB.$$

□

The final question we would like to answer now is: Under which condition is the preconditioner symmetric positive definite? The following result gives a necessary and sufficient condition. In the following we will say that the preconditioner (4.10) is *well-defined* when H_k exists, i.e., when $I - U_k^TEV_k$ is nonsingular.

Theorem 4.9.1 *Let the assumptions of Proposition 4.9.1 be satisfied. Then the preconditioner given by (4.10) is well defined and symmetric positive definite if and only if $\rho(U_k^TEV_k) < 1$.*

Proof. Recall from Proposition (4.9.1) that the matrix $U_k^TEV_k$ is a symmetric positive definite matrix. If $\rho(U_k^TEV_k) < 1$ then clearly the eigenvalues of $I - U_k^TEV_k$ are all positive. Therefore, the matrix $I - U_k^TEV_k$ is symmetric positive definite and it is nonsingular so the preconditioner M is well defined. Its inverse, H_k is SPD. As a result the matrix M in (4.10) is also SPD.

To prove the converse, we consider expression (4.30) of the preconditioner and make the assumption that it is positive definite. Under this assumption, $U_k^T M U_k$ is SPD since U_k is of full rank. Now observe that if we set $S \equiv U_k^T B U_k$ then

$$U_k^T M U_k = U_k^T B U_k - U_k^T B U_k U_k^T B U_k = S - S^2.$$

The eigenvalues of $S - S^2$ are positive. Since S is symmetric positive definite any eigenvalue λ of S is positive. Therefore λ is positive and such that $\lambda - \lambda^2$ is also positive. This implies that $0 < \lambda < 1$ and the proof is complete since $U_k^T E V_k = U_k^T B U_k = S$ (Proposition 4.9.1). □

Finally, the above theorem can be extended to the recursive multilevel preconditioner defined by (4.13). The following result shows a sufficient condition for the preconditioning matrix M_i to be SPD. If we denote by $st(i)$ the subtree rooted at i, then in the following, we let $l(i)$ be a set consisting of all the leaf nodes of $st(i)$ and $nl(i)$ be a set consisting of all the non-leaf nodes of $st(i)$. We will refer to $(\mathbf{H_i})$ as the assumption **(H)** above at node i: $B^{-1}E$ in **(H)** is replaced with $\tilde{B}_i^{-1}E_i$ where $\tilde{B}_i = \mathrm{diag}(M_{i_1}, M_{i_2})$, and $U_k V_k^T$ is now $U_i V_i^T$ and V_k becomes V_i.

Corollary 4.9.1 *Assume that* $(\mathbf{H_i})$ *is satisfied and that for all* $j \in l(i)$ *the matrices* M_j *are SPD. Then the preconditioning matrix* M_i *given by* (4.13) *is well defined and SPD if* $\rho(U_j^T E_j V_j) < 1$ *for all* $j \in nl(i)$.

Proof. The proof consists of a simple inductive argument which exploits the previous result which is valid for two levels. □

4.9.2 Two-level analysis for a 2-D model problem

The analysis of the multilevel preconditioner proposed in this paper is difficult for general problems. In the simplest case when the matrix A originates from a 2-D Laplacian discretized by centered differences, we can easily perform a spectral analysis of the preconditioner to provide some insight. Specifically, one of the questions which we wish to address is why a low rank approximation, sometimes very low, yields a reasonable preconditioner. Looking at equation (4.5) this can be understood only by a rapid decay of the eigenvalues of the difference between A^{-1} and B^{-1}. This leads us to a spectral analysis of the matrix $B^{-1}EX^{-1}E^TB^{-1}$ in (4.5). Note that the method essentially approximates this matrix with a low-rank approximation, which is itself obtained from approximating $B^{-1}E$. As will be seen, $B^{-1}E$ itself may not show as good a decay in its singular values as does $B^{-1}EX^{-1}E^TB^{-1}$.

Assume that $-\Delta$ is discretized on a grid of size $(n_x+2)\times(n_y+2)$, with Dirichlet boundary conditions and that the ordering is major along the x direction. Call T_x the tridiagonal matrix of dimension $n_x \times n_x$ which discretizes $\partial^2/\partial x^2$, and similarly T_y the tridiagonal matrix of dimension $n_y \times n_y$ which discretizes $\partial^2/\partial y^2$. The scaling term $1/h^2$ is omitted so these matrices have the constant 2 on their main diagonal and -1 on their co-diagonals. Finally, we will call I_x, I_y the identity matrices of size n_x, n_y respectively. Then, the matrix A which results from discretizing $-\Delta$ can be written using Kronecker products as follows:

$$A = T_y \otimes I_x + I_y \otimes T_x. \tag{4.31}$$

In this section, we will make extensive use of Kronecker products. We need to recall a few basic rules when working with such products. First, for any $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$, $C \in \mathbb{R}^{n\times s}$, $D \in \mathbb{R}^{q\times t}$ we have

$$(A \otimes B)\ (C \otimes D) = (AC) \otimes (BD)\ . \tag{4.32}$$

This is valid in particular when $s = t = 1$, i.e., when B and D are vectors, which we denote by $u \in \mathbb{R}^n, v \in \mathbb{R}^q$:

$$(A \otimes B)(u \otimes v) = (Au) \otimes (Bv)\ . \tag{4.33}$$

Another simple rule involves the transposition operation:

$$(A \otimes B)^T = A^T \otimes B^T. \tag{4.34}$$

Eigenvalues and eigenvectors of A can be easily obtained from the expression (4.31) by using the relation (4.33). The eigenvectors of A are of the form $u_i \otimes v_j$ where u_i is an eigenvector of T_y associated with an eigenvalue μ_i and v_j is an eigenvector of T_x associated with an eigenvalue λ_i. The eigenvalues of A are $\mu_i + \lambda_j$.

Consider now the case when B is given by (4.4). In terms of the grid, the domain is separated horizontally, keeping $n_y/2$ horizontal lines in one half of the domain and the remaining lines in the other half. The matrix E itself can be written as a Kronecker product. If e is the vector of $\mathbb{R}^{n_y}$ which has zero entries except in locations $n_y/2$ and $n_y/2+1$ where the entries are 1, then clearly $E = e \otimes I_x$ and from (4.34) and (4.32)

$$EE^T = (e \otimes I_x)(e \otimes I_x)^T = (e \otimes I_x)(e^T \otimes I_x) = (ee^T) \otimes I_x,$$

so

$$B = A + EE^T = T_y \otimes I_x + I_y \otimes T_x + (ee^T) \otimes I_x) = (T_y + ee^T) \otimes I_x + I_y \otimes T_x.$$

The eigenvalues and eigenvectors of B can be obtained in a similar way to those of A. The only difference is that T_y is now perturbed by the rank-one matrix ee^T into $\tilde{T}_y = T_y + ee^T$. In fact this perturbation corresponds to splitting the one-dimensional domain (y direction) in two and applying the Neumann boundary condition at the interface. As it turns out the eigenvalues of $\tilde{T}_y$ are simply related to those of T_y. Details of these eigenvalues will be shown in the next section.

Eigenvalues of T_y and $\tilde{T}_y$

We consider the $n \times n$ matrix

$$T = \text{Tridiag}[-1, 2, -1]$$

whose main diagonal entries are all 2 and its co-diagonals entries are all -1. It is assumed that n is even and we set $m = n/2$. As is well-known [12, Sec. 13.2], the eigenvalues of T are given by

$$\mu_k = 4\sin^2\frac{\theta_k}{2} \quad \text{with} \quad \theta_k = \frac{k\pi}{n+1}, \quad k = 1, 2, \cdots, n$$

and each corresponding eigenvector u_k has components

$$u_{k,i} = \sin(i\ \theta_k), \quad i = 1, \cdots, n. \tag{4.35}$$

We are interested in the eigenvalues and eigenvectors of $\tilde{T} = T + ee^T$, where e was defined in Section (4.9.2) as the vector of $\mathbb{R}^n$ that has entries equal to 1 in locations m

and $m+1$ and zero elsewhere. This matrix is shown below for the case $n=8$:

$$\tilde{T} = T + ee^T = \left[\begin{array}{cccc|cccc} 2 & -1 & & & & & & \\ -1 & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 3 & & & & \\ \hline & & & & 3 & -1 & & \\ & & & & -1 & 2 & -1 & \\ & & & & & -1 & 2 & -1 \\ & & & & & & -1 & 2 \end{array}\right]$$

As it turns out the eigenvalues and eigenvectors of $\tilde{T}$ can be readily obtained from those of T. Consider an eigenvalue μ_k of T when k is even, which we write as $k = 2k'$.

In this situation, an eigenvector of T associated with the eigenvalue μ_k can be written in the form

$$u_k = \begin{bmatrix} v \\ -v^b \end{bmatrix},$$

where $.^b$ indicates a backward ordering, i.e., a permutation of a vector with the permutation: $m, m-1, \cdots, 1$, so $v^b(i) = v(n+1-i)$ for $i = 1, \cdots, n$. This is because if we set $i = (n+1) - j$ in (4.35), we get

$$u_{k,i} = \sin\left(\frac{2k'(n+1-j)\pi}{n+1}\right) = -\sin\left(\frac{2k'j\pi}{n+1}\right) = -\sin\left(kj\theta_k\right) = -u_{k,j}$$

which shows that the first half of the components of u_k are the same as those of the second half, listed backward and with a negative sign. In particular note that $u_{k,m} = -u_{k,m+1}$ and so $e^T u_k = 0$. As a result, $(T - \mu_k I)u_k = 0$ implies that $(T - \mu_k I + ee^T)u_k = 0$, i.e., *each eigenpair μ_k, u_k of T for an even k is also an eigenpair of $\tilde{T}$.* This accounts for half of the eigenpairs of $\tilde{T}$. To get the other half (odd k), observe that the particular structure of $\tilde{T}$ shows that its eigenvalues are all double. The eigenvectors corresponding to the odd eigenvalues ($\mu_{k-1} = \mu_k$ for k even), can be easily seen to be given by

$$u_{k-1} = \begin{bmatrix} v \\ v^b \end{bmatrix}.$$

The whole set defined in this way does indeed form an orthonormal system of eigenvectors. This way of defining the set of eigenvectors is of course not unique. In the end we have the following eigenvalues defined for $k = 2, 4, \cdots, n-2, n$:

$$\mu_{k-1} = \mu_k = 4\sin^2 \frac{k\pi}{2(n+1)} \ .$$

For the associated eigenvectors, we will now use the same notation as that of Section 4.9.2, and, for a more convenient notation, we will swap the odd and even vectors,

$$\tilde{u}_{k-1} = \begin{bmatrix} v_k \\ -v_k^b \end{bmatrix}, \quad \tilde{u}_k = \begin{bmatrix} v_k \\ v_k^b \end{bmatrix}, \quad v_{k,i} = \sin\left(i\,\frac{k\pi}{n+1}\right), \quad i = 1 \cdots, m \ .$$

Note that with this change of notation, $e^T\tilde{u}_k = 0$ for odd values of k, whereas $e^T\tilde{u}_k = 2\sin(m\theta_k)$ when k is even.

The decay property of $\mathbf{A^{-1} - B^{-1}}$

We will denote by $\tilde{\mu}_i$ and $\tilde{u}_i$ the eigenvalues and vectors of $\tilde{T}_y$, noting in passing that, due to the structure of $\tilde{T}_y$, all eigenvalues are double. The eigenvalues and vectors of T_x are denoted by λ_i and v_i. It is easy to invert B by using its spectral decomposition:

$$B^{-1} = \sum_{i=1}^{n_y}\sum_{j=1}^{n_x} \frac{1}{\tilde{\mu}_i + \lambda_j}(\tilde{u}_i \otimes v_j)(\tilde{u}_i \otimes v_j)^T = \sum_{i,j} \frac{1}{\tilde{\mu}_i + \lambda_j}(\tilde{u}_i\tilde{u}_i^T) \otimes (v_j v_j^T),$$

which is a *linear combination of Kronecker products of eigenprojectors*. Thus,

$$B^{-1}E = \sum_{i,j} \frac{1}{\tilde{\mu}_i + \lambda_j}\left[(\tilde{u}_i\tilde{u}_i^T) \otimes (v_j v_j^T)\right](e \otimes I_x) = \sum_{i,j} \frac{\tilde{u}_i^T e}{\tilde{\mu}_i + \lambda_j}\tilde{u}_i \otimes (v_j v_j^T) \, . \qquad (4.36)$$

We are interested in the difference $A^{-1} - B^{-1}$ which is the very last term of (4.5), i.e., the matrix $B^{-1}EX^{-1}E^TB^{-1}$ where $X = I - E^TB^{-1}E$. The nonzero eigenvalues of this matrix are the same as those of $X^{-1}E^TB^{-2}E$. First, consider the matrix $B^{-2}E$ which can be expanded in a way similar to that of $B^{-1}E$ above:

$$B^{-2}E = \sum_{i,j} \frac{\tilde{u}_i^T e}{(\tilde{\mu}_i + \lambda_j)^2}\tilde{u}_i \otimes (v_j v_j^T).$$

Next, for the matrix X^{-1}, we have from (4.36):

$$\begin{aligned} E^TB^{-1}E &= \sum_{i,j} \frac{\tilde{u}_i^T e}{\tilde{\mu}_i + \lambda_j}(e^T \otimes I_x)(\tilde{u}_i \otimes (v_j v_j^T)) \\ &= \sum_{i,j} \frac{(\tilde{u}_i^T e)^2}{\tilde{\mu}_i + \lambda_j}(v_j v_j^T) \equiv \sum_{j=1}^{n_x} \alpha_j v_j v_j^T \, , \end{aligned} \qquad (4.37)$$

where we have denoted by α_j the sum:

$$\alpha_j = \sum_{i=1}^{n_y} \frac{(\tilde{u}_i^T e)^2}{\tilde{\mu}_i + \lambda_j} \ . \tag{4.38}$$

Equation (4.37) expresses $E^T B^{-1} E$ as a linear combination of the eigenprojectors $v_j v_j^T$ of T_x. The matrix X^{-1} can be readily obtained from this since $X = I - \sum \alpha_j v_j v_j^T = \sum (1 - \alpha_j) v_j v_j^T$:

$$X^{-1} = \sum_{j=1}^{n_x} \frac{1}{1 - \alpha_j} v_j v_j^T.$$

Finally, a spectral expansion for $E^T B^{-2} E$ can be obtained in a very similar way as in (4.37):

$$E^T B^{-2} E = \sum_{j=1}^{n_x} \beta_j v_j v_j^T, \quad \beta_j = \sum_{i=1}^{n_y} \frac{(\tilde{u}_i^T e)^2}{(\tilde{\mu}_i + \lambda_j)^2}. \tag{4.39}$$

In the end, the eigenvalues of $X^{-1} E^T B^{-2} E$ are given by

$$\gamma_j = \frac{\beta_j}{1 - \alpha_j}, \quad j = 1, \cdots, n_x \ . \tag{4.40}$$

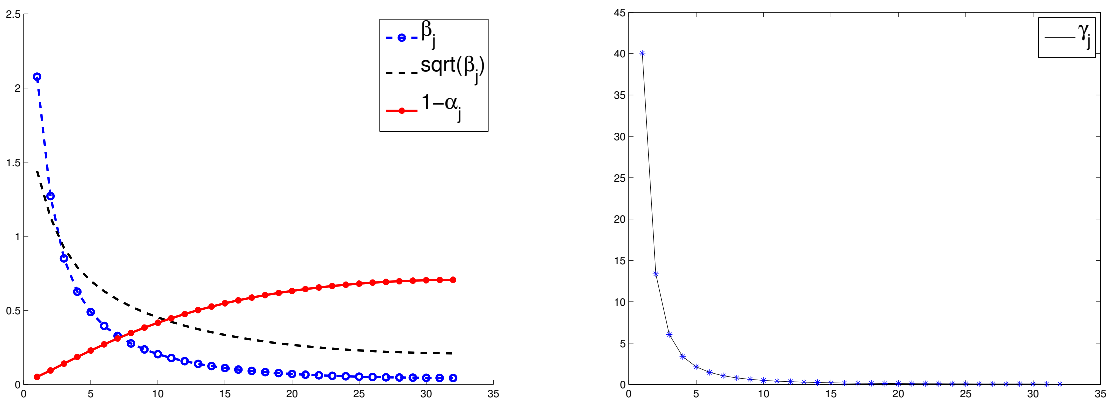
We can now have an explicit expression of these eigenvalues since the quantities above are all known. From the appendix, we see that $e^T \tilde{u}_i = 0$ for i odd, and so we set $i = 2k$, for $k = 1, \cdots, n_y/2$ in (4.39) and (4.38), to obtain:

$$\beta_j = \sum_{k=1}^{n_y/2} \frac{\sin^2 \frac{n_y k\pi}{n_y+1}}{4\left(\sin^2 \frac{k\pi}{n_y+1} + \sin^2 \frac{j\pi}{2(n_x+1)}\right)^2}, \quad \alpha_j = \sum_{k=1}^{n_y/2} \frac{\sin^2 \frac{n_y k\pi}{n_y+1}}{\sin^2 \frac{k\pi}{n_y+1} + \sin^2 \frac{j\pi}{2(n_x+1)}} \ . \tag{4.41}$$

An illustration of these two functions along with the eigenvalues γ_j is shown in Figure 4.4. The scalars β_j decay from the largest value β_1. Note that the singular values of $B^{-1}E$ are the square roots of the β_j's and these do not decay particularly fast. The scalars β_j's, their squares, decay somewhat rapidly. However, the more interesting observation is the very rapid decay of the ratios (4.40). While the decay of the singular values of $B^{-1}E$ is not sufficient by itself to explain a good approximation of $A^{-1} - B^{-1}$ by a low rank matrix, *what makes a difference is the squaring of* $B^{-1}E$ *and the strong damping effect from the term* X^{-1}*, which contributes the divisions by* $1 - \alpha_j$*.*

Because $B^{-1}E$ does not necessarily have fast-decaying singular values, it may be argued that the approach based on extracting a low-rank approximation from it may have problems. However, the above expansion shows why this is not an issue. For example, assume that we use, say 15 singular vectors when approximating $B^{-1}E$, while only 3 are needed to approximate $A^{-1} - B^{-1}$ well. Then in the related approximation of $B^{-1}EX^{-1}E^TB^{-1}$, the resulting rank-1 terms beyond the 3rd rank will be damped by the factors γ_j which are relatively small, and so they will contribute very little to the resulting preconditioner.

Figure 4.4: Illustration of the decay of the eigenvalues of $B^{-1}EX^{-1}E^TB^{-1}$ for the case when $nx = ny = 32$. The left panel shows the coefficients $\beta_j, 1-\alpha_j$ and the square roots of β_j's, which are the singular values of $B^{-1}E$. The right panel shows the ratios γ_j, the eigenvalues of $B^{-1}EX^{-1}E^TB^{-1}$. In this particular case 3 eigenvectors will capture 92 % of the inverse whereas 5 eigenvectors will capture 97% of the inverse.



4.10 Compression of the U-matrices with tensors

The major memory requirement in MLR preconditioners is for storing the U-matrices for the low-rank corrections. As discussed in 4.6, for $A \in \mathbb{R}^{n\times n}$, the memory cost of storing U_i in a rank-k MLR preconditioner with m levels is of the order $\mathcal{O}\big((m-1)kn\big)$. Clearly, this memory cost grows linearly with m. For a larger problem, typically we use more levels in order to control the sizes of the matrices at the last level such that the cost of solving systems with these matrices is still affordable. So, the memory requirement will eventually be an issue when considering solving larger and larger problems. It is desired, if possible at all, to compress U_i. Here we consider an approach for the model problems on regular grids to reduce the memory cost by using tensor representations and tensor approximations.

4.10.1 Some tensor definitions

In the following, we recall a few definitions of tensors. More definitions and properties of tensors can be found in [146–149] and the references therein.

Definition 4.10.1 (Matricization) *An N-th order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ admits N types of matricization (or termed unfolding [146]) with the N modes. The mode-n matricization, denoted by $A_{(n)}$, is an $I_n \times (I_{n+1} I_{n+2} \ldots I_N I_1 I_2 \ldots I_{n-1})$ matrix. Assume that the columns of $A_{(n)}$ are in the forward cyclic order as the one used in [147] and then we have*

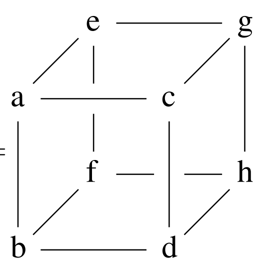
$$\left[A_{(n)}\right]_{u,v} = \mathcal{A}_{i_1 i_2 \ldots i_N}$$

with the row and column indices

$$u = i_n, \quad v = 1 + \sum_{k=1, k\neq n}^{N} \left((i_k - 1) \prod_{j=1, j\neq n}^{k-1} I_j \right).$$

It will be helpful to illustrate the matricization of 3rd order tensors using the following $2 \times 2 \times 2$ tensor.

$\mathcal{A} =$



$$A_{(1)} = \begin{bmatrix} a & c & e & g \\ b & d & f & h \end{bmatrix}$$

$$A_{(2)} = \begin{bmatrix} a & b & e & f \\ c & d & g & h \end{bmatrix}$$

$$A_{(3)} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}$$

Definition 4.10.2 (n-mode product) *Let tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and matrix $U \in \mathbb{R}^{J \times I_n}$. Then, the n-mode tensor-matrix product, $\mathcal{B} = \mathcal{A} \times_n U$ is defined as*

$$\mathcal{B}_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} \mathcal{A}_{i_1,\ldots,i_N} U_{j,i_n},$$

where $\mathcal{B}$ is a tensor of size $I_1 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N$.

Definition 4.10.3 (Scalar product) *The scalar product $\langle \mathcal{A}, \mathcal{B} \rangle$ of two tensors $\mathcal{A}, \mathcal{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ is defined as*

$$\langle \mathcal{A}, \mathcal{B} \rangle = \sum_{i_1} \sum_{i_2} \cdots \sum_{i_N} \mathcal{B}_{i_1, i_2, \ldots, i_N} \mathcal{A}_{i_1, i_2, \ldots, i_N}. \tag{4.42}$$

Definition 4.10.4 (Frobenius norm) *The Frobenius norm of a tensor $\mathcal{A}$ is defined as*

$$\|\mathcal{A}\| = \langle \mathcal{A}, \mathcal{A} \rangle^{1/2}. \tag{4.43}$$

4.10.2 Tensor approximation by the higher-order SVD

The Higher-Order SVD (HOSVD) [146], which is also known as the Tucker decomposition, is a generalization of the matrix SVD to tensors.

Theorem 4.10.1 (HOSVD [146]) *Every N-th order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ can be written as the product*

$$\mathcal{A} = \mathcal{S} \times_1 U^{(1)} \times_2 U^{(2)} \ldots \times_N U^{(N)}, \tag{4.44}$$

such that

1. *$U^{(n)} \in \mathbb{R}^{I_n \times I_n}$, $n = 1, 2, \ldots, N$ is a unitary matrix.*
2. *$\mathcal{S}$ is a real tensor of the same dimensions as $\mathcal{A}$. Let $\mathcal{S}_{i_n=j}$ denotes the j-th slice (matrix) of mode n. We have the following properties for $\mathcal{S}$:*
   - *(a) (all-orthogonality) any two different slices of the same mode are orthogonal, i.e., we have*

   $$\langle \mathcal{S}_{i_n=j}, \mathcal{S}_{i_n=k} \rangle = 0, \quad \textit{when} \quad j \neq k \tag{4.45}$$

   - *(b) ordering*

   $$\|\mathcal{S}_{i_n=1}\| \geq \|\mathcal{S}_{i_n=2}\| \geq \ldots \|\mathcal{S}_{i_n=I_n}\| \geq 0 \tag{4.46}$$

   for $n = 1, 2, \ldots, N$,

   where the scalar product in (4.45) and the norm in (4.46) were defined in (4.42) and (4.43) respectively.

3. *The Frobenius norm* $\|\mathcal{S}_{i_n=i}\| = \sigma_i^{(n)}$, *where* $\sigma_i^{(n)}$ *is the i-th singular value of the mode-n matricization of* $\mathcal{A}$, *namely* $A_{(n)}$, *and the i-th column of* $U^{(n)}$ *is the corresponding left singular vector.*

Theorem 4.10.1 actually indicates how to compute the HOSVD of $\mathcal{A}$ via the SVD of the matricizations of $\mathcal{A}$:

1. Compute the SVD

$$A_{(n)} = U^{(n)}\Sigma^{(n)}\left(V^{(n)}\right)^T, \quad \text{for} \quad n = 1, 2, \ldots, N \tag{4.47}$$

2. Compute the tensor $\mathcal{S}$ by

$$\mathcal{S} = \mathcal{A} \times_1 \left(U^{(1)}\right)^T \times_2 \left(U^{(2)}\right)^T \ldots \times_N \left(U^{(N)}\right)^T.$$

Note that in the SVD (4.47), we do not need to compute matrix $V^{(n)}$ associated with the right singular vectors. The tensor approximation by the HOSVD is analogous to the matrix approximation via the SVD. Let the HOSVD of tensor $\mathcal{A}$ be given as in (4.44). Then, an approximation of $\mathcal{A}$ can be obtained by

$$\mathcal{A} \approx \mathcal{B} = \mathcal{G} \times_1 W^{(1)} \times_2 W^{(2)} \ldots \times_N W^{(N)}, \tag{4.48}$$

where $W^{(n)} \in \mathbb{R}^{I_n \times k_n}$ has the first k_n columns of $U^{(n)} \in \mathbb{R}^{I_n \times I_n}$, for $n = 1, 2, \ldots, N$ and the core tensor $\mathcal{G}$ is the leading $k_1 \times k_2 \ldots \times k_N$ subtensor of $\mathcal{S}$. In other words, we have $\mathcal{G} = \mathcal{S}(1 : k_1, \ldots, 1 : k_N)$ with the MatLab notation. Unlike the matrix approximation via the SVD, the tensor approximation in general is not optimal under the given rank constrains [146]. Nevertheless, when $\sigma_{k_n}^{(n)} \gg \sigma_{k_n+1}^{(n)}$, according to the ordering property in Theorem 4.10.1, approximation (4.48) can still be a good approximation of $\mathcal{A}$, where the approximation error $\|\mathcal{A} - \mathcal{B}\|$ can be determined by examining the small singular values discarded, see, [146, Property 10] for details.

4.10.3 HOSVD of the tensor representation of U

Consider the 2D model problem on an $n_x \times n_y$ grid. We rearrange the entries of the matrix $U \in \mathbb{R}^{n_x n_y \times k}$, used for the low-rank approximation $B^{-1}E \approx UV^T$, into a 3rd order tensor $\mathcal{U} \in \mathbb{R}^{k \times n_x \times n_y}$ that satisfies $U^T = U_{(1)}$, where $U_{(1)}$ denotes the mode-1 matricization of $\mathcal{U}$. It follows that the i-th column of U, denoted by u_i, for $i = 1, \cdots, k$, will correspond to the i-th horizontal (mode-1) slice of $\mathcal{U}$, i.e., $\mathcal{U}_{i,:,:} \in \mathbb{R}^{n_x \times n_y}$, in a way such that $\operatorname{vec}(\mathcal{U}_{i,:,:}) = u_i$, where $\operatorname{vec}(\cdot)$ denotes the vectorization of a matrix formed by stacking the columns into a column vector. Assume that the HOSVD of $\mathcal{U}$ reads

$$\mathcal{U} = \mathcal{S} \times_1 Z \times_2 V \times_3 W, \tag{4.49}$$

where $\mathcal{S} \in \mathbb{R}^{k \times n_x \times n_y}$, $Z \in \mathbb{R}^{k \times k}$, $V \in \mathbb{R}^{n_x \times n_x}$ and $W \in \mathbb{R}^{n_y \times n_y}$. Then, the approximation to $\mathcal{U}$ by the truncated HOSVD is of the form

$$\mathcal{U} \approx \tilde{\mathcal{U}} = \mathcal{G} \times_1 Z_{k_1} \times_2 V_{k_2} \times_3 W_{k_3}. \tag{4.50}$$

where $Z_{k_1} \in \mathbb{R}^{k \times k_1}$, $V_{k_2} \in \mathbb{R}^{n_x \times k_2}$, $W_{k_3} \in \mathbb{R}^{n_y \times k_3}$ and $\mathcal{G} \in \mathbb{R}^{k_1 \times k_2 \times k_3}$, with $k_1 \leq k$, $k_2 \leq n_x$ and $k_3 \leq n_y$. The quality of the approximation in (4.50) can be measured by the decays of the singular values of $U_{(n)}$, $\sigma_i^{(n)}$, of mode n for $n = 1, 2, 3$. In the case of the rank-k MLR preconditioner for the 2D Laplacian on an $n_x \times n_y$ grid, we found that, at least numerically,

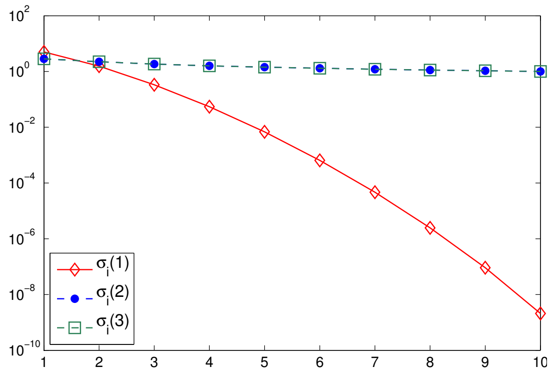
- $\sigma_i^{(n)} = 0$, for $i > k$, $n = 2, 3$, and
- $\sigma_i^{(1)} = \sigma_i^{(2)}$, for $i = 1, \ldots, k$.

In Figure 4.5, $\sigma_i^{(n)}$ for $i = 1, \ldots, 10$ and $n = 1, 2, 3$ are shown for the Laplacian on a 128×128 grid. We can see that the singular values of the third mode have a fast decay. Note that the singular values in the first mode are actually the k largest singular values of $B^{-1}E$, the expression of which has been given in (4.41). Since these singular values do not decay fast (cf. Figure 4.4) and k is typically small compared with n_x and n_y, we do not use approximations along the first and the second modes of $\mathcal{U}$. In other words, we take $k_1 = k_2 = k$. For this particular case with $k_1 = k_2 = 10$ and $k_3 = 3$, $\tilde{\mathcal{U}}$ can approximate $\mathcal{U}$ very well, as indicated by the fact that $\|U^T - \tilde{U}_{(1)}\|_2 / \|U^T\|_2 \approx 2.67\%$, while the memory required for storing $\tilde{\mathcal{U}}$ is only 1.26% of that for U.

When $k_1 = k_2 = k$, (4.50) reduces to the so-called Tucker2 decomposition in the 2nd and the 3rd modes, which reads

$$\tilde{\mathcal{U}} = \mathcal{C} \times_2 V_k \times_3 W_{k_3}, \quad \mathcal{C} = \mathcal{G} \times_1 Z_k, \ \mathcal{C} \in \mathbb{R}^{k \times k \times k_3}. \tag{4.51}$$

Figure 4.5: Singular values $\sigma_i^{(n)}$ of the mode-n matricization of $\mathcal{U} \in \mathbb{R}^{10\times 128\times 128}$, $n = 1, 2, 3$ and $i = 1, \ldots, 10$, for the Laplacian on a 128 × 128 grid. Log scale on the y-axis.



This will be the form of the tensor approximation used for the U-matrices in the MLR preconditioner that will be discussed in the next section. Two properties will be used there, which are given as follows.

Proposition 4.10.1 *Let $\mathcal{A} = \mathcal{S} \times_1 U^{(1)} \times_2 U^{(2)} \ldots \times_N U^{(N)}$ be the HOSVD. The mode-n matricization of $\mathcal{A}$ of is given by*

$$A_{(n)} = U^{(n)} S_{(n)} \left(U^{(N)} \otimes \ldots \otimes U^{(n+1)} \otimes U^{(n-1)} \otimes \ldots U^{(1)} \right)^T, \tag{4.52}$$

where $\otimes$ denotes the Kronecker product.

See [150] for the proof.

Proposition 4.10.2 *Let $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times q}$ and $X \in \mathbb{R}^{n\times p}$.*

$$\left(B^T \otimes A\right) \text{vec}(X) = \text{vec}(AXB). \tag{4.53}$$

4.10.4 MLR preconditioners with tensor approximations

Let $\mathcal{U}$ be the tensor representation of the matrix U_k in (4.10) such that $U_k^T = U_{(1)}$, which leads to the MLR preconditioner of the form $B^{-1} + U_{(1)}^T H_k U_{(1)}$ with U_k^T replaced with $U_{(1)}$. When $\mathcal{U}$ is approximated by $\tilde{\mathcal{U}}$ as obtained from the HOSVD of the form (4.51), the resulting MLR preconditioner with tensor approximations is given by

$$M^{-1} = B^{-1} + \tilde{U}_{(1)}^T H_k \tilde{U}_{(1)}, \tag{4.54}$$

where, from Proposition 4.10.1, we have

$$\tilde{U}_{(1)} = C_{(1)} \left(W_{k_3} \otimes V_k \right)^T .$$

Applying (4.54) to a vector x needs the matrix-vector products with $\tilde{U}_{(1)}$ and $\tilde{U}_{(1)}^T$. Let $X \in \mathbb{R}^{n_x \times n_y}$ such that $\operatorname{vec}(X) = x$. The matrix-vector product with $\tilde{U}_{(1)}$ is given by

$$\begin{aligned} y = \tilde{U}_{(1)} x = C_{(1)} \left(W_{k_3} \otimes V_k \right)^T x &= C_{(1)} \left(W_{k_3}^T \otimes V_k^T \right) \operatorname{vec}(X) \\ &= C_{(1)} \operatorname{vec} \left(V_k^T X W_{k_3} \right) . \end{aligned} \tag{4.55}$$

For the matrix-vector product with $\tilde{U}_{(1)}^T$, we have

$$\begin{aligned} z = \tilde{U}_{(1)}^T x = \left(W_{k_3} \otimes V_k \right) C_{(1)}^T x &= \left(W_{k_3} \otimes V_k \right) q = \left(W_{k_3} \otimes V_k \right) \operatorname{vec}(Q) \\ &= \operatorname{vec} \left(V_k Q W_{k_3}^T \right), \quad \text{with} \quad q = C_{(1)}^T x = \operatorname{vec}(Q),\ Q \in \mathbb{R}^{k \times k_3}. \end{aligned} \tag{4.56}$$

Next, we analyze the computational and memory cost of the preconditioner (4.54). The memory cost of $\tilde{\mathcal{U}}$ is of the order $\mathcal{O}(k^2 k_3 + k n_x + k_3 n_y)$, where the first term is for storing the core tensor $\mathcal{C}$ and the second and the third terms are for the factor matrices, V_k and W_{k_3}. Compared with the memory requirement of U_k, which is $\mathcal{O}(k n_x n_y)$, when $n_y \gg 1$ and $k n_x \gg k_3$, significant reduction of the memory cost can be obtained. The computational costs of (4.55) and (4.56) are $\mathcal{O}\left(k_3 n_x n_y + k k_3 n_x + k^2 k_3\right)$ and $\mathcal{O}\left(k n_x n_y + k k_3 n_y + k^2 k_3\right)$ respectively. When k and k_3 are small, these costs are close to that of the matrix-vector product with U_k, which is $\mathcal{O}(k n_x n_y)$.

The extension to the 3-D case, where the above scheme can be carried out similarly with 4th order tensors, is straightforward.

4.11 Numerical experiments

The experiments were conducted on a machine equipped with an Intel Xeon X5675 processor (12 MB Cache, 3.06 GHz, 6-core) and 96 GiB of main memory. The implementation of the MLR preconditioner was written in C/C++ and compiled by g++

using the -O3 optimization level. BLAS and LAPACK routines from Intel Math Kernel Library (MKL, version 10.2) were used to enhance the performance on multiple cores.

We first report on the results of solving symmetric linear systems from a 2-D and a 3-D partial differential equation (PDE) on regular grids using MLR preconditioners combined with Krylov subspace methods. For these problems, a recursive geometric bisection is used for the DD in the MLR preconditioner. Specifically, a 2-D regular grid is cut in half always along the shorter side, while a 3-D regular grid is cut in half along the face of the smallest area. Next, MLR preconditioners were tested for solving a sequence of general symmetric linear systems. For these problems, a graph partitioning algorithm `PartGraphRecursive` from METIS [76] was used for the DD.

Three types of preconditioners were compared in our experiments: 1) the MLR preconditioner, 2) incomplete Cholesky factorization with threshold dropping (ICT), for the SPD cases and 3) incomplete LDL factorization with threshold dropping (ILDLT) for the symmetric indefinite cases. The accelerators considered were the CG method for the SPD cases and the GMRES method with a restart dimension of 40 for the symmetric indefinite cases. For all the cases, iterations were stopped whenever the residual norm had been reduced by 8 orders of magnitude or the maximum number of iterations allowed, which is 500, was exceeded. The results are summarized in Tables 4.2-4.7 where all the CPU times are reported in seconds. When comparing preconditioners, the following factors are considered: 1) fill-ratio, 2) time for building preconditioners, 3) the number of iterations and 4) time for the iterations. In all tables, '`F`' indicates non-convergence within the maximum allowed number of steps.

In the Lanczos bidiagonalization method, the convergence is checked every 10 iterations and the tolerance ϵ used for checking the convergence in (4.15) is set to 10^{-3}. In addition, we set the maximum number of Lanczos steps as 10 times the number of requested singular values. At the last level of the MLR preconditioner, we preorder the matrices by the approximate minimum degree (AMD) ordering [2,110,111] to reduce fill-ins. In addition, when using an ICT or ILDLT factorization as a global preconditioner, we also preorder the matrix by the AMD ordering.

4.11.1 2-D/3-D model problems

We will examine a 2-D PDE problem,

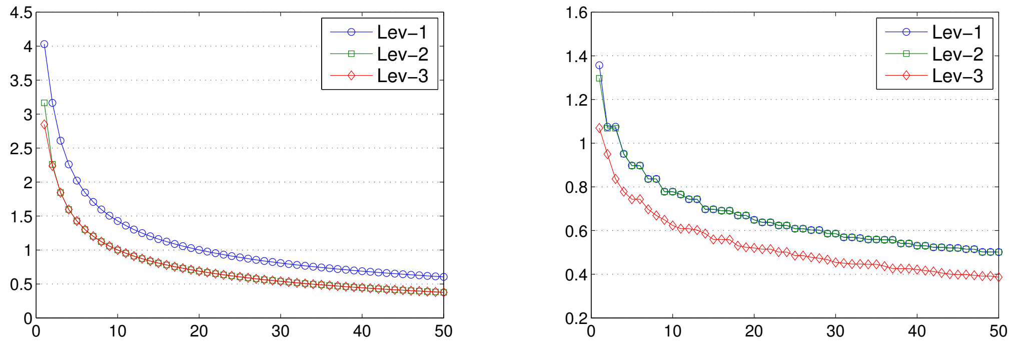
$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - cu = -\left(x^2 + y^2 + c\right) e^{xy} \text{ in } \Omega,$$
$$u = e^{xy} \text{ on } \partial\Omega, \tag{4.57}$$

and a 3-D one,

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} - cu = -6 - c\left(x^2 + y^2 + z^2\right) \text{ in } \Omega,$$
$$u = x^2 + y^2 + z^2 \text{ on } \partial\Omega. \tag{4.58}$$

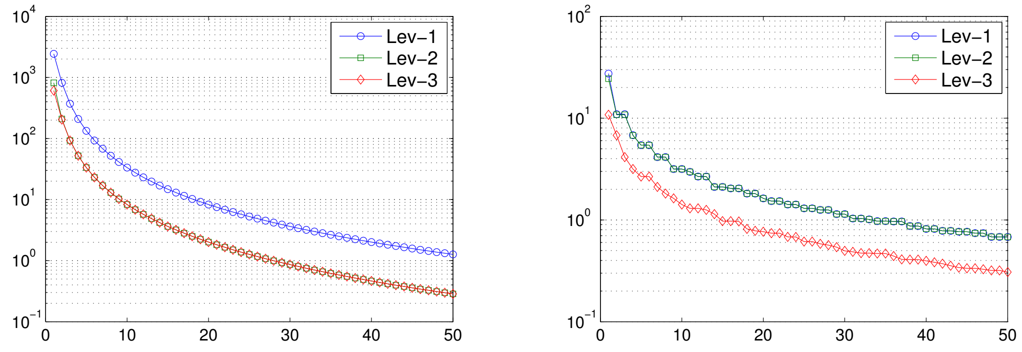
The domain is $\Omega = (0,1) \times (0,1)$ for the 2-D problem and for the 3-D problem, is $\Omega = (0,1) \times (0,1) \times (0,1)$, while $\partial\Omega$ is the boundary where we use we use the Dirichlet boundary condition. The exact solutions of (4.57) and (4.58) are $u = e^{xy}$ and $u = x^2+y^2+z^2$ respectively. We take the 5-point (7-point) centered difference approximation on regular 2-D (3-D) grids.

Figure 4.6: Illustration of the decay of singular values of Q_i for $i = 0, 1, 3$, for a 2-D model problem (left) and a 3-D model problem (right).



To begin with, we examine the singular values of Q_i and $\tilde{Q}_i$ and the eigenvalues of $B_i^{-1}E_iX_i^{-1}E_i^TB_i^{-1}$ for the 2-D and 3-D model problems on a 256×256 grid and a $32 \times 32 \times 64$ grid, where Q_i and $\tilde{Q}_i$ were defined in (4.14) and (4.16). For both problems, we set $c = 0$ in (4.57) and (4.58). The first 50 singular values of Q_i and the first 50 eigenvalues of $B_i^{-1}E_iX_i^{-1}E_i^TB_i^{-1}$, for $i = 0, 1, 3$, from the first three levels of the MLR

Figure 4.7: Illustration of the decay of eigenvalues of $B_i^{-1}E_iX^{-1}E_i^TB_i^{-1}$ (log scale) for $i = 0, 1, 3$, for a 2-D model problem (left) and a 3-D model problem (right).



preconditioner are depicted in Figure 4.6 and Figure 4.7 (Note log scale on the y-axis in Figure 4.7). As shown, compared with the singular values of Q_i, the eigenvalues of $B_i^{-1}E_iX_i^{-1}E_i^TB_i^{-1}$ decay much more rapidly due to the damping effect of X_i^{-1}.

Recall that when building an MLR preconditioner, instead of Q_i, we compute approximate singular values and vectors of a nearby matrix $\tilde{Q}_i$ that is defined in (4.16). In Table 4.1, we compare the singular values of $\tilde{Q}_i$ and those of Q_i at the first 3 levels of the MLR preconditioners for 2-D and 3-D Laplacian matrices with 5 levels and 3 ranks. We tabulate the 8 largest singular values of Q_i and the k largest singular values of $\tilde{Q}_i$ for $i = 0, 1, 3$. From the results in Table 4.1, we can make several observations:

1. For a certain Q_i and rank k, smaller singular values of $\tilde{Q}_i$ approximate those corresponding ones of Q_i better;

2. For a certain Q_i, among different ranks, singular values of $\tilde{Q}_i$ with a higher rank k approximate those corresponding ones of Q_i better;

3. For different Q_i's, singular values of $\tilde{Q}_i$ at a lower level approximate those corresponding ones of Q_i better.

Table 4.1: Approximations to the singular values of Q_i, $i = 0, 1, 3$, at the first 3 levels of MLR preconditioners for the 2-D and 3-D model problems.

		rank	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8
2-D	Q_0	-	4.029	3.166	2.609	2.260	2.021	1.845	1.707	1.596
		k=2	2.873	2.605	-	-	-	-	-	-
	$\tilde{Q}_0$	k=4	3.381	2.926	2.507	2.260	-	-	-	-
		k=8	3.494	2.996	2.555	2.260	1.987	1.813	1.679	1.596
	Q_1	-	3.166	2.260	1.845	1.596	1.427	1.301	1.203	1.123
		k=2	2.847	2.260	-	-	-	-	-	-
	$\tilde{Q}_1$	k=4	2.931	2.260	1.813	1.596	-	-	-	-
		k=8	2.987	2.260	1.834	1.596	1.415	1.301	1.191	1.123
	Q_3	-	2.849	2.238	1.843	1.596	1.427	1.301	1.203	1.123
		k=2	2.777	2.238	-	-	-	-	-	-
	$\tilde{Q}_3$	k=4	2.824	2.238	1.813	1.596	-	-	-	-
		k=8	2.841	2.238	1.833	1.596	1.415	1.300	1.191	1.123
3-D	Q_0	-	1.357	1.075	1.075	0.951	0.898	0.898	0.836	0.836
		k=2	1.130	1.012	-	-	-	-	-	-
	$\tilde{Q}_0$	k=4	1.161	1.034	1.012	0.951	-	-	-	-
		k=8	1.193	1.053	1.034	0.951	0.839	0.824	0.812	0.799
	Q_1	-	1.297	1.069	1.069	0.950	0.897	0.897	0.836	0.836
		k=2	1.114	1.007	-	-	-	-	-	-
	$\tilde{Q}_1$	k=4	1.167	1.024	1.003	0.951	-	-	-	-
		k=8	1.184	1.045	1.017	0.950	0.840	0.824	0.811	0.788
	Q_3	-	1.069	0.950	0.836	0.778	0.743	0.743	0.697	0.669
		k=2	1.021	0.950	-	-	-	-	-	-
	$\tilde{Q}_3$	k=4	1.026	0.950	0.797	0.749	-	-	-	-
		k=8	1.039	0.950	0.805	0.760	0.743	0.673	0.642	0.610

We now report on the performance of the MLR preconditioner for solving the 2-D and 3-D model problems (4.57) and (4.58). First, we set $c = 0$, so that the coefficient matrices are SPD. Recall from Corollary 4.9.1 in Section 4.9, that if we have $\rho(U_i^T E V_i) < 1$ for all i, for an SPD matrix the MLR preconditioner (4.13) is also SPD. Therefore, for these cases, we can use the MLR preconditioner along with the CG method. Numerical experiments were carried out to compare the performance of the MLR preconditioner and the ICT preconditioner. We solved each of the problems on 3 different grids. The size of each grid and the order of the matrix (N) are shown in Table 4.2. The fill-ratios (fill), the numbers of iterations (its), the time for building preconditioners (p-t) and iterations (i-t) for both preconditioners are also tabulated. In these experiments, the fill-ratios of both preconditioners were controlled to be roughly equal. For the 2-D problems, the number of levels of MLR preconditioners was fixed at 5. On the other hand, for the 3-D problems, we increased the number of levels by one as the size of the grid doubles. In this way, the sizes of the matrices at the last level in the MLR preconditioner were kept about the same for the 3 grids. Specifically, we set the number of levels to 5 for the first problem and used 7 and 10 levels for the other two problems respectively. For all the problems, the rank used in the MLR preconditioner is 2.

Table 4.2: Comparison between ICT preconditioners and MLR preconditioners for solving the SPD linear systems along with CG.

Grid	N	ICT-CG				MLR-CG			
		fill	p-t	its	i-t	fill	p-t	its	i-t
256×256	$65,536$	3.1	0.08	69	0.19	3.2	0.45	84	0.12
512×512	$262,144$	3.2	0.32	133	1.61	3.5	1.57	132	1.06
1024×1024	$1,048,576$	3.4	1.40	238	15.11	3.5	4.66	215	9.77
$32 \times 32 \times 64$	65,536	2.9	0.14	33	0.10	3.0	0.46	43	0.08
$64 \times 64 \times 64$	262,144	3.0	0.66	47	0.71	3.1	3.03	69	0.63
$128 \times 128 \times 128$	2,097,152	3.0	6.59	89	13.47	3.2	24.61	108	10.27

The results in Table 4.2 indicate that compared with the ICT preconditioner, building an MLR preconditioner requires more CPU time, 4 times more on average in this set of experiments. For the 2-D problems, the MLR-CG method achieved convergence in slightly fewer iterations than the ICT-CG method, whereas for the 3-D problems, it required more iterations. For all the cases, we observed performance gains when using the MLR preconditioner in terms of reduced iteration time. The CPU time for building an MLR preconditioner is typically dominated by the cost of matrix-vector operations in (4.17) and (4.18) when performing the Lanczos algorithm on $\tilde{Q}_i$. Furthermore, this cost is actually governed by the cost of the triangular solves at the last level. Different layers of parallelism exist in building an MLR preconditioner. For instance, in a matrix-vector operation of each Q_i, recursive calls in lines 6 and 7 of Algorithm 6 are independent.

Moreover the Lanczos algorithm can be performed on all $\tilde{Q}_i$'s within the same level in parallel. These features have not yet been implemented in our current code.

Table 4.3: Performance of solving the 2-D SPD problem by the MLR preconditioner with different ranks and numbers of levels along with CG.

rank	nlev	fill	mem.lrk		mem.fact		p-t	its	i-t
2	5	3.5	45%	(8.4m)	55%	(10.1m)	4.66	215	9.77
2	6	3.9	52%	(10.5m)	48%	(9.7m)	5.40	251	11.06
2	7	4.2	58%	(12.6m)	42%	(9.3m)	5.80	230	9.86
2	8	4.5	63%	(14.7m)	37%	(8.8m)	6.65	254	10.89
2	9	4.8	67%	(16.8m)	33%	(8.4m)	7.26	250	11.20
3	5	4.3	56%	(12.6m)	44%	(10.1m)	6.23	192	10.00
4	5	5.1	62%	(16.8m)	38%	(10.1m)	7.60	185	10.11
5	5	5.9	68%	(21.0m)	32%	(10.1m)	9.81	168	11.06
6	5	6.7	71%	(25.2m)	29%	(10.1m)	12.13	167	11.98

In the next set of experiments, we examined the behavior of MLR preconditioners with different ranks and numbers of levels. We solved the 2-D problem on a 1024×1024 regular grid. We present the results in Table 4.3, where 'mem.lrk' stands for the memory requirement for storing the matrices for the low-rank approximations and 'mem.fact' indicates the memory requirement for storing the factors at the last level. For each case, the percentage and the number of nonzeros (measured in millions) are reported. Here are a few observations from Table 4.3. First, when we fixed the rank $k = 2$ and increased the number of levels, the fill-ratio grew and the MLR preconditioner required more CPU time to build. Note that the number of nonzeros in the factors decreased since the orders of the matrices at the last level became smaller. In addition, more iterations were required for convergence and the total iteration time increased as well. However, the CPU time cost per iteration was almost the same. Second, we fixed the number of levels to $nlev = 5$ and increased the rank. In this case, the fill-ratio and the time for building the preconditioner grew as before, whereas the number of iterations dropped. In spite of the fewer iterations required, the total iteration time still increased with the rank since performing an iteration was more expensive.

Next, we solved the indefinite problems by setting $c > 0$ in (4.57) and (4.58), which shifts the discretized negative Laplacian (a positive definite matrix) by subtracting the matrix sI for a certain s. In this set of experiments, we solved the 2-D problems with $s = .01$ and the 3-D problems with $s = .05$. The MLR preconditioner was compared with the ILDLT preconditioner along with the GMRES(40) method. For the 2-D problems, the number of levels was set to 4, whereas for the 3-D problems, this number was 5. Moreover, for both sets of 2-D and 3-D problems, we used the MLR preconditioner of rank 5 for the first problem, but a rank of 7 for the other two. Results are shown in Table 4.4. For most problems, the ILDLT-GMRES method failed even though higher fill-ratios were used compared with the SPD cases. In contrast, the MLR preconditioner appeared to be more effective. Except for the two largest cases, the MLR-GMRES method achieved convergence and great savings in CPU time.

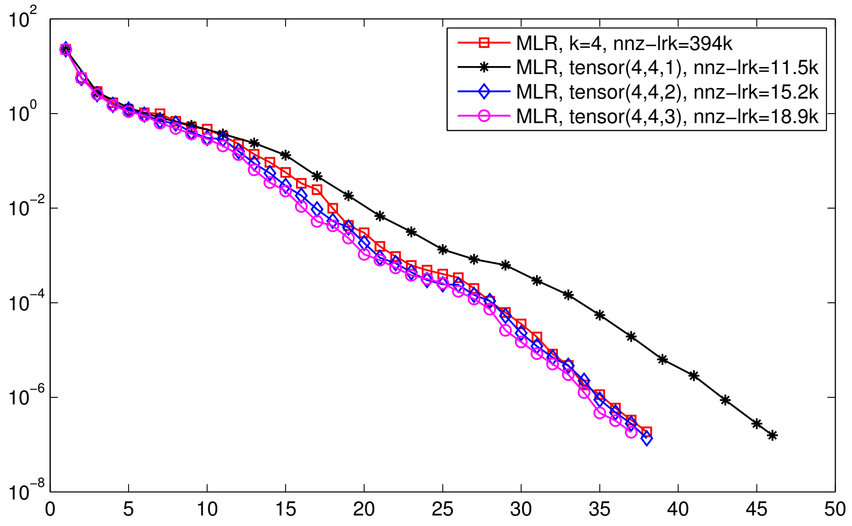
A few difficulties were encountered for large indefinite problems. Typically, an MLR preconditioner with many levels (e.g., 7 or 8) will not lead to convergence for these problems and this makes matrices at the last level still quite large. As a result, factoring these matrices will be inefficient in terms of both CPU time and memory requirement. Furthermore, approximations with higher ranks were required compared with those for the SPD cases. This increased the memory requirement, the CPU time of building the MLR preconditioner, and also the computational cost of applying the preconditioner.

Table 4.4: Comparison between the ILDLT and the MLR preconditioners for solving the symmetric indefinite linear systems with GMRES(40).

Grid	ILDLT-GMRES				MLR-GMRES			
	fill	p-t	its	i-t	fill	p-t	its	i-t
256×256	6.5	0.16	F	-	6.0	0.39	84	0.30
512×512	8.4	1.25	F	-	8.2	2.24	246	6.03
1024×1024	10.3	10.09	F	-	9.0	15.05	F	-
$32 \times 32 \times 64$	5.6	0.25	61	0.38	5.4	0.98	62	0.22
$64 \times 64 \times 64$	7.0	1.33	F	-	6.6	6.43	224	5.43
$128 \times 128 \times 128$	8.8	15.35	F	-	6.5	28.08	F	-

Finally, for the 2-D model problem on a 128×128 grid, we show the performance of using 3-order tensors to compress the U-matrices as described in Section 4.10. The convergence profile is shown in Figure 4.8, where the standard MLR method with rank $k = 4$ was compared with the ones which used 3-order tensors to represent U_i's. Note that we used the rank k in the first and the second modes of the tensor, whereas in mode 3, we changed the rank from 1 to 3. The legend of this figure shows the ranks used in the HOSVD and also the numbers of the elements stored for the low-rank approximations (measured in thousands). In the standard rank-4 MLR preconditioner, this number is about 394 thousand. In contrast, the MLR-tensor method with ranks $(4, 4, 3)$ only required 18.9 thousand entries and achieved convergence with one iteration fewer, and the one with ranks $(4, 4, 2)$ reduced this number to 15.2 thousand and had almost the same convergence rate. For even lower ranks, although the memory usage was further reduced, the number of iterations increased significantly.

Figure 4.8: Convergence history and memory usage of the MLR preconditioner and the MLR preconditioner with tensor representation for the 2-D model problem.



4.11.2 General matrices

We selected 10 matrices from the University of Florida sparse matrix collection [26] for the following tests. Among these matrices 7 are SPD and 3 are symmetric indefinite. Table 4.5 lists the name, the order (N), the number of nonzeros (NNZ), the positive definiteness, and a short description of each matrix. If the actual right-hand-side is not provided, the linear system was obtained by creating an artificial one as $b = Ae$, where

e is the vector of all ones. The results are shown in Table 4.6-4.7, where we include the CPU time for performing the DD using the graph partitioner.

Table 4.5: Names, orders (N), numbers of nonzeros (NNZ), positive definiteness and short descriptions of the test matrices.

Matrix	N	NNZ	SPD	Description
Andrews/Andrews	60,000	760,154	yes	computer graphics problem
Williams/cant	62,451	4,007,383	yes	FEM cantilever
UTEP/Dubcova2	65,025	1,030,225	yes	2-D/3-D PDE problem
Rothberg/cfd1	70,656	1,825,580	yes	CFD problem
Schmid/thermal1	82,654	574,458	yes	thermal problem
Rothberg/cfd2	123,440	3,085,406	yes	CFD problem
Schmid/thermal2	1,228,045	8,580,313	yes	thermal problem
Cote/vibrobox	12,328	301,700	no	vibroacoustic problem
Cunningham/qa8fk	66,127	1,660,579	no	3-D acoustics problem
Koutsovasilis/F2	71,505	5,294,285	no	structural problem

In Section 4.7, we discussed three balancing options: the first one with a scalar, the second one with a diagonal matrix and the third one with an LU factorization. The third option did not yield results as good as with the other two and the related experimental results are omitted. For the first option, the scalar α defined in (4.22) is estimated by a power iteration. Table 4.6 shows the performance of the MLR preconditioner with the first balancing method, along with the number of levels and the rank used for each problem. The MLR-GMRES method achieved convergence for all cases, whereas for many cases, it failed to converge with the ICT or ILDLT preconditioner. Similar to the experiments for the model problems, the MLR preconditioner required more CPU time to build than the ILU preconditioners but achieved significant CPU time savings in the iteration phase. We note here that for two problems (`cant` and `vibrobox`), a higher rank was required than with the other problems. This might be due to the fact that in these problems the ratio of the number of interface nodes to the number of the total nodes is higher than for the others, which results in the need of higher rank approximations to reach enough accuracy. Experimental results using smaller ranks did not lead to convergence within the maximum number of iterations for these two problems.

The last experiment we carried out was to test the MLR preconditioner with the

Table 4.6: Results of the ICT/ILDLT and the MLR preconditioners for solving general symmetric linear systems with CG or GMRES(40). Scalar balancing in MLR.

Matrix	ICT/ILDLT				MLR					
	fill	p-t	its	i-t	rank	nlev	fill	p-t	its	i-t
Andrews	2.6	0.44	32	0.16	2	6	2.3	1.32	29	0.08
cant	4.3	2.47	F	-	10	5	4.3	8.09	380	7.56
Dubcova2	1.4	0.14	42	0.21	4	4	1.4	0.65	46	0.10
cfd1	2.8	0.56	314	3.42	5	5	2.5	3.65	322	1.92
thermal1	3.1	0.15	108	0.51	2	5	3.1	0.75	110	0.32
cfd2	3.6	1.14	F	12.27	5	4	3.1	6.52	396	6.57
thermal2	5.3	4.11	148	20.45	5	5	5.3	15.42	175	14.63
vibrobox	3.3	0.19	F	-	10	4	2.9	0.50	259	0.32
qa8fk	1.8	0.58	56	0.60	2	8	1.6	2.53	61	0.29
F2	2.3	1.37	F	-	5	5	2.4	4.77	289	5.68

second balancing method for the same set of matrices. The results are reported in Table 4.7. The ranks and the numbers of levels used here are the same as the ones displayed in Table 4.6. Compared with the MLR preconditioners with the first balancing method, these preconditioners required almost the same amount of memory and similar CPU time to build. However, in six out of the ten cases this option led to fewer steps to converge, significantly so (at least 21% fewer) in four of the cases.

Table 4.7: Results of the ICT/ILDLT and the MLR preconditioners for solving general symmetric linear systems with CG or GMRES(40). Diagonal matrix balancing for MLR.

Matrix	rank	nlev	fill	p-t	its	i-t
Andrews	2	6	2.3	1.38	27	0.08
cant	10	5	4.3	7.89	253	5.30
Dubcova2	4	4	1.5	0.60	47	0.09
cfd1	5	5	2.3	3.61	244	1.45
thermal1	2	5	3.2	0.69	109	0.33
cfd2	5	4	3.1	4.70	312	4.70
thermal2	5	5	5.4	15.15	178	14.96
vibrobox	10	4	3.0	0.45	183	0.22
qa8fk	2	8	1.6	2.33	75	0.36
F2	5	5	2.5	4.17	371	7.29

4.12 Summary and discussion

This chapter has presented a preconditioning method for solving symmetric sparse linear systems, based on a recursive multilevel low-rank approximation approach. It is useful to compare the potential advantages and disadvantages of an approach of this type relative to those of the traditional ILU-type preconditioners. These advantages and disadvantages are somewhat shared with the other two preconditioning methods of this type to be introduced in the next two chapters, which are also based on low-rank approximation techniques. So we summarize them here.

On the negative side, first, the divide-and-conquer algorithm used in this method may lead to some obvious difficulties in the implementation. Exploiting recursivity as much as possible can simplify the programming complexity. On the other hand, this method appears to be simpler than those based on hierarchical matrices [129–133]. We note here that the other two methods of this type discussed in the next two chapters will adopt a more general domain decomposition (DD) framework, so that they are much easier to implement. Second, building the low-rank correction based preconditioners can be time consuming, although several mitigating factors should be taken into account. First, we note that the proposed method is especially suitable for massively parallel machines, such as those equipped with the GPUs or with the Intel Xeon Phi processors, since the computations required for building and for applying the preconditioners can be easily vectorized. As such, the set-up phase is likely to be far more advantageous than that in a factorization-based method that tends to be much more sequential. The second is that there are situations in which many systems with the same matrix but different right-hand sides must be solved, in which case more expensive but more effective preconditioners may be justified as their cost will be amortized.

The potential advantages of the proposed low-rank correction based preconditioner outnumber the disadvantages. First, this type of preconditioners is very suitable for single-instruction-multiple-data (SIMD) parallelism as when using GPUs. In addition, the DD framework lends itself easily to a macro-level parallelism. Thus, one can easily imagine implementing these approaches on a multiprocessor system based on a multi(many)-core architecture exploiting the two levels of parallelism.

A second appeal is that this type of methods appears to be more robust than the

ILU-based methods for indefinite problems, since the low-rank correction based preconditioner is essentially a form of approximate inverse technique and as such it is not prone to the difficulties seen with the ILU-type preconditioners. Therefore, what matters is how well we approximate the inverses of the matrices at the lowest level, and how good is the low rank approximation used. This could lead to an analysis regarding the quality of the preconditioner, independently of the spectrum of the original matrix.

A third appeal, shared by the approximate inverse-type preconditioners, is that the preconditioner is "updatable" in the sense that if one is not satisfied with the performance on a particular problem, the accuracy of the preconditioner can be improved without foregoing work previously done. The heart of this type of methods consists of obtaining a low-rank approximation to a certain matrix. Improving this approximation would consist in merely adding a few more vectors and this can be easily achieved in a number of ways, e.g., by resorting to a form of deflation (see Section 4.8), without having to throw away the vectors already computed.

Chapter 5

Low-rank correction methods for algebraic domain decomposition preconditioners

5.1 Introduction

Preconditioning distributed sparse linear systems remains a challenging problem in high-performance multi-processor environments. Simple domain decomposition (DD) algorithms such as the additive Schwarz method [80–83, 151] are widely used and they usually yield good parallelism. A well-known problem with these preconditioners is that they often require a large number of iterations when the number of domains used is large. As a result, the benefits of increased parallelism is often outweighed by the increased number of iterations. Algebraic multigrid (AMG) methods have achieved a good success and can be extremely fast when they work. However, their success is still somewhat restricted to certain types of problems. Methods based on the Schur complement technique such as the ones discussed in Section 2.4.4, which consist of eliminating interior unknowns first and then focus on solving in some ways the interface unknowns in the reduced system, are designed to be general-purpose. The difficulty in this type of methods is to find effective and efficient preconditioners for the distributed global reduced system.

5.2 Domain decomposition with local corrections

In this chapter, we present a preconditioning method for *symmetric* matrices [152], which extends the idea of the Multilevel Low-Rank (MLR) preconditioner described in Chapter 4 to the general framework of distributed sparse linear systems via the standard DD approach. The DD method used in this preconditioner is based on the *vertex-based* partitioning, so that the reordered global linear system has the form

$$\begin{pmatrix} B_1 & & & & \hat{E}_1 \\ & B_2 & & & \hat{E}_2 \\ & & \ddots & & \vdots \\ & & & B_p & \hat{E}_p \\ \hat{E}_1^T & \hat{E}_2^T & \dots & \hat{E}_p^T & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_p \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_p \\ g \end{pmatrix}, \tag{5.1}$$

or a more compact form,

$$\begin{pmatrix} B & \hat{E} \\ \hat{E}^T & C \end{pmatrix} \begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{5.2}$$

with $B \in \mathbb{R}^{m\times m}$, $\hat{E} \in \mathbb{R}^{m\times s}$, $C \in \mathbb{R}^{s\times s}$ and $n = m + s$. See (2.23) and (2.24) for the block structures of the global Schur complement and matrix C, and the left part of Figure 2.5 for an illustration for the structure of the coefficient matrix. Writing the system in the local forms

$$\begin{pmatrix} B_i & E_i \\ E_i^T & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j\in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix}, \quad i = 1, \dots, p, \tag{5.3}$$

is commonly adopted in practice when solving distributed sparse linear systems, while (5.1) will be more convenient for analysis. In what follows, we will assume that the global matrix is put in the form as in (5.1), while (5.3) will return in Section 5.3 that deals with the parallel implementations.

A major difference in this method compared with the Schur complement based methods is that we do not try to solve the reduced system or even form the global Schur complement. Instead, an approximate inverse preconditioner to the original matrix is obtained by exploiting a low-rank property and the Sherman-Morrison-Woodbury (SMW) formula. We refer to a preconditioner obtained by this approach as a DD based Low-Rank (DDLR) preconditioner.

5.2.1 Splitting

We begin by splitting the coefficient matrix as

$$A = \begin{pmatrix} B & \hat{E} \\ \hat{E}^T & C \end{pmatrix} = \begin{pmatrix} B & \\ & C \end{pmatrix} + \begin{pmatrix} & \hat{E} \\ \hat{E}^T & \end{pmatrix}, \tag{5.4}$$

and defining the $n \times s$ matrix,

$$E \equiv \begin{pmatrix} \alpha^{-1}\hat{E} \\ -\alpha I \end{pmatrix}, \tag{5.5}$$

where I is the $s \times s$ identity matrix and α is a parameter. Then from (5.4) we immediately get the identity,

$$\begin{pmatrix} B & \hat{E} \\ \hat{E}^T & C \end{pmatrix} = \begin{pmatrix} B + \alpha^{-2}\hat{E}\hat{E}^T & \\ & C + \alpha^2 I \end{pmatrix} - EE^T$$

A remarkable property is that the operator $\hat{E}\hat{E}^T$ is *local* in that it does not involve inter-domain couplings. Specifically, we have the following proposition.

Proposition 5.2.1 *Consider the matrix $X = \hat{E}\hat{E}^T$ and its blocks X_{ij} associated with the same blocking as for the matrix in (5.1). Then, for $1 \le i, j \le p$ we have:*

$$X_{ij} = 0, \quad \text{for} \quad i \neq j$$
$$X_{ii} = E_i E_i^T.$$

Proof. This follows from the fact that the columns of $\hat{E}$ associated with different subdomains are structurally orthogonal illustrated on the left side of Figure 2.5. □

Thus, we can write

$$A = A_0 - EE^T, \quad A_0 = \begin{pmatrix} B + \alpha^{-2}\hat{E}\hat{E}^T & \\ & C + \alpha^2 I \end{pmatrix} \in \mathbb{R}^{n \times n}, \tag{5.6}$$

with the matrix E defined in (5.5). From (5.6) and the SMW formula, we can derive the expression for the inverse of A. First define,

$$G = I - E^T A_0^{-1} E. \tag{5.7}$$

Then, we have

$$A^{-1} = A_0^{-1} + A_0^{-1} E (\underbrace{I - E^T A_0^{-1} E}_{G})^{-1} E^T A_0^{-1} \equiv A_0^{-1} + A_0^{-1} E G^{-1} E^T A_0^{-1}. \tag{5.8}$$

Note that the matrix C is often *strongly diagonally dominant* for matrices arising from the discretization of PDEs, and the parameter α can serve to improve diagonal dominance in the indefinite cases.

5.2.2 Low-rank approximation to matrix G

In this section we will consider the case when A is symmetric positive definite (SPD). A preconditioner of the form

$$M^{-1} = A_0^{-1} + (A_0^{-1}E)\tilde{G}^{-1}(E^T A_0^{-1}),$$

can be readily obtained from (5.8) if we had an approximation $\tilde{G}^{-1}$ to G^{-1}. Note that the application of this preconditioner will involve two solves with A_0 instead of only one. It will also involve a solve with $\tilde{G}$ which operates on the interface unknowns. Let us, at least formally, assume that we know the spectral factorization of $E^T A_0^{-1} E$

$$H \equiv E^T A_0^{-1} E = U \Lambda U^T,$$

where $H \in \mathbb{R}^{s \times s}$, U is orthogonal, and Λ is diagonal. From (5.6), we have $A_0 = A + EE^T$, and thus A_0 is SPD since A is SPD. Therefore, H is at least symmetric positive semidefinite (SPSD) and the following lemma shows that its eigenvalues are all less than one.

Lemma 5.2.1 *Let $H = E^T A_0^{-1} E$. Assume that A is SPD and the matrix $I - H$ is nonsingular. Then we have $0 \leq \lambda < 1$, for each eigenvalue λ of H.*

Proof. From (5.8), we have

$$E^T A^{-1} E = H + H(I-H)^{-1}H = H\left(I + (I-H)^{-1}H\right) = H(I-H)^{-1}.$$

Since A is SPD, $E^T A^{-1} E$ is at least SPSD. Thus, the eigenvalues of $H(I-H)^{-1}$ are nonnegative, i.e., $\lambda/(1-\lambda) \geq 0$. So, we have $0 \leq \lambda < 1$. $\square$

The goal now is to see what happens if we replace Λ by a diagonal matrix $\tilde{\Lambda}$. This will include the situation when a low-rank approximation is used for G but it can also include other possibilities. Suppose that H is approximated as follows:

$$H \approx U \tilde{\Lambda} U^T. \tag{5.9}$$

Then, from the SMW formula, the corresponding approximation to G^{-1} is:

$$G^{-1} \approx \tilde{G}^{-1} \equiv (I - U\tilde{\Lambda}U^T)^{-1} = I + U[(I - \tilde{\Lambda})^{-1} - I]U^T. \tag{5.10}$$

Note in passing that the above expression can be simplified to $U(I - \tilde{\Lambda})^{-1}U^T$. However, we keep the above form because it will still be valid when U has only k ($k < s$) columns and $\tilde{\Lambda}$ is $k \times k$ diagonal, in which case we denote by G_k^{-1} the approximation in (5.10). At the same time, the exact G can be obtained as a special case of (5.10), where $\tilde{\Lambda}$ is simply equal to Λ. Then we have

$$A^{-1} = A_0^{-1} + (A_0^{-1}E)G^{-1}(E^TA_0^{-1}), \tag{5.11}$$

and the preconditioner

$$M^{-1} = A_0^{-1} + (A_0^{-1}E)G_k^{-1}(E^TA_0^{-1}), \tag{5.12}$$

from which it follows by subtraction that

$$A^{-1} - M^{-1} = (A_0^{-1}E)(G^{-1} - G_k^{-1})(E^TA_0^{-1}),$$

and therefore,

$$AM^{-1} = I - A(A_0^{-1}E)(G^{-1} - G_k^{-1})(E^TA_0^{-1}). \tag{5.13}$$

A first consequence of (5.13) is that there will be at lease m eigenvalues of AM^{-1} that are equal to one, where $m = n - s$ is the dimension of B in (5.2) or in other words, the number of the interior unknowns. From (5.10) we obtain

$$G^{-1} - G_k^{-1} = U[(I - \Lambda)^{-1} - (I - \tilde{\Lambda})^{-1}]U^T. \tag{5.14}$$

The simplest selection of $\tilde{\Lambda}$ is the one that ensures that the k largest eigenvalues of $(I - \tilde{\Lambda})^{-1}$ match the largest eigenvalues of $(I - \Lambda)^{-1}$. This simply minimizes the 2-norm of (5.14) under the assumption that the approximation in (5.9) is of rank k. Assume that the eigenvalues of H are $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_s$. This means that the diagonal entries $\tilde{\lambda}_i$ of $\tilde{\Lambda}$ are selected such that

$$\tilde{\lambda}_i = \begin{cases} \lambda_i & \text{if} \quad i \leq k \\ 0 & \text{otherwise} \end{cases}. \tag{5.15}$$

Observe that from (5.14) the eigenvalues of $G^{-1} - G_k^{-1}$ are

$$\left\{ \begin{array}{cl} 0 & \text{if} \quad i \le k \\ (1-\lambda_i)^{-1} - 1 & \text{otherwise} \end{array} \right. .$$

Thus, from (5.13) we can infer that k more eigenvalues of AM^{-1} will take the value one in addition to the existing m ones revealed above independently of the choice of $\tilde{G}^{-1}$. Noting that $(1-\lambda_i)^{-1} - 1 = \lambda_i/(1-\lambda_i) \ge 0$, since $0 \le \lambda_i < 1$ and we can say that the remaining $s-k$ eigenvalues of AM^{-1} will be between 0 and 1. Therefore, the result in this case is that the preconditioned matrix AM^{-1} in (5.13) will have $m+k$ eigenvalues equal to one, and $s-k$ other eigenvalues between 0 and 1.

From an implementation point of view, it is clear that a full diagonalization of H is not needed. All we need is U_k, the $s \times k$ matrix consisting of the first k columns of U, along with the diagonal matrix Λ_k of the corresponding eigenvalues $\lambda_1, \cdots, \lambda_k$. Then, noting that (5.10) is still valid with U replaced by U_k and Λ replaced by Λ_k, we can get the approximation G_k and its inverse directly:

$$G_k = I - U_k \Lambda_k U_k^T, \quad G_k^{-1} = I + U_k[(I - \tilde{\Lambda}_k)^{-1} - I]U_k^T. \tag{5.16}$$

It may have become clear that it is possible to select $\tilde{\Lambda}$ so that AM^{-1} will have eigenvalues larger than one. Consider defining $\tilde{\Lambda}$ such that

$$\tilde{\lambda}_i = \left\{ \begin{array}{ll} \lambda_i & \text{if} \quad i \le k \\ \theta & \text{if} \quad i > k \end{array} \right. ,$$

and denote by $G_{k,\theta}^{-1}$ the related analogue of (5.16). Then, from (5.14) the eigenvalues of $G^{-1} - G_{k,\theta}^{-1}$ are

$$\left\{ \begin{array}{cl} 0 & \text{if} \quad i \le k \\ (1-\lambda_i)^{-1} - (1-\theta)^{-1} & \text{if} \quad i > k \end{array} \right. . \tag{5.17}$$

Note that for $i > k$, we have

$$\frac{1}{1-\lambda_i} - \frac{1}{1-\theta} = \frac{\lambda_i - \theta}{(1-\lambda_i)(1-\theta)},$$

and these eigenvalues can be made negative by selecting $\lambda_{k+1} \le \theta < 1$ and the choice that yields the smallest 2-norm is $\theta = \lambda_{k+1}$. The earlier definition of Λ_k in (5.15) that truncates the eigenvalues of H to zero corresponds to selecting $\theta = 0$.

Theorem 5.2.1 *Assume that A is SPD and θ is selected so that $\lambda_{k+1} \le \theta < 1$. Then the eigenvalues η_i of AM^{-1} are such that,*

$$1 \le \eta_i \le 1 + \frac{1}{1-\theta} \, \|A^{1/2}A_0^{-1}E\|_2^2. \tag{5.18}$$

Furthermore, the term $\|A^{1/2}A_0^{-1}E\|_2^2$ is bounded from above by a constant:

$$\|A^{1/2}A_0^{-1}E\|_2^2 \le \frac{1}{4}.$$

Proof. We rewrite (5.13) as $AM^{-1} = I + A(A_0^{-1}E)(G_k^{-1} - G^{-1})(E^TA_0^{-1})$ or upon applying a similarity transformation with $A^{1/2}$

$$A^{1/2}M^{-1}A^{1/2} = I + (A^{1/2}A_0^{-1}E)(G_k^{-1} - G^{-1})(E^TA_0^{-1}A^{1/2}). \tag{5.19}$$

From (5.17) we see that for $j \le k$ we have $\lambda_j(G_k^{-1} - G^{-1}) = 0$, and for $j > k$,

$$0 \le \lambda_j(G_k^{-1} - G^{-1}) = (1-\theta)^{-1} - (1-\lambda_j)^{-1} \le (1-\theta)^{-1}.$$

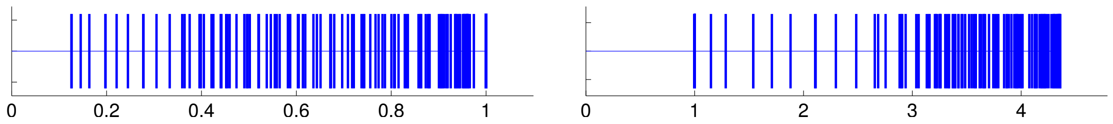
This is because $1/(1-t)$ is an increasing function and for $j > k$, we have $0 \le \lambda_j \le \lambda_{k+1} \le \theta$. The rest of the proof follows by taking the Rayleigh quotient of an arbitrary vector x and utilizing (5.19).

For the second part, first note that $\|A^{1/2}A_0^{-1}E\|_2^2 = \rho\left(E^TA_0^{-1}AA_0^{-1}E\right)$, where $\rho(\cdot)$ denotes the spectral radius of a matrix. Then, from $A = A_0 - EE^T$, we have

$$E^TA_0^{-1}AA_0^{-1}E = E^TA_0^{-1}E - \left(E^TA_0^{-1}E\right)\left(E^TA_0^{-1}E\right) \equiv H - H^2.$$

Lemma 5.2.1 states that each eigenvalue λ of H satisfies $0 \le \lambda < 1$. Hence, for each eigenvalue μ of $E^TA_0^{-1}AA_0^{-1}E$, $\mu = \lambda - \lambda^2$, which is between 0 and 1/4 for $\lambda \in [0, 1)$. This gives the desired bound $\|A^{1/2}A_0^{-1}E\|_2^2 \le 1/4$. ∎

Figure 5.1: DDLR-1: eigenvalues of AM^{-1} with $\theta = 0$ (left) and $\theta = \lambda_{k+1}$ (right) using $k = 5$ eigenvectors for a 900×900 2-D Laplacian with 4 subdomains and $\alpha = 1$.



An illustration of the spectra of AM^{-1} for the two cases when $\theta = 0$ and $\theta = \lambda_{k+1}$ with $k = 5$ is shown in Figure 5.1. The original matrix is a 900×900 2-D Laplacian obtained from a finite difference discretization of a square domain using 30 mesh points in each direction. The number of the subdomains used is 4, resulting in 119 interface unknowns. The reordered matrix associated with this example were shown in Figure 2.5.

For the second choice $\theta = \lambda_{k+1}$, Theorem 5.2.1 proved that $\|A^{1/2}A_0^{-1}E\|_2^2$ does not exceed 1/4, regardless of the mesh size and regardless of α, Numerical experiments will show that this term is close to 1/4 for Laplacian matrices. For the case with $\alpha = 1$, $\theta = \lambda_6 \approx 0.93492$ and $\|A^{1/2}A_0^{-1}E\|_2^2 \approx 0.24996$, so that the bound of the eigenvalues of AM^{-1} given by (5.18) is 4.8413, which is fairly close to the largest eigenvalue, which is 4.3581 (cf. the right part of Figure 5.1). When $\alpha = 2$, $\theta = \lambda_6 \approx 0.93987$ and $\|A^{1/2}A_0^{-1}E\|_2^2 \approx 0.25000$, so that the eigenvalue bound is 5.1575 whereas the largest eigenvalue is 4.6724. When $\alpha = 0.5$, $\theta = \lambda_6 \approx 0.96945$ and $\|A^{1/2}A_0^{-1}E\|_2^2 \approx 0.24999$, and thus the bound is 9.1840, compared with the largest eigenvalue 8.6917. Therefore, we conclude for this case $\alpha = 1$ gives the best spectral condition number of the preconditioned matrix, which is a typical result for SPD matrices.

We now address some implementation issues of the preconditioner related to the second choice with $\theta = \lambda_{k+1}$. Again all that is needed are U_k, Λ_k and θ. We can show an analogue to the expression (5.16) in the following proposition.

Proposition 5.2.2 *The following expression for $G_{k,\theta}^{-1}$ holds:*

$$G_{k,\theta}^{-1} = \frac{1}{1-\theta} I \; + \; U_k \left[(I - \Lambda_k)^{-1} - (1-\theta)^{-1} I\right] U_k^T. \tag{5.20}$$

Proof. We write $U = [U_k, W]$, where U_k is as before and W contains the remaining columns $u_{k+1}, \cdots, u_s$. Note that W is not available but we use the fact that $WW^T = I - U_k U_k^T$ for the purpose of this proof. With this, (5.10) becomes:

$$\begin{aligned} G_{k,\theta}^{-1} &= I + [U_k, W] \begin{pmatrix} (I-\Lambda_k)^{-1} - I & \\ & ((1-\theta)^{-1} - 1)I \end{pmatrix} [U_k, W]^T \\ &= I + U_k \left[(I-\Lambda_k)^{-1} - I\right] U_k^T + \left[(1-\theta)^{-1} - 1\right](I - U_k U_k^T) \\ &= \frac{1}{1-\theta} I \; + \; U_k \left[(I-\Lambda_k)^{-1} - (1-\theta)^{-1} I\right] U_k^T. \end{aligned}$$

$\square$

Proposition 5.2.3 *Let the assumptions of Lemma 5.2.1 be satisfied. The preconditioner* (5.12) *with the matrix* $G_{k,\theta}^{-1}$ *defined by* (5.20) *is well-defined and SPD when* $\theta < 1$.

Proof. From (5.20), the eigenvalues of $G_{k,\theta}^{-1}$ are $(1-\lambda_i)^{-1}$, $i = 1, \dots, k$ or $(1-\theta)^{-1}$. Recall from Lemma 5.2.1, $0 \le \lambda_i < 1$ for all i and thus $G_{k,\theta}^{-1}$ is well-defined and SPD when $\theta < 1$. Hence, preconditioner (5.12) is SPD. □

We refer to the preconditioner (5.12) with $G_k^{-1} = G_{k,\theta}^{-1}$ as the *one-sided* DDLR preconditioner, abbreviated by DDLR-1.

5.2.3 Two-sided low-rank approximation

The method to be presented in this section uses low-rank approximations for more terms in (5.8), which yields a preconditioner that has a simpler form. Compared with the DDLR-1 method, the resulting preconditioner is less expensive to apply and less accurate in general. Suppose that $A_0^{-1}E \in \mathbb{R}^{n\times s}$ is factored in the form

$$A_0^{-1}E = UV^T, \tag{5.21}$$

as obtained from the singular value decomposition (SVD), where $U \in \mathbb{R}^{n\times s}$ and $V \in \mathbb{R}^{s\times s}$ is orthogonal. Then, for the matrix G in (5.7), we have the following lemma.

Lemma 5.2.2 *Let* $G = I - E^T A_0^{-1} E$ *as defined by* (5.7) *be nonsingular. Then,*

$$G^{-1} = I + V\left(I - U^T E V\right)^{-1} U^T E.$$

Furthermore, the following relation holds,

$$V^T G^{-1} V = \left(I - U^T E V\right)^{-1}. \tag{5.22}$$

Proof. For G^{-1}, we can write

$$G^{-1} = \left(I - (E^T A_0^{-1})E\right)^{-1} = \left(I - VU^T E\right)^{-1} = I + V\left(I - U^T E V\right)^{-1} U^T E.$$

Relation (5.22) follows from

$$V^T G^{-1} V = V^T (I + V\left(I - U^T E V\right)^{-1} U^T E) V = \left(I - U^T E V\right)^{-1}.$$

□

From (5.21), the best 2-norm rank-k approximation to $A_0^{-1}E$ is of the form

$$A_0^{-1}E \approx U_k V_k^T, \tag{5.23}$$

where $U_k \in \mathbb{R}^{n\times k}$ and $V_k \in \mathbb{R}^{s\times k}$ with $V_k^T V_k = I$ consist of the first k columns of U and V respectively. For an approximation to G, we define the matrix G_k as

$$G_k = I - V_k U_k^T E \ . \tag{5.24}$$

Then, the expression of A^{-1} in (5.8) will yield the preconditioner:

$$M^{-1} = A_0^{-1} + U_k(V_k^T G_k^{-1} V_k)U_k^T .$$

This means that we can build an approximate inverse based on a low-rank correction of the form that avoids the use of V_k explicitly,

$$M^{-1} = A_0^{-1} + U_k H_k U_k^T \quad \text{with} \quad H_k = V_k^T G_k^{-1} V_k. \tag{5.25}$$

Note that Lemma 5.2.2 will also hold if U and V are replaced with U_k and V_k. As a result, the matrix H_k has an alternative expression that is more amenable to computation. Specifically, we can show the following lemma.

Lemma 5.2.3 *Let G_k be defined by (5.24) and assume that matrix $I - U_k^T E V_k$ is non-singular. Then,*

$$G_k^{-1} = I + V_k \hat{H}_k U_k^T E \quad \textit{with} \quad \hat{H}_k = (I - U_k^T E V_k)^{-1}.$$

Furthermore, the following relation holds:

$$V_k^T G_k^{-1} V_k = \hat{H}_k$$

i.e., the matrix H_k in (5.25) and the matrix $\hat{H}_k$ are equal.

Proof. A proof can be directly obtained from the proof of Lemma 5.2.2 by replacing matrices U,V and G with U_k,V_k and G_k respectively. □

The application of (5.25) requires one solve with A_0 and a low-rank correction with U_k and H_k. Since $A_0^{-1}E$ is approximated on both sides of G in (5.8), we refer to this preconditioner as a *two-sided* DDLR preconditioner and use the abbreviation DDLR-2.

Proposition 5.2.4 *Assume that $U_kV_k^T$ in* (5.23) *is the best 2-norm rank-k approximation to $A_0^{-1}E$, and that A_0 is SPD. Then the preconditioner given by* (5.25) *is well-defined and SPD if and only if $\rho(U_k^TEV_k) < 1$.*

Proof. The proof follows from Proposition 4.9.2 for showing the symmetry of H_k, and Proposition 4.9.3 and Theorem 4.9.1 for the if-and-only-if condition. □

Next, we will show that the eigenvalues of the preconditioned matrix AM^{-1} are between zero and one. Suppose that $U_kV_k^T$ is obtained as in (5.23), so that we have $\left(A_0^{-1}E\right)V_k = U_k$. Then, the preconditioner (5.25) can be rewritten as

$$\begin{aligned} M^{-1} &= A_0^{-1} + U_kH_kU_k^T = A_0^{-1} + \left(A_0^{-1}E\right)V_kH_kV_k^T\left(E^TA_0^{-1}\right) \\ &= A_0^{-1} + \left(A_0^{-1}E\right)V\begin{pmatrix} H_k & 0 \\ 0 & 0 \end{pmatrix}V^T\left(E^TA_0^{-1}\right), \end{aligned} \tag{5.26}$$

where U and V are defined in (5.21). We write $U = \left[U_k, \bar{U}\right]$ and $V = \left[V_k, \bar{V}\right]$, where $\bar{U}$ and $\bar{V}$ consist of the $s-k$ columns of U and V that are not contained in U_k and V_k. Recall that $H_k^{-1} = I - U_k^TEV_k$ and define $X = I - \bar{U}^TE\bar{V}$, $Z = -U_k^TE\bar{V}$. From (5.11) and (5.21), we have

$$A^{-1} = A_0^{-1} + \left(A_0^{-1}E\right)V\left(V^TG^{-1}V\right)V^T\left(E^TA_0^{-1}\right),$$

from which and (5.22), it follows that

$$\begin{aligned} A^{-1} &= A_0^{-1} + \left(A_0^{-1}E\right)V\left(I - U^TEV\right)^{-1}V^T\left(E^TA_0^{-1}\right), \\ &= A_0^{-1} + \left(A_0^{-1}E\right)V\begin{pmatrix} H_k^{-1} & Z \\ Z^T & X \end{pmatrix}^{-1}V^T\left(E^TA_0^{-1}\right). \end{aligned} \tag{5.27}$$

Let the Schur complement of H_k^{-1} be

$$S_k = X - Z^TH_kZ \in \mathbb{R}^{(s-k)\times(s-k)}, \tag{5.28}$$

and define matrix $\bar{S}_k \in \mathbb{R}^{2(s-k)\times 2(s-k)}$ by

$$\bar{S}_k = \begin{pmatrix} S_k^{-1} & -I \\ -I & S_k \end{pmatrix}. \tag{5.29}$$

Then, the following lemma shows that S_k is SPD and $\bar{S}_k$ is SPSD.

Lemma 5.2.4 *Assume that G defined by* (5.8) *is nonsingular as well as the matrix $I - U_k^T E V_k$. Then, the Schur complement S_k defined by* (5.28) *is SPD. Moreover, matrix $\bar{S}_k$ is SPSD with $s - k$ positive eigenvalues and $s - k$ zero eigenvalues.*

Proof. From Lemma 5.2.1, we can infer that the eigenvalues of G are all positive. Thus, G is SPD and so is matrix $V^T G^{-1} V$. In the end, the Schur complement S_k is SPD when H_k is nonsingular. The signs of the eigenvalues of $\bar{S}_k$ can be easy revealed by a block LDL factorization. □

Theorem 5.2.2 *Assume that A is SPD. Then the eigenvalues η_i of AM^{-1} with M^{-1} given by* (5.25) *satisfy $0 < \eta_i \leq 1$.*

Proof. From (5.26) and (5.27), it follows by subtraction that

$$\begin{aligned} A^{-1} - M^{-1} &= \left(A_0^{-1}E\right) V \left[\begin{pmatrix} H_k^{-1} & Z \\ Z^T & X \end{pmatrix}^{-1} - \begin{pmatrix} H_k & 0 \\ 0 & 0 \end{pmatrix} \right] V^T \left(E^T A_0^{-1}\right), \\ &= \left(A_0^{-1}E\right) V \begin{pmatrix} H_k Z S_k^{-1} Z^T H_k & -H_k Z S_k^{-1} \\ -S_k^{-1} Z^T H_k & S_k^{-1} \end{pmatrix} V^T \left(E^T A_0^{-1}\right), \\ &= \left(A_0^{-1}E\right) V \begin{pmatrix} H_k Z & 0 \\ 0 & S_k^{-1} \end{pmatrix} \bar{S}_k \begin{pmatrix} Z^T H_k & 0 \\ 0 & S_k^{-1} \end{pmatrix} V^T \left(E^T A_0^{-1}\right), \end{aligned}$$

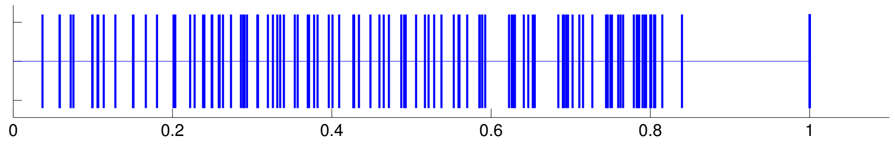
where $\bar{S}_k$ is defined in (5.29), so that

$$AM^{-1} = I - A\left(A_0^{-1}E\right) V \begin{pmatrix} H_k Z & 0 \\ 0 & S_k^{-1} \end{pmatrix} \bar{S}_k \begin{pmatrix} Z^T H_k & 0 \\ 0 & S_k^{-1} \end{pmatrix} V^T \left(E^T A_0^{-1}\right).$$

Hence, the eigenvalues of AM^{-1}, η_i, satisfy $0 < \eta_i \leq 1$, since $\bar{S}_k$ is SPSD, and $(n-s+k)$ of these eigenvalues are equal to one. □

The spectrum of AM^{-1} for the same matrix used for Figure 5.1 is shown in Figure 5.2. Compared with the spectrum with the DDLR-1 method with $\theta = 0$ shown in the left part of Figure 5.1, the eigenvalues of the preconditioned matrix AM^{-1} are more dispersed between 0 and 1 and the small eigenvalues are closer to zero. This suggests that the quality of the DDLR-2 preconditioner will be lower than that of DDLR-1, which is supported by the numerical results in Section 5.4.

Figure 5.2: DDLR-2: eigenvalues of AM^{-1} with $k = 5$ eigenvectors for a 900×900 2-D Laplacian with 4 subdomains and $\alpha = 1$.



5.3 Parallel implementation

In this section, we address the implementation details for building and applying the DDLR preconditioners, especially focusing on the implementations in a parallel/distributed environment.

5.3.1 Building a DDLR preconditioner

The construction of a DDLR preconditioner involves the following steps. In the first step, a graph partitioner is called on the adjacency graph to partition the domain. For each obtained subdomain, we separate the interior nodes and the interface nodes, and reorder the local matrix into the form of (5.3). The second step is to build a solver for each $B_{i,\alpha} \equiv B_i + \alpha^{-2} E_i E_i^T$. These two steps can be done in parallel. The third step is to build a solver for the *global* matrix C_α. We will focus on the solution methods for the linear systems with C_α in Section 5.3.3. The last step, which is the most expensive one, is to compute the low-rank approximations. This will be discussed in Section 5.3.4.

5.3.2 Applying the DDLR preconditioner

First, consider the DDLR-1 preconditioner (5.12), which we can rewrite as

$$M^{-1} = A_0^{-1} \left(I + E G_{k,\theta}^{-1} E^T A_0^{-1} \right). \tag{5.30}$$

The steps involved in applying M^{-1} to a vector x are listed in Algorithm 7. The vector u resulting from the last step will be the desired vector $u = M^{-1}x$. The solve with A_0 required in steps 1 and 5 of Algorithm 7, can in turn be viewed as consisting of the p independent local solves with $B_{i,\alpha}$ and the global solve with $C_\alpha \equiv C + \alpha^2 I$ as is inferred from (5.6). Recall that the matrix C, which has the block structure (2.24), is the global interface matrix that couples all the interface unknowns. So, solving a linear system with C_α will require communication if C_α is assigned to different processors. The multiplication with E^T in step 2 transforms a vector of the interior unknowns into a vector of the interface unknowns. This can be likened to a *descent* operation that moves objects from a “fine” space to a “coarse” space. The multiplication with E in step 4 performs the reverse operation, which can be termed an *ascent* operation, consisting of going from the interface unknowns to the interior unknowns. Finally, the operation with $G_{k,\theta}^{-1}$ in step 3 involves all the interface unknowns, and it will also require communication. In summary, there are essentially 4 types of operations: (1) the solve with $B_{i,\alpha}$; (2) the solve with C_α; (3) products with E and E^T, which are dual of one another; and (4) the application of $G_{k,\theta}^{-1}$ to vectors.

Algorithm 7 Preconditioning operations of the DDLR-1 preconditioner.

```
1: Solve: $A_0 z = x$                      {$B_{i,\alpha}$ solves and $C_\alpha$ solve}
2: Compute: $y = E^T z$                    {Interior unknowns to interface neighbors}
3: Compute: $w = G_{k,\theta}^{-1} y$      {Use (5.20)}
4: Compute: $v = Ew$                       {Interface unknowns to interior neighbors}
5: Solve: $A_0 u = x + v$                  {$B_{i,\alpha}$ solves and $C_\alpha$ solve}
```

Next, consider the DDLR-2 preconditioner given by (5.25). Applying this preconditioner is much simpler, which consists of one solve with A_0 and a low-rank correction. Communication will be required for applying the low-rank correction term, $U_k H_k U_k^T$, to a vector because it involves all the unknowns. We assume that the $k \times k$ matrix H_k is stored on every processor.

Parallel implementations of the DDLR methods will depend on how the interface unknowns are mapped to processors. A few of the mapping schemes will be discussed in Section 5.3.5.

5.3.3 Global solve with $\mathbf{C}_\alpha$

This section addresses the solution methods for C_α required in both the DDLR-1 and the DDLR-2 methods whenever solving a linear system with A_0 is needed. It is an important part of the computations, especially for DDLR-1 as it takes place twice for each iteration. In addition, it is a non-local computation and can be costly due to the communication. An important characteristic of C_α is that it can be made strongly diagonally dominant by selecting a proper scaling factor α. Therefore, the first approach one can think about is to use a few steps of the Chebyshev iterations. The Chebyshev method was used with a block Jacobi preconditioner D_α consisting of all the local diagonal blocks C_i (see, e.g., [153, §2.3.9] for the preconditioned Chebyshev method). An appealing property in the Chebyshev iterations is that no inner product is needed. This avoids communications among processors, which makes this method efficient in particular for distributed memory architectures [12]. The price one pays for avoiding communication is that this method requires enough knowledge of the spectrum. Therefore, prior to the Chebyshev iterations, we performed a few steps of the Lanczos iterations on the matrix pair (C_α, D_α) [154, §9.2.6] for some estimates (not bounds) of the smallest and the largest eigenvalues. The safeguard terms used in [119] were included in order to have bounds of the spectrum (see [120, §13.2] for the definitions of these terms).

Another approach is to resort to an approximate inverse $X \approx C_\alpha^{-1}$, so that the solve with C_α will be reduced to a matrix vector product with X. A simple scheme known as the method of Hotelling and Bodewig [155] is given by the iteration

$$X_{k+1} = X_k(2I - C_\alpha X_k).$$

In the absence of dropping, this scheme squares the residual norm $\|I - C_\alpha X_k\|$ from one step to the next, so that it converges quadratically provided that the initial guess X_0 is such that $\|I - C_\alpha X_0\| < 1$ for some matrix norm. The global self-preconditioned minimal residual (MR) iterations were shown to have superior performance [122]. We adopted this method to build an approximate inverse of C_α. Given an initial guess X_0, the self-preconditioned MR iterations can be obtained by the sequence of operations shown in Algorithm 8. X_0 was selected as the inverse of the diagonal of C_α. The numerical dropping was performed by a dual threshold strategy based on a drop tolerance and a maximum number of nonzeros per column.

Algorithm 8 Self-preconditioned global MR iterations with dropping.

```
1: Compute: $R_k = I - C_\alpha X_k$                                  {residual}
2: Compute: $Z_k = X_k R_k$                          {self-preconditioned residual}
3: Apply numerical dropping to $Z_k$
4: Compute: $\beta_k = \mathrm{tr}(R_k^T C_\alpha Z_k)/\|C_\alpha Z_k\|_F^2$        {tr(·) denotes the trace}
5: Compute: $X_{k+1} = X_k + \beta_k Z_k$
```

5.3.4 Computing the low-rank approximations

For the DDLR-1 method, we use the Lanczos algorithm [118] to compute the low-rank approximation to $E^T A_0^{-1} E$ of the form $U_k \Lambda_k U_k^T$. For the DDLR-2 method, the low-rank approximation to $A_0^{-1}E$ has the form $U_k V_k^T$, which can be computed by applying the Lanczos algorithm to $E^T A_0^{-2} E$, where V_k is computed and U_k can be obtained by $U_k = A_0^{-1} E V_k$. Alternatively, for the DDLR-2 method, we can also use the Lanczos bidiagonalization method [140, §10.4] to compute U_k and V_k at the same time as in the MLR preconditioner discussed in Section 4.5. At each step of the Lanczos procedure, a matrix-vector product is required. This means that for each step, we need to solve linear systems with A_0: one for the DDLR-1 method and two for the DDLR-2 method.

Partial reorthogonalization [143, 144] was used in the Lanczos algorithm, the cost of which will not be an issue when only a small number of eigenpairs are computed. The scheme in (4.15) was used to monitor convergence of the computed eigenvalues.

5.3.5 Parallel implementations: standard mapping

Considerations of the parallel implementations have been mentioned in the previous sections, which suggest several possible schemes for distributing the interface unknowns. Before discussing these schemes, it will be helpful to overview the issues at hand. Major computations in building and applying the DDLR preconditioners are the following:

1. solve with $B_{i,\alpha}$, (local)
2. solve with C_α, (nonlocal)
3. products with E^T and E, (local)

4a. for DDLR-1, applying $G_{k,\theta}^{-1}$ in (5.20), (nonlocal)

4b. for DDLR-2, products with U_k and U_k^T, (nonlocal)

5. reorthogonalizations in the Lanczos procedure. (nonlocal)

The most straightforward mapping we can consider might be to map the unknowns of each subdomain to a processor. If p subdomains are used, global matrices A and C_α or its approximate inverse X are distributed among the p processors. So, processor i will hold $d_i + s_i$ rows of A and s_i rows of C_α or X, where d_i is the number of the local interior unknowns and s_i is the number of the local interface unknowns of subdomain i. In the DDLR-1 method, $U_k \in \mathbb{R}^{s \times k}$ is distributed such that processor i will keep s_i

rows, while in the DDLR-2 method, $d_i + s_i$ rows of $U_k \in \mathbb{R}^{n\times k}$ will reside in processor i. For all the nonlocal operations, communication is among all the p processors. The operations labeled by (2.) and (4a.) involve interface to interface communication, while the operations (4b.) and (5.) involve communication among all the unknowns. From another perspective, the communication in (4a.), (4b.) and (5.) is of the all-reduction type required by vector inner products, while the communication in (2.) is point-to-point such as that in the distributed sparse matrix vector products. If an iterative process is used for the solve with C_α, it is important to select α carefully so as to reach a compromise between the number of the inner iterations (each of which requires communication) and the number of the outer iterations (each of which involves solves with C_α). The scalar α will also play a role if an approximate inverse is used, since the convergence of the MR iterations will be affected.

5.3.6 Unbalanced mapping: interface unknowns together

Since communication is required among the interface nodes, an idea that comes to mind is to map the interior unknowns of each subdomain to a processor, and all the interface unknowns to another separated one. In a case of p subdomains, $p+1$ processors will be used and A is distributed in such a way that processor i owns the rows corresponding to the local interior unknowns for $i = 1, \ldots p$, while processor $p+1$ holds the rows related to all the interface unknowns. Thus, C_α or X will reside entirely on the processor $p+1$.

A clear advantage of this mapping is that the solve with C_α will require no communication. However, the operations with E and E^T are no longer local. Indeed, E^T can be viewed as a restriction operator, which "scatters" interface data from processor $p+1$ to the other p processors. Specifically, referring to (5.3), each y_i will be sent to processor i from processor $p+1$. Analogously, the product with E, as a prolongation, will perform a dual operation that "gathers" from processors 1 to p to processor $p+1$. In Algorithm 7, the scatter operation goes before step 2 and the gather operation should be executed after step 4. Likewise, if we store the vectors in U_k on processor $p+1$, applying $G_{k,\theta}$ will not require communication but another pair of the "gather-and-scatter" operations will be needed before and after step 3. Therefore, at each application of the DDLR-1 preconditioner, *two* pairs of the scatter-and-gather operations for the interface unknowns will be required. A middle ground approach is to distribute U_k to processors 1 to p as it is in the standard mapping. In this way, applying $G_{k,\theta}$ will require communication but only *one* pair of the scatter-and-gather operations is necessary. On the other hand, in the DDLR-2 method, the distribution of U_k should be consistent with that of A.

The main issue with this mapping is that it is hard to achieve load balancing in general. Indeed for a good balancing, we need to have the interior unknowns of each subdomain and all the interface unknowns of roughly the same size. However, this is difficult to achieve in practice. The load balancing issue is further complicated by the fact that the equations needed to be solved on processor $p+1$ are completely different from those on the other processors. A remedy to the load balancing issue is to use q processors instead of just one dedicated to the global interface (a total of $p+q$ processors used in all), which provides a compromise. Then, the communication required for solving with C_α and applying $G_{k,\theta}$ is confined within the q processors.

5.4 Numerical experiments

The experiments were conducted on Itasca, an HP ProLiant BL280c G6 Linux cluster at Minnesota Supercomputing Institute, which has $2,186$ Intel Xeon X5560 processors. Each processor has four cores, 8 MB cache, and communicates with memory on a QuickPath Interconnect (QPI) interface. An implementation of the DDLR preconditioners was written in C/C++ with the Intel Math Kernel Library, the Intel MPI library and PETSc [156–158], compiled by the Intel MPI compiler using the -O3 optimization level.

The accelerators used were the CG method when both the matrix and the preconditioner are SPD, and GMRES(40) for the indefinite cases. Three types of preconditioners were compared in our experiments: 1) the DDLR preconditioners, 2) the pARMS method [88], and 3) the RAS preconditioner [83] (with overlapping). Recall that for an SPD matrix, the DDLR preconditioners given by (5.12) and (5.25) will also be SPD if the assumptions in Propositions 5.2.3 and 5.2.4 are satisfied. However, these propositions will not hold when the solves with A_0 are approximate, which is typical in practice. Instead, the positive definiteness can be determined by checking if the largest eigenvalue is less than one for DDLR-1 or by checking the positive definiteness of H_k for DDLR-2.

Each $B_{i,\alpha}$ was reordered by the approximate minimum degree ordering (AMD) [2, 110,111] to reduce fill-ins and then we used an incomplete Cholesky or LDL factorization as the local solver. A more efficient and robust solver, for example, the ARMS approach in [62], can lead to better performance in terms of both the memory requirement and the speed. This has not been implemented in our current code. A typical setting of the scalar α for C_α and $B_{i,\alpha}$ is $\alpha = 1$. It usually gives the best overall performance, the exceptions being the 3 cases in Section 5.4.2, for which choosing $\alpha > 1$ improved the performance. Regarding solves with C_α, using the approximate inverse is generally more efficient than the Chebyshev iterations, especially in the iteration phase. On the other hand, computing the approximate inverse can be costly, in particular for indefinite 3-D cases. The standard mapping was adopted unless specially stated. It in general gives better performance than the unbalanced mapping. The behavior of these two mappings will be analyzed by the results in Table 5.3. In the Lanczos algorithm, the convergence was checked every 10 steps and the tolerance ϵ in (4.15) was 10^{-4}. In addition, the maximum number of steps was five times the number of the requested eigenvalues.

For pARMS, the ARMS method was used to be the local preconditioner and the Schur complement method was used as the global preconditioner, where the reduced system was solved by a few inner Krylov subspace iterations preconditioned by the block-Jacobi preconditioner. For the details of these options in pARMS, we refer the readers to [62, 159]. We point out that when the inner iterations are enabled, flexible Krylov subspace methods will be required for the outer iterations, since the preconditioning is no longer fixed from one outer iteration to the next. So, the flexible GMRES [160] was used. For the RAS method, ILU(k) was used as the local solver, and overlapping between subdomains was used, where the level of overlapping was increased with the problem size. Note that the RAS preconditioner is nonsymmetric even for a symmetric matrix, so that GMRES was used with it.

We first report on the results of solving the linear systems from a 2-D and a 3-D PDEs on regular meshes. Next, we will show the results for solving a sequence of general sparse symmetric linear systems. For all the problems, a parallel multilevel k-way graph partitioning algorithm from ParMetis [76, 161] was used for the DD. Iterations were stopped whenever the residual norm had been reduced by 6 orders of magnitude or the maximum number of iterations allowed, which is 500, was exceeded. The results are shown in Tables 5.1, 5.2 and 5.5, where all timings are reported in seconds and '`F`' indicates non-convergence within the maximum allowed number of steps.

5.4.1 2-D/3-D model problems

We examine a 2-D and a 3-D PDE,

$$\begin{aligned} -\Delta u - cu = f \ \text{ in } \Omega, \\ u = 0 \text{ on } \partial\Omega, \end{aligned} \tag{5.31}$$

where the domain is the unit box with $\Omega = (0,1)^2$ or $\Omega = (0,1)^3$, and $\partial\Omega$ is the boundary where we use we use the Dirichlet boundary condition. We take the 5-point (or 7-point) centered difference approximation on regular 2-D (3-D) grids.

To begin with, we solve (5.31) with $c = 0$. The matrix is SPD, so that we use the DDLR preconditioners with the CG method. Numerical experiments were carried out to compare the performance of DDLR with those of pARMS and RAS. The results are shown in Table 5.1. The mesh sizes, the number of processors (Np), the rank (rk), the fill-ratios (nz), the numbers of iterations (its), the time for building the preconditioners (p-t) and the time for iterations (i-t) are tabulated. We tested the problems on 6 2-D and 6 3-D meshes of increasing sizes, where the number of processors was growing proportionally such that the problem size on each processor was kept roughly the same. This can serve as a *weak scaling* test. We increased the rank k used in DDLR with the meshes sizes. The fill-ratios of DDLR-1 and pARMS were controlled to be roughly equal, whereas the fill of DDLR-2 was much higher, which comes mostly from the matrix U_k when k is large. For pARMS, the inner Krylov subspace dimension used was 3.

Compared with the pARMS method and the RAS preconditioner, the time for building DDLR is much higher and it grows with the rank and the number of the processors. In contrast, the time to build pARMS and RAS is roughly constant. This set-up time for DDLR is typically dominated by the Lanczos algorithm, where solves with $B_{i,\alpha}$ and C_α are required at each iteration. Moreover, when k is large, the cost of reorthogonalization becomes significant. As shown in Table 5.1, DDLR-1 and pARMS were more robust as they succeeded for all the 2-D and 3-D cases, while DDLR-2 failed for the largest 2-D case and RAS failed for the two largest ones. For most of the 2-D problems, DDLR-1/CG achieved convergence in the fewest iterations and the best iteration time. For the 3-D problems, DDLR-1 required more iterations but performance gain was still achieved in terms of the reduced iteration time.

Table 5.1: Comparison between the DDLR, pARMS and RAS preconditioners for solving SPD linear systems from the 2-D/3-D PDE with CG.

Mesh	Np	DDLR-1					DDLR-2				
		rk	nz	its	p-t	i-t	rk	nz	its	p-t	i-t
128^2	2	8	6.6	15	.209	.027	8	8.2	30	.213	.031
256^2	8	16	6.6	34	.325	.064	16	9.7	69	.330	.083
512^2	32	32	6.8	61	.567	.122	32	13.0	132	.540	.194
1024^2	128	64	7.0	103	1.12	.218	64	19.3	269	1.03	.570
1448^2	256	91	7.2	120	1.67	.269	91	24.7	385	1.72	1.05
2048^2	512	128	7.6	168	3.02	.410	128	32.2	F	-	-
25^3	2	8	7.2	11	.309	.025	8	8.3	17	.355	.021
50^3	16	16	7.5	27	.939	.064	16	9.3	52	.958	.076
64^3	32	16	7.4	36	1.06	.089	16	9.2	67	1.07	.102
100^3	128	32	8.0	52	1.57	.136	32	11.5	101	1.48	.190
126^3	256	32	8.2	60	2.07	.178	32	12.5	126	1.87	.265
159^3	512	51	8.7	65	2.92	.251	51	14.2	156	2.50	.387

Mesh	Np	pARMS				RAS			
		nz	its	p-t	i-t	nz	its	p-t	i-t
128^2	2	6.7	15	.062	.037	6.5	25	.004	.022
256^2	8	6.7	30	.066	.082	6.8	56	.014	.027
512^2	32	6.9	52	.072	.194	6.8	103	.049	.279
1024^2	128	6.6	104	.100	.359	6.4	175	.055	.559
1448^2	256	6.6	247	.073	.820	7.6	F	.078	-
2048^2	512	7.8	282	.080	1.06	7.8	F	.112	-
25^3	2	7.3	9	.100	.032	7.5	13	.005	.094
50^3	16	8.1	17	.179	.095	7.8	26	.016	.177
64^3	32	8.2	20	.142	.121	8.3	32	.027	.300
100^3	128	8.3	29	.170	.198	8.0	46	.011	.349
126^3	256	8.4	34	.166	.216	8.9	48	.015	.513
159^3	512	8.5	40	.179	.275	8.9	68	.075	.688

Next, we consider solving symmetric indefinite problems by setting $c > 0$ in (5.31), which corresponds to shifting the discretized negative Laplacian by subtracting σI with a certain $\sigma > 0$. In this set of experiments, we reduce the size of the shift as the problem size increases in order to make the problems fairly difficult but not too difficult to solve for all the methods. We used higher ranks in the two DDLR methods and a higher inner iteration number, from 10 to 20, in pARMS. Results are reported in Table 5.2.

Table 5.2: Comparison between the DDLR, pARMS and RAS preconditioners for solving symmetric indefinite linear systems from the 2-D/3-D PDE with GMRES(40).

Mesh	Np	σ	DDLR-1					DDLR-2				
			rk	nz	its	p-t	i-t	rk	nz	its	p-t	i-t
128^2	2	1e-1	16	6.8	18	.233	.034	16	13.2	146	.310	.234
256^2	8	1e-2	32	6.8	38	.674	.080	16	13.0	F	1.01	-
512^2	32	1e-3	64	7.1	48	1.58	.105	64	19.4	F	1.32	-
1024^2	128	2e-4	128	7.6	68	4.15	.160	128	32.3	F	4.45	-
1448^2	256	5e-5	182	8.1	100	7.14	.253	182	43.2	F	7.77	-
2048^2	512	2e-5	256	8.8	274	12.6	.749	256	58.4	F	13.1	-
25^3	2	3e-1	16	8.3	29	.496	.099	16	9.6	62	.595	.130
50^3	16	7e-2	32	8.5	224	1.64	1.50	32	10.2	F	1.66	-
64^3	32	4e-2	64	8.9	103	3.02	.741	64	16.3	F	2.08	-
100^3	128	2e-2	128	11.4	319	5.17	2.82	128	28.7	F	5.29	-
126^3	256	1e-2	128	11.4	253	8.77	2.75	128	28.3	F	9.21	-
159^3	512	6e-3	160	12.0	221	19.6	2.54	160	33.0	F	20.3	-

Mesh	Np	σ	pARMS				RAS			
			nz	its	p-t	i-t	nz	its	p-t	i-t
128^2	2	1e-1	11.4	76	.114	.328	6.7	F	.003	-
256^2	8	1e-2	13.9	F	.126	-	6.7	F	.004	-
512^2	32	1e-3	12.3	298	.181	1.53	7.7	F	.005	-
1024^2	128	2e-4	12.5	232	.230	1.46	7.7	F	.008	-
1448^2	256	5e-5	12.5	F	.256	-	8.3	F	.011	-
2048^2	512	2e-5	12.6	314	.195	2.13	8.7	F	.015	-
25^3	2	3e-1	8.8	72	.156	.445	8.2	112	.011	.152
50^3	16	7e-2	14.3	361	.239	6.78	10.0	F	.024	-
64^3	32	4e-2	10.7	F	.144	-	9.8	360	.017	.877
100^3	128	2e-2	11.3	F	.180	-	11.0	F	.052	-
126^3	256	1e-2	11.5	226	.205	5.64	10.6	332	.053	1.55
159^3	512	6e-3	11.6	258	.223	16.4	10.5	F	.155	-

First, we can see that the DDLR-2 method did not perform well as it failed for almost all the problems. Second, the RAS method failed for all the 2-D problems and three 3-D ones. But for the case on a 126^3 mesh, it yielded the best iteration time. Third, the DDLR-1 preconditioner achieved convergence in all the cases whereas pARMS failed for 4 problems. When comparing the DDLR-1 method with the pARMS method, DDLR-1 appears to be more robust for indefinite problems, and moreover, for all the 2-D and

3-D cases, DDLR-1 required fewer iterations and less iteration time.

In all the previous tests, DDLR-1 was used with the standard mapping. In the next set of experiments, we examined the behavior of the unbalanced mapping discussed in Section 5.3.6. In these experiments, we tested the problem on a 128×128 mesh and a $25 \times 25 \times 25$ mesh. Both of them were divided into 128 subdomains. Note here that the problem size per processor is remarkably small. This was made on purpose since it can make the communication cost more significant (and likely to be dominant) in the overall cost for the solve with C_α such that it can make the effect of the unbalanced mapping more prominent. Table 5.3 lists the iteration time (in milliseconds) for solving the SPD PDE problems using the standard mapping and the unbalanced mapping with different settings. Two solution methods for C_α were tested, the one with the approximate inverse and the preconditioned Chebyshev iterations (5 iterations were used per solve). In the unbalanced mapping, q processors were used dedicated to the interface unknowns ($p+q$ processors were used totally). The unbalanced mapping was tested with 8 different q values from 1 to 96. $q = 1$ is a special case where no communication is involved in the solve with C_α. The matrix U_k was stored on the p processors, so that only one pair of the scatter-and-gather communication was required at each outer iteration as discussed in Section 5.3.6. The standard mapping is indicated by $q = 0$.

Table 5.3: Iteration time of the DDLR-1-CG method with the standard mapping and the unbalanced mapping for solving the SPD PDE problems.

Mesh	C_α^{-1}	$q=0$	1	2	4	8	16	32	64	96
128^2	`AINV`	9.0	53.4	27.8	15.4	13.9	10.8	**9.1**	9.4	13.5
	`Cheb`	9.2	116.8	60.7	29.9	15.8	10.7	10.0	**8.3**	9.1
25^3	`AINV`	15.1	119.7	66.3	34.7	26.0	19.2	17.2	**14.8**	18.2
	`Cheb`	13.8	368.3	166.0	78.3	38.5	20.4	14.4	**12.3**	13.5

As the results indicated, the iteration time kept decreasing at the beginning as q increased but after some point it started to increase. This is a typical situation corresponding to the balance between communication and computation: when q is small, the amount of computation on each of the q processors is high and it dominates the overall cost, so that the overall cost will keep being reduced as q increases until the point when the communication cost starts to affect the overall performance. The optimal numbers of the interface processors that yielded the best iteration time are shown in bold in Table 5.3. For these two cases, the optimal iteration time with the unbalanced mapping was slightly better than that with the standard mapping. However, we need to point out that this is not a typical case in practice. For all the other tests in this section, we used the standard mapping with the DDLR-1 preconditioner.

5.4.2 General matrices

We selected 12 symmetric matrices from the University of Florida sparse matrix collection [26] for the following tests. Table 5.4 lists the name, the order (N), the number of nonzeros (NNZ), and a short description for each matrix. If the actual right-hand side is not provided, an artificial one was created as $b = Ae$, where e is a random vector.

Table 5.4: Names, orders (N), numbers of nonzeros (NNZ) and short descriptions of the test matrices.

Matrix	N	NNZ	Description
Andrews/Andrews	60,000	760,154	computer graphics problem
UTEP/Dubcova2	65,025	1,030,225	2-D/3-D PDE problem
Rothberg/cfd1	70,656	1,825,580	CFD problem
Schmid/thermal1	82,654	574,458	thermal problem
Rothberg/cfd2	123,440	3,085,406	CFD problem
UTEP/Dubcova3	146,689	3,636,643	2-D/3-D PDE problem
Botonakis/thermo_TK	204,316	1,423,116	thermal problem
Wissgott/para_fem	525,825	3,674,625	CFD problem
CEMW/tmt_sym	726,713	5,080,961	electromagnetics problem
McRae/ecology2	999,999	4,995,991	landscape ecology problem
McRae/ecology1	1,000,000	4,996,000	landscape ecology problem
Schmid/thermal2	1,228,045	8,580,313	thermal problem

Table 5.5 shows the result for each problem. DDLR-1 and DDLR-2 were used with GMRES(40) for three problems `tmt_sym`, `ecology1` and `ecology2`, where the preconditioners were found not to be SPD, while for the other problems CG was applied. We set the scalar $\alpha = 2$ for two problems `ecology1` and `ecology2`, where it turned out to reduce the numbers of iterations, but for elsewhere we use $\alpha = 1$. As shown by the results, DDLR-1 achieved convergence for all the cases, whereas the other three

preconditioners all had a few failures. Similar to the experimental results for the model problems, the DDLR preconditioners required more time to construct. Compared with pARMS and RAS, DDLR-1 achieved time savings in the iteration phase for 7 (out of 12) problems and DDLR-2 did so for 4 cases.

Table 5.5: Comparison between the DDLR, pARMS and RAS preconditioners for solving general symmetric linear systems with CG or GMRES(40).

Matrix	Np	DDLR-1					DDLR-2				
		rk	nz	its	p-t	i-t	rk	nz	its	p-t	i-t
Andrews	8	8	4.7	33	.587	.220	8	5.2	53	.824	.175
Dubcova2	8	16	3.5	18	.850	.054	16	4.5	44	.856	.079
cfd1	8	8	18.1	17	7.14	.446	8	18.4	217	6.44	2.97
thermal1	8	16	6.0	48	.493	.145	16	8.3	126	.503	.234
cfd2	16	8	13.2	12	4.93	.232	8	13.4	F	5.11	-
Dubcova3	16	16	2.6	16	1.70	.061	16	3.2	44	1.71	.107
thermo_TK	16	32	6.4	24	.568	.050	32	10.8	63	.537	.096
para_fem	16	32	7.8	59	4.02	.777	32	12.3	159	4.12	1.35
tmt_sym	16	16	7.3	33	5.56	.668	16	9.5	62	5.69	.790
ecology2	32	32	8.9	39	3.67	.433	32	15.2	89	3.79	.709
ecology1	32	32	8.8	40	3.48	.423	32	15.1	82	3.59	.656
thermal2	32	32	6.8	140	5.06	2.02	32	11.3	F	5.11	-

Matrix	Np	pARMS				RAS			
		nz	its	p-t	i-t	nz	its	p-t	i-t
Andrews	8	4.3	15	.217	.109	3.6	19	.010	.073
Dubcova2	8	3.5	25	.083	.090	3.5	43	.008	0.11
cfd1	8	16.1	F	.091	-	10.6	153	.013	3.55
thermal1	8	5.4	39	.089	.153	4.6	156	.006	.235
cfd2	16	26.0	F	.120	-	11.9	310	.012	3.26
Dubcova3	16	2.6	37	.130	.200	4.2	39	.013	.212
thermo_TK	16	4.9	16	.048	.035	5.5	34	.004	.067
para_fem	16	6.5	89	.586	1.36	5.1	247	.019	1.18
tmt_sym	16	6.9	16	.587	.361	3.7	26	.026	.222
ecology2	32	9.9	15	.662	.230	5.8	28	.017	.165
ecology1	32	10.0	14	.664	.220	5.8	27	.017	.161
thermal2	32	6.1	205	.547	3.70	4.7	F	.025	-

5.5 Summary and discussion

This chapter has presented a preconditioning method for solving distributed symmetric sparse linear systems, based on an approximate inverse of the original matrix that exploits the Sherman-Morrison-Woodbury (SMW) formula and low-rank approximations. Two low-rank approximation strategies were discussed, yielding two variants of this method, namely DDLR-1 and DDLR-2. The DDLR method extends the idea of the MLR method introduced in the previous chapter via the standard domain decomposition (DD) approach. A difference between these two methods is that the DDLR preconditioner is not recursive. As a result, this method is much easier to implement.

Experimental results indicate that for SPD problems, the DDLR-1 preconditioner can be an efficient alternative to other standard DD-type approaches such as the pARMS method that is based on the distributed Schur complement or the RAS method. Moreover, this preconditioner appears to be more robust than the pARMS method and the RAS method for indefinite problems. On the other hand, the DDLR-2 method that uses low-rank approximations for more terms has a simpler form and is less expensive to apply, but this method is less accurate. In general, it does not work well compared with the DDLR-1 method.

Building the DDLR preconditioners is much more expensive than the standard DD-based methods. Some improvements can be made to reduce the set-up time. For example, more efficient local solvers can be used instead of the current ILUs; and more efficient algorithms than the Lanczos method, e.g., randomized techniques [162], can be exploited for computing the extreme eigenvalues and vectors. Also, we should also take into account the mitigating factors pointed out in Section 4.12.

Chapter 6

Schur complement based domain decomposition preconditioners with low-rank corrections

6.1 Introduction

In this chapter, we extend the preconditioning methods based on low-rank approximations to the classical Schur complement techniques with domain decomposition (DD) approaches and propose a Schur complement based low-rank correction preconditioner that is referred to as the SLR preconditioner [163], for solving general sparse linear systems. In a nutshell, the idea of the SLR method is that if the difference between the inverse of the Schur complement and the inverse of the interface matrix, i.e., the matrix associated with the interface unknowns resulting from DD, can be well approximated by a low-rank matrix, we can directly obtain an approximate inverse of the Schur complement by the inverse of the interface matrix with a low-rank correction. We first introduce the SLR method for symmetric positive definite (SPD) matrices and symmetric indefinite matrices if the interface matrices are SPD. This assumption usually holds for matrices arising from discretization of partial differential equations (PDEs). Then, we present a multilevel scheme aimed at large scale problems. The extensions to general symmetric indefinite matrices as well as to nonsymmetric matrices are also discussed.

6.2 Low-rank corrections based on Schur complements

The SLR preconditioner is directly applicable to the class of linear systems that arise in standard DD methods, including the vertex-based and the edge-based partitionings. In the SLR method, we observed that there is often *a decay property* when approximating the inverse of the Schur complement by the inverse of a close-by matrix in other contexts. By this we mean that the difference between the two inverses has very rapidly decaying eigenvalues, which makes it possible to approximate this difference by small-rank matrices. The best framework where this property takes place is that of the DD-based Schur complement approach discussed in Section 2.4.4 with the block factorization

$$A = \begin{pmatrix} B & E \\ E^T & C \end{pmatrix} = \begin{pmatrix} I & \\ E^T B^{-1} & I \end{pmatrix} \begin{pmatrix} B & E \\ & S \end{pmatrix}, \quad \text{with} \quad S = C - E^T B^{-1} E, \tag{6.1}$$

where $B = \operatorname{diag}(B_1, B_2, \dots, B_p)$ corresponds to the interior unknowns of the decoupled subdomains, C represents the global interface, and $S \in \mathbb{R}^{s \times s}$ is the corresponding Schur complement. In multilevel ILU preconditioners, e.g., [62, 85, 86, 88], approximations to S are formed by dropping small terms and then ILU factorizations of S are computed. In contrast, the SLR method *approximates the inverse of S directly* by the sum of C^{-1} and a low-rank correction term, resulting in improved robustness for indefinite problems. Details on the low-rank property for $S^{-1} - C^{-1}$ will be discussed in the next section.

6.3 Spectral analysis

In this section, we study the fast eigenvalue decay property of $S^{-1} - C^{-1}$. In other words, our goal is to show that $S^{-1} \approx C^{-1} + \text{LRC}$, where LRC stands for a low-rank correction matrix.

6.3.1 Decay properties of $\mathbf{S^{-1} - C^{-1}}$

Assuming that the matrix C in (6.1) is SPD and $C = LL^T$ is its Cholesky factorization, then we can write

$$S = L\left(I - L^{-1} E^T B^{-1} E L^{-T}\right) L^T \equiv L(I - H)L^T. \tag{6.2}$$

Consider now the spectral factorization of $H \in \mathbb{R}^{s \times s}$

$$H = L^{-1}E^T B^{-1} E L^{-T} = U \Lambda U^T, \tag{6.3}$$

where U is unitary, and $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_s)$ is the diagonal matrix of eigenvalues. When A is SPD, then H is at least symmetric positive semidefinite (SPSD) and the following lemma shows that the eigenvalues λ_i's are all less than one.

Lemma 6.3.1 *Let $H = L^{-1}E^T B^{-1} E L^{-T}$ and assume that A is SPD. Then we have $0 \leq \lambda_i < 1$, for each eigenvalue λ_i of H, $i = 1, \ldots, s$.*

Proof. If A is SPD, then B, C and S are all SPD. Since an arbitrary eigenvalue $\lambda(H)$ of H satisfies

$$\lambda(H) = \lambda(C^{-1}E^T B^{-1} E) = \lambda(C^{-1}(C - S)) = 1 - \lambda(C^{-1}S) < 1,$$

and H is at least SPSD, we have $0 \leq \lambda_i < 1$. □

From (6.2), we know that the inverse of S reads

$$S^{-1} = L^{-T}(I - H)^{-1} L^{-1}. \tag{6.4}$$

Thus, we wish to show that *the matrix $(I-H)^{-1}$ can be well approximated by an identity matrix plus a low rank matrix*, from which it would follow that $S^{-1} \approx C^{-1} + \text{LRC}$ as desired. We have the following relations,

$$(I - H)^{-1} - I = L^T S^{-1} L - I = L^T (S^{-1} - C^{-1}) L \equiv X, \tag{6.5}$$

and thus we obtain

$$S^{-1} = C^{-1} + L^{-T} X L^{-1}. \tag{6.6}$$

Note that the eigenvalues of X are the same as those of the matrix $S^{-1}C - I$. Thus, we will ask the question: Can X be well approximated by a low rank matrix? The answer can be found by examining the decay properties of the eigenvalues of X, which in turn can be assessed by checking the rate of change of the large eigenvalues of X. We can state the following result.

Lemma 6.3.2 *The matrix X in (6.5) has the nonnegative eigenvalues $\theta_k = \lambda_k/(1-\lambda_k)$ for $k = 1, \cdots, s$, where λ_k is the eigenvalue of the matrix H in* (6.3).

Proof. From (6.5) the eigenvalues of the matrix X are $(1-\lambda_k)^{-1}-1=\lambda_k/(1-\lambda_k)$. These are nonnegative because the λ_k's are between 0 and 1 from Lemma 6.3.1. □

Now we consider the derivative of θ_k with respect to λ_k:

$$\frac{d\theta_k}{d\lambda_k}=\frac{1}{(1-\lambda_k)^2}\ .$$

This indicates a rapid increase when λ_k increases toward one. In other words, this means that the largest eigenvalues of X tend to be well separated and X can be approximated accurately by a low-rank matrix in general. We were initially interested in the difference between $S^{-1}-C^{-1}$, i.e., $L^{-T}XL^{-1}$, at the beginning of Section 6.3. The following results show that the eigenvalues of $S^{-1}-C^{-1}$ are related to those of X by the largest and the smallest eigenvalues of C, where we use $\lambda_1(\cdot)\geq\cdots\geq\lambda_n(\cdot)$ (ordered if they are real) to denote the eigenvalues of a matrix.

Theorem 6.3.1 (Lidskiĭ [164, p. 248]) *If $G,H\in\mathbb{R}^{n\times n}$ are positive semidefinite symmetric and $1\leq i_1<\cdots<i_k\leq n$, then*

$$\prod_{t=1}^{k}\lambda_{i_t}(GH)\leq\prod_{t=1}^{k}\lambda_{i_t}(G)\lambda_t(H) \tag{6.7}$$

with equality for $k=n$.

We have the following result for the eigenvalues of $S^{-1}-C^{-1}$.

Proposition 6.3.1 *The eigenvalues of $S^{-1}-C^{-1}\in\mathbb{R}^{s\times s}$ and those of $X\in\mathbb{R}^{s\times s}$ satisfy the following inequality.*

$$\lambda_i(X)\lambda_1(C)^{-1}\leq\lambda_i\left(S^{-1}-C^{-1}\right)\leq\lambda_i(X)\lambda_s(C)^{-1},\quad i=1,2,\ldots,s$$

Proof. Note that the eigenvalues of X are the same as those of $Y\equiv(S^{-1}-C^{-1})C$. By (6.7) with $k=1$, we have

$$\lambda_i(X)=\lambda_i(Y)\leq\lambda_i(S^{-1}-C^{-1})\lambda_1(C)\Rightarrow\lambda_i(S^{-1}-C^{-1})\geq\lambda_i(X)\lambda_1(C)^{-1}.$$

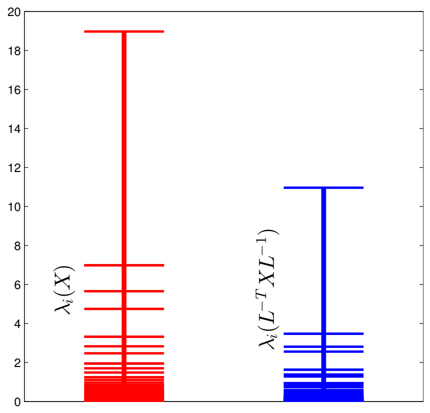
Likewise, the second part is obtained by

$$\lambda_i(S^{-1}-C^{-1})=\lambda_i(YC^{-1})\leq\lambda_i(Y)\lambda_1(C^{-1})=\lambda_i(X)\lambda_s(C)^{-1}.$$

□

Figure 6.1 illustrates the decay of the eigenvalues of $L^{-T}XL^{-1}$ and X for a 2-D Laplacian matrix that is precisely the matrix shown in Figure 2.5. As can be seen, using just a few eigenvalues and vectors will represent the matrix X (or $L^{-T}XL^{-1}$) quite well. In this particular situation, 5 eigenvectors (out of the total of 127) will capture 82.5% of X and 85.1% of $L^{-T}XL^{-1}$, whereas 10 eigenvectors will capture 89.7% of X and 91.4% of $L^{-T}XL^{-1}$.

Figure 6.1: Illustration of the eigenvalues of X (left) and $L^{-T}XL^{-1}$ (right) for a 2-D Laplacian on a 32×32 mesh partitioned into 4 subdomains. The dimension of S is 127. 5 eigenvectors will capture 82.5% of the spectrum of X and 85.1% of the spectrum of $L^{-T}XL^{-1}$, whereas 10 eigenvectors will capture 89.7% of the spectrum of X and 91.4% of the spectrum of $L^{-T}XL^{-1}$.



6.3.2 Two-domain analysis in a 2-D model problem

The spectral analysis of the matrix $S^{-1} - C^{-1}$ is difficult for general problems and general partitionings. In the simplest case when the matrix A originates from a 2-D Laplacian on a regular grid, discretized by centered differences, and it is partitioned into 2 subdomains, the analysis becomes feasible. The goal of this section is to show that the eigenvalues of X and $L^{-T}XL^{-1}$ decay rapidly.

Assume that $-\Delta$ is discretized on a grid Ω of size $n_x \times (2n_y + 1)$ with Dirichlet boundary conditions and that the ordering is major along the x direction. The grid is partitioned horizontally into three parts: the two disconnected $n_x \times n_y$ grids, namely Ω_1

and Ω_2, which are the same, and the $n_x \times 1$ separator denoted by Γ. See Figure 6.2(a) for an illustration. Let T_x be the tridiagonal matrix corresponding to Γ of dimension $n_x \times n_x$ which discretizes $-\partial^2/\partial x^2$. The scaling term $1/h^2$ is omitted so that T_x has the constant 2 on its main diagonal and -1 on the co-diagonals. Finally, we denote by A the matrix which results from discretizing $-\Delta$ on Ω and reordered according to the partitioning $\Omega = \{\Omega_1, \Omega_2, \Gamma\}$. In Ω_1 and Ω_2, the interface nodes are ordered at the end. Hence, A has the form:

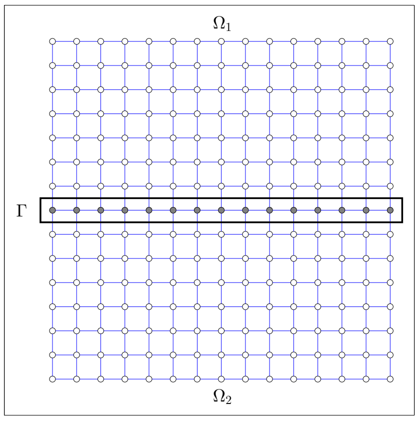
$$A = \begin{pmatrix} A_y & & E_y \\ & A_y & E_y \\ E_y^T & E_y^T & \hat{T}_x \end{pmatrix}, \tag{6.8}$$

where A_y corresponds to the $n_x \times n_y$ grid (i.e., Ω_1 or Ω_2), E_y defines the couplings between Ω_1 (or Ω_2) and Γ, and the matrix $\hat{T}_x$ is associated with Γ, for which we have

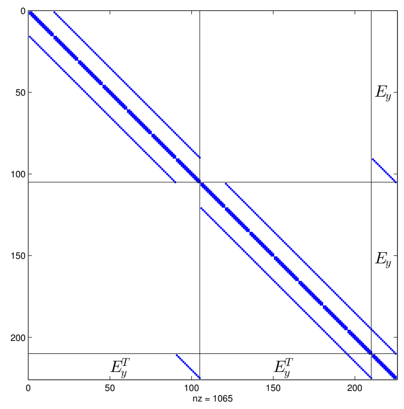
$$\hat{T}_x = T_x + 2I. \tag{6.9}$$

Figure 6.2(b) is an illustration of the nonzero pattern of A.

Figure 6.2: Illustration of the matrix A and the corresponding partitioning of the 2-D mesh.



(a) Partition of a regular mesh into 3 parts.



(b) Nonzero pattern of the reordered matrix.

Therefore, the Schur complement associated with Γ in (6.8) reads

$$S_\Gamma = \hat{T}_x - 2E_y^T A_y^{-1} E_y \ , \tag{6.10}$$

and the eigenvalues of X and $L^{-T}XL^{-1}$ correspond to those of $S_\Gamma^{-1}\hat{T}_x - I$ and $S_\Gamma^{-1} - \hat{T}_x^{-1}$, respectively, in this case. The coupling matrix E_y has the form $E_y^T = (0, I_x)$, where I_x denotes the identity matrix of size n_x. Clearly, the matrix $R_y = E_y^T A_y^{-1} E_y$ is simply the bottom right (corner) block of the inverse of A_y, which can be readily obtained from a standard block factorization. Noting that A_y is of the form

$$A_y = \begin{pmatrix} \hat{T}_x & -I & & & \\ -I & \hat{T}_x & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & \hat{T}_x \end{pmatrix},$$

we write its block LU factorization as:

$$A_y = \begin{pmatrix} I & & & & \\ -D_1^{-1} & I & & & \\ & -D_2^{-1} & \ddots & & \\ & & \ddots & \ddots & \\ & & & -D_{n_y}^{-1} & I \end{pmatrix} \begin{pmatrix} D_1 & -I & & & \\ & D_2 & -I & & \\ & & \ddots & \ddots & \\ & & & \ddots & -I \\ & & & & D_{n_y} \end{pmatrix}.$$

The D_i's satisfy the recurrence: $D_k = \hat{T}_x - D_{k-1}^{-1}$, for $k = 2, \cdots, n_y$, starting with $D_1 = \hat{T}_x$. The result is that each D_k is a continued fraction in $\hat{T}_x$. As can be easily verified R_y is equal to $D_{n_y}^{-1}$. The scalar version of the above recurrence is of the form:

$$d_k = 2a - \frac{1}{d_{k-1}} \ , \quad k = 2, \cdots, n_y \ , \quad \text{with} \quad d_1 \equiv 2a \ .$$

The d_i's are the diagonal entries of the U-matrix of an LU factorization similar to the one above but applied to the $n_y \times n_y$ tridiagonal matrix T that has $2a$ on the diagonal and -1 on the co-diagonals. For reasons that will become clear we replaced the matrix $\hat{T}_x$ by the scalar $2a$. We are interested in the inverse of the last entry, i.e., $d_{n_y}^{-1}$. Using Chebyshev polynomials we can easily see that $d_{n_y}^{-1} = U_{n_y-1}(a)/U_{n_y}(a)$, where $U_k(t)$ is the Chebyshev polynomial of the second kind. This result is shown in the following proposition.

Proposition 6.3.2 *Let*

$$T = \begin{pmatrix} 2a & -1 & & & \\ -1 & 2a & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2a \end{pmatrix}$$

and

$$T = \begin{pmatrix} 1 & & & & \\ -d_1^{-1} & 1 & & & \\ & -d_2^{-1} & \ddots & & \\ & & \ddots & \ddots & \\ & & & -d_n^{-1} & 1 \end{pmatrix} \begin{pmatrix} d_1 & -1 & & & \\ & d_2 & -1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & -1 \\ & & & & d_n \end{pmatrix}$$

be the LU factorization of T. We have

$$d_n^{-1} = \frac{U_{n-1}(a)}{U_n(a)},$$

where $U_k(t)$ is the Chebyshev polynomial of the second kind

$$U_k(t) = \frac{\sinh\big((k+1)\cosh^{-1}(t)\big)}{\sinh\big(\cosh^{-1}(t)\big)}.$$

Proof. If we solve $Tx = e_n$, where e_n is the n-th canonical basis vector for $\mathbb{R}^n$, and $x = [\xi_0, \cdots, \xi_{n-1}]^T$, then clearly $\xi_{n-1} = 1/d_n$, which is what we need to calculate. Let $\xi_k = U_k(a)$, for $k = 0, 1, \cdots, n-1$, where U_k is the k-th degree Chebyshev polynomial of the second kind. These polynomials satisfy the recurrence: $U_{k+1}(t) = 2tU_k(t) - U_{k-1}(t)$, starting with $U_0(t) = 1$ and $U_1(t) = 2t$. Then clearly, equations $k = 1, \cdots, n-1$ of the system $Tx = e_n$ are satisfied. For the last equation we get $U_n(a)$ instead of the wanted value of 1. Scaling x by $U_n(a)$ yields the result $1/d_n = \xi_{n-1} = U_{n-1}(a)/U_n(a)$. □

In terms of the original matrix A_y, the scalar a needs to be substituted by $\hat{T}_x/2 = I + T_x/2$. In the end, the matrix $S^{-1} - C^{-1} = S_\Gamma^{-1} - \hat{T}_x^{-1}$ is a rational function of $\hat{T}_x/2$. We denote this rational function by $s(t)$, i.e., $S_\Gamma^{-1} - \hat{T}_x^{-1} = s(\hat{T}_x/2)$ and note that s is well-defined in terms of the scalar a. Indeed, from the above:

$$s(a) = \frac{1}{2a - 2\frac{U_{n_y-1}(a)}{U_{n_y}(a)}} - \frac{1}{2a} = \frac{U_{n_y-1}(a)}{a(2aU_{n_y}(a) - 2U_{n_y-1}(a))} = \frac{U_{n_y-1}(a)}{a\left[U_{n_y+1}(a) - U_{n_y-1}(a)\right]}.$$

Everything can now be expressed in terms of the eigenvalues of $\hat{T}_x/2$ which are

$$\eta_k = 1 + 2\sin^2 \frac{k\pi}{2(n_x+1)} \ , \quad k = 1, \cdots, n_x \ . \tag{6.11}$$

We can then state the following.

Proposition 6.3.3 *Let η_k be defined in (6.11) and $\theta_k = \cosh^{-1}(\eta_k)$, $k = 1, \cdots, n_x$. Then, the eigenvalues γ_k of $S_\Gamma^{-1} - \hat{T}_x^{-1}$ are given by*

$$\gamma_k = \frac{\sinh(n_y\theta_k)}{\eta_k\left[\sinh((n_y+2)\theta_k) - \sinh(n_y\theta_k)\right]} \ , \quad k = 1, \cdots, n_x \ . \tag{6.12}$$

Note that we have $e^{\theta_k} = \eta_k + \sqrt{\eta_k^2 - 1}$ and $\sinh(n\theta_k) = [(\eta_k + \sqrt{\eta_k^2-1})^n - (\eta_k + \sqrt{\eta_k^2-1})^{-n}]/2$, which is well approximated by $(\eta_k + \sqrt{\eta_k^2-1})^n/2$ for a large n. In the end, assuming n_y is large enough, we have

$$\gamma_k \approx \frac{1}{\eta_k\left[(\eta_k + \sqrt{\eta_k^2-1})^2 - 1\right]} = \frac{1}{2\eta_k\left[(\eta_k^2 - 1) + \eta_k\sqrt{\eta_k^2-1}\right]} \ . \tag{6.13}$$

This shows that for those eigenvalues of $\hat{T}_x$ that are close to one, we would have a big amplification to the value $1/\eta_k$. These eigenvalues correspond to the smallest eigenvalues of T_x. We can also show that

$$\gamma_k \approx \frac{1}{2}\left[\frac{1}{\sqrt{\eta_k^2-1}} - \frac{1}{\eta_k}\right],$$

while for the eigenvalues ζ_k of $S_\Gamma^{-1}\hat{T}_x - I$, we have

$$\zeta_k = 2\eta_k\gamma_k \approx \frac{\eta_k}{\sqrt{\eta_k^2-1}} - 1 \ .$$

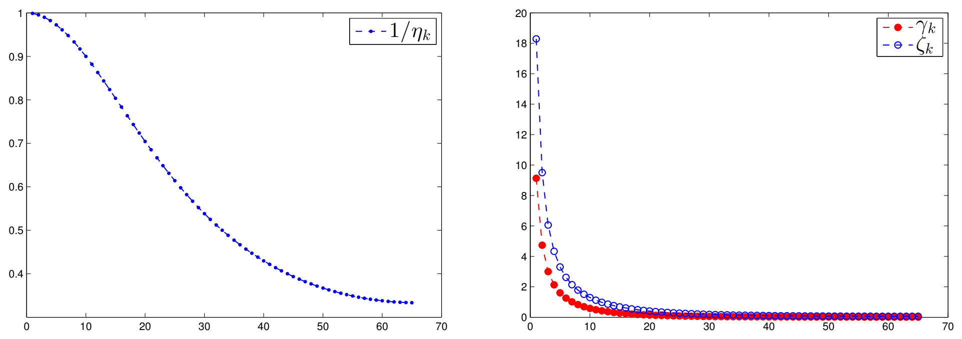
An illustration of γ_k, ζ_k and $1/\eta_k$ is shown in Figure 6.3.

6.4 Schur complement based preconditioning with low-rank corrections

The goal of this section is to build a preconditioner for matrix A in the form (6.1) obtained from the DD method. The preconditioning matrix M is of the form

$$M = \begin{pmatrix} I & \\ E^TB^{-1} & I \end{pmatrix} \begin{pmatrix} B & E \\ & \tilde{S} \end{pmatrix}, \tag{6.14}$$

Figure 6.3: Illustration of the decay of the eigenvalues γ_k of the matrix $S^{-1} - C^{-1}$ and the eigenvalues ζ_k of the matrix $S^{-1}C - I$, and $1/\eta_k$ for $-\Delta$ on a 2-D grid of size $n_x \times (2n_y + 1)$ with $n_x = 65, n_y = 32$, which is partitioned into 2 subdomains.



where $\tilde{S}$ is an approximation to S. The above is approximate factorization of (6.1) whereby (only) S is approximated. In fact we will approximate directly the inverse of S instead of S by exploiting low-rank properties. Specifically, we seek an approximation of the form $\tilde{S}^{-1} = C^{-1} + \text{LRC}$, where LRC stands for a low-rank correction matrix. From a practical point of view, it will be difficult to compute directly an approximation to the matrix $S^{-1} - C^{-1}$, since S^{-1} is not available and we do not (yet) have an efficient means for solving linear systems with the matrix S. Instead we will extract this approximation from that of the matrix X defined in Section 6.3.1, see (6.5). Recall the expression (6.2) and the eigen-decomposition of H in (6.3), which yield,

$$S = L(I - U\Lambda U^T)L^T = LU(I - \Lambda)U^T L^T. \tag{6.15}$$

The inverse of S is then

$$S^{-1} = L^{-T}U(I - \Lambda)^{-1}U^T L^{-1},$$

which we write in the form,

$$\begin{aligned} S^{-1} &= L^{-T}\left(I + U[(I - \Lambda)^{-1} - I]U^T\right)L^{-1}, \\ &= C^{-1} + L^{-T}U[(I - \Lambda)^{-1} - I]U^T L^{-1}. \end{aligned}$$

Now, assuming that H has an approximation of the following form,

$$\tilde{H} \approx U\tilde{\Lambda}U^T, \quad \tilde{\Lambda} = \text{diag}(\tilde{\lambda}_1, \ldots, \tilde{\lambda}_s), \tag{6.16}$$

we will obtain the following approximation to S^{-1}:

$$\tilde{S}^{-1} = L^{-T}U(I-\tilde{\Lambda})^{-1}U^TL^{-1}, \tag{6.17}$$

$$= C^{-1} + L^{-T}U[(I-\tilde{\Lambda})^{-1} - I]U^TL^{-1}. \tag{6.18}$$

Proposition 6.4.1 *Let S and H be defined by (6.2) and (6.3) respectively and let $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_s)$ with the σ_i's defined by*

$$\sigma_i = \frac{1-\lambda_i}{1-\tilde{\lambda}_i}\ , \quad i = 1, \ldots, s. \tag{6.19}$$

Then, the eigendecomposition of $S\tilde{S}^{-1}$ is given by:

$$S\tilde{S}^{-1} = (LU)\Sigma(LU)^{-1}. \tag{6.20}$$

Proof. From (6.15) and (6.17), we have

$$\begin{aligned} S\tilde{S}^{-1} &= LU(I-\Lambda)U^TL^TL^{-T}(U(I-\tilde{\Lambda})^{-1}U^T)L^{-1} \\ &= (LU)(I-\Lambda)(I-\tilde{\Lambda})^{-1}(U^TL^{-1}) = (LU)\Sigma(LU)^{-1}. \end{aligned}$$

□

The simplest selection of $\tilde{\Lambda}$ is the one that ensures that the k largest eigenvalues of $(I-\tilde{\Lambda})^{-1}$ match the largest eigenvalues of $(I-\Lambda)^{-1}$. Assume that the eigenvalues of H are $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_s$, which means that the diagonal entries $\tilde{\lambda}_i$ of $\tilde{\Lambda}$ are selected such that

$$\tilde{\lambda}_i = \begin{cases} \lambda_i & \text{if} \quad i \leq k \\ 0 & \text{otherwise} \end{cases}. \tag{6.21}$$

Proposition 6.4.1 indicates that in this case the eigenvalues of $S\tilde{S}^{-1}$ are

$$\begin{cases} 1 & \text{if} \quad i \leq k \\ 1-\lambda_i & \text{otherwise} \end{cases}.$$

Thus, we can infer that in this situation k eigenvalues of $S\tilde{S}^{-1}$ will take the value one and the other $s-k$ eigenvalues σ_i satisfy $0 < 1-\lambda_{k+1} \leq \sigma_i < 1-\lambda_s < 1$.

Another choice for $\tilde{\Lambda}$ will make the eigenvalues of $S\tilde{S}^{-1}$ larger than or equal to one. Consider defining $\tilde{\Lambda}$ such that

$$\tilde{\lambda}_i = \begin{cases} \lambda_i & \text{if} \quad i \leq k \\ \theta & \text{if} \quad i > k \end{cases}. \tag{6.22}$$

Then, from (6.19) the eigenvalues of $S\tilde{S}^{-1}$ are

$$\begin{cases} 1 & \text{if} \quad i \le k \\ (1-\lambda_i)/(1-\theta) & \text{if} \quad i > k \end{cases} . \tag{6.23}$$

The earlier definition of Λ_k in (6.21) which truncates the lowest eigenvalues of H to zero corresponds to selecting $\theta = 0$. This is essentially an *eigenvalue deflation* scheme to matrix SC^{-1}. Note that the eigenvalues of SC^{-1} are the same as those of $I - H$, which are $1 - \lambda_i$. In the above scheme, the first k eigenvalues of SC^{-1} are moved to 1 and the others are scaled by $1/(1 - \theta)$. For $i > k$, the eigenvalues can be made greater than or equal to one by selecting $\lambda_{k+1} \le \theta < 1$. In this case, the eigenvalues σ_i for $i > k$ which are equal to $\sigma_i = (1 - \lambda_i)/(1 - \theta)$ belong to the interval

$$\left[1, \quad \frac{1-\lambda_s}{1-\theta}\right] \subseteq \left[1, \quad \frac{1}{1-\theta}\right] . \tag{6.24}$$

Thus, the spectral condition number of the preconditioned matrix is $(1 - \lambda_s)/(1 - \theta)$. The choice leading to the smallest 2-norm deviation is letting $\theta = \lambda_{k+1}$. One question that may be asked is how does the condition number $\kappa = \max \sigma_i / \min \sigma_i$ vary when θ varies between 0 and 1?

First observe that a general expression for the eigenvalues of $S\tilde{S}^{-1}$ is given by (6.23) regardless of the value of θ. When $\lambda_{k+1} \le \theta < 1$, we just saw that the spectral condition number is equal to $(1 - \lambda_s)/(1 - \theta)$. The smallest value of this condition number is reached when θ takes the smallest value which, recalling our restriction $\lambda_{k+1} \le \theta < 1$, is $\theta = \lambda_{k+1}$. There is a second situation, which corresponds to when $\lambda_s \le \theta \le \lambda_{k+1}$. Here the largest eigenvalue is still $(1 - \lambda_s)/(1 - \theta)$ which is larger than one. The smallest one is now smaller than one, which is $(1 - \lambda_{k+1})/(1 - \theta)$. So the condition number now is again $(1 - \lambda_s)/(1 - \lambda_{k+1})$, which is independent of θ in the interval $[\lambda_s, \ \lambda_{k+1}]$. The third and final situation corresponds to the case when $0 \le \theta \le \lambda_s$. The largest eigenvalue is now one, because $(1 - \lambda_s)/(1 - \theta) < 1$, while the smallest one is still $(1 - \lambda_{k+1})/(1 - \theta)$. This leads to the condition number $(1 - \theta)/(1 - \lambda_{k+1})$ and the smallest spectral condition number for θ in this interval is reached when $\theta = \lambda_s$ leading to the same optimal condition number $(1 - \lambda_s)/(1 - \lambda_{k+1})$. This result is summarized in the following proposition.

Proposition 6.4.2 *The spectral condition number $\kappa(\theta)$ of $S\tilde{S}^{-1}$ is equal to*

$$\kappa(\theta) = \begin{cases} \dfrac{1-\theta}{1-\lambda_{k+1}} & \text{if } \theta \in [0, \lambda_s) \\ \dfrac{1-\lambda_s}{1-\lambda_{k+1}} & \text{if } \theta \in [\lambda_s, \lambda_{k+1}] \\ \dfrac{1-\lambda_s}{1-\theta} & \text{if } \theta \in (\lambda_{k+1}, 1) \end{cases} . \tag{6.25}$$

It has a minimum value of $(1-\lambda_s)/(1-\lambda_{k+1})$, which is reached for any θ in the second interval.

Figure 6.4: Illustration of the condition number $\kappa(\theta)$ for the case of a 2-D Laplacian matrix with $n_x = n_y = 256$ and the number of the subdomains $p = 2$, where 64 eigenvectors are used ($i.e., k = 64$). $\lambda_s = .05719$, $\lambda_{k+1} = .36145$, and the optimal condition number is $\kappa = 1.4765$.

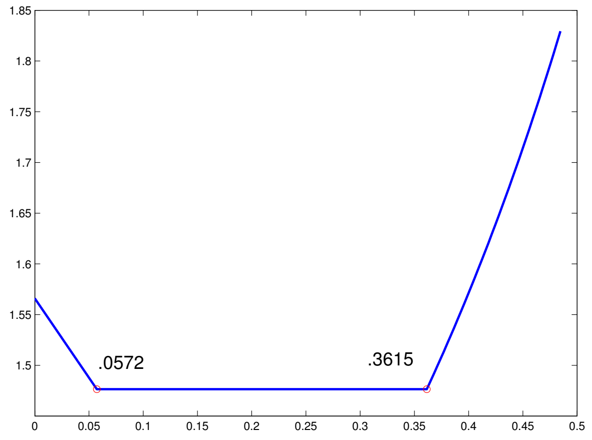
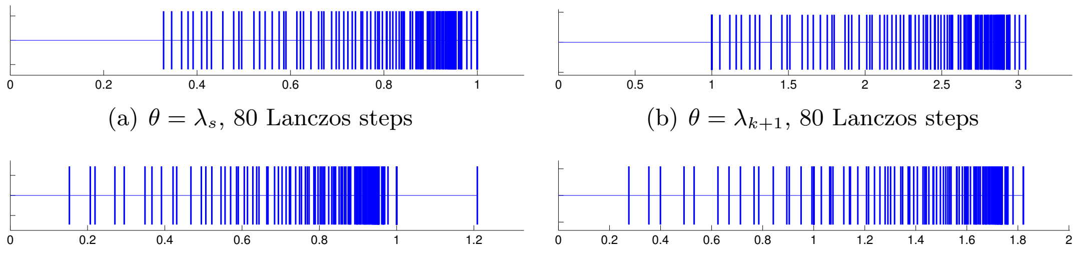


Figure 6.4, shows the variation of the condition number $\kappa(\theta)$ as a function of θ, for a 2-D Laplacian matrix. One may conclude from this result that there is no reason for selecting a particular $\theta \in [\lambda_s, \lambda_{k+1}]$ over another one as long as θ belongs to the middle interval, since the spectral condition number $\kappa(\theta)$ is the same. In fact, in practice when approximate eigenpairs are used, that are computed, for example, by the Lanczos procedure, the choice $\theta = \lambda_{k+1}$ often gives better performance than $\theta = \lambda_s$ in this context because for the former choice, the perturbed eigenvalues are less likely to be close to zero. An example can be found in Figure 6.5, which shows that when using

accurate enough eigenpairs, both choices of θ will give the same condition number (which is also the optimal one), whereas when relatively inaccurate eigenpairs are used, setting $\theta = \lambda_{k+1}$ can give a better condition number than that obtained from setting $\theta = \lambda_s$. In what follows, we assume that the approximation scheme (6.22) is used with $\theta = \lambda_{k+1}$, and we will denote by $S_{k,\theta}^{-1}$ the related approximate inverse of S.

Figure 6.5: Illustration of the eigenvalues of $S\tilde{S}^{-1}$ for the case of a 2-D Laplacian matrix with $n_x = n_y = 128$, the number of subdomains $p = 2$ and the rank $k = 16$, such that the optimal spectral condition number $\kappa(\theta) = 3.0464$, for $\lambda_s \le \theta \le \lambda_{k+1}$. The two top figures show the eigenvalues of $S\tilde{S}^{-1}$ with the Ritz values and vectors from 80 steps of the Lanczos iterations, where for both choices $\theta = \lambda_s$ and $\theta = \lambda_{k+1}$, $\kappa(\theta) = 3.0464$. The bottom two figures show the eigenvalues of $S\tilde{S}^{-1}$ in the cases with 32 steps of the Lanczos iterations, where $\kappa(\lambda_s) = 7.8940$ while $\kappa(\lambda_{k+1}) = 6.6062$.



(a) $\theta = \lambda_s$, 80 Lanczos steps

(b) $\theta = \lambda_{k+1}$, 80 Lanczos steps

(c) $\theta = \lambda_s$, 32 Lanczos steps

(d) $\theta = \lambda_{k+1}$, 32 Lanczos steps

From an implementation point of view, it is clear that only the k largest eigenvalues and the associated eigenvectors as well as the $(k+1)$-st largest eigenvalue of the matrix $C^{-1}E^TB^{-1}E$ are needed. We prove this result in the following proposition.

Proposition 6.4.3 *Let Z_k be the matrix whose column vectors are eigenvectors of $C^{-1}E^TB^{-1}E$ associated with the k largest eigenvalues, and let $\theta = \lambda_{k+1}$. Then the following expression for $S_{k,\theta}^{-1}$ holds:*

$$S_{k,\theta}^{-1} = \frac{1}{1-\theta}C^{-1} \ + \ Z_k \left[(I-\Lambda_k)^{-1} - (1-\theta)^{-1}I\right] Z_k^T. \qquad (6.26)$$

Proof. We write $U = [U_k, W]$, where $U_k = [u_1, \ldots, u_k]$ contains the eigenvectors of H associated with the largest k eigenvalues and W contains the remaining columns $u_{k+1}, \cdots, u_s$. Note that W is not available but we use the fact that $WW^T = I - U_kU_k^T$

for the purpose of this proof. With this, (6.18) becomes:

$$\begin{aligned}
S_{k,\theta}^{-1} &= C^{-1} + L^{-T}[U_k, W] \begin{pmatrix} (I-\Lambda_k)^{-1} - I & 0 \\ 0 & ((1-\theta)^{-1}-1)I \end{pmatrix} [U_k, W]^T L^{-1} \\
&= C^{-1} + Z_k \left[(I-\Lambda_k)^{-1} - I\right] Z_k^T + \left[(1-\theta)^{-1} - 1\right] L^{-T} W W^T L^{-1} \\
&= C^{-1} + Z_k \left[(I-\Lambda_k)^{-1} - I\right] Z_k^T + \left[(1-\theta)^{-1} - 1\right] L^{-T} (I - U_k U_k^T) L^{-1} \\
&= \frac{1}{1-\theta} C^{-1} + Z_k \left[(I-\Lambda_k)^{-1} - (1-\theta)^{-1} I\right] Z_k^T .
\end{aligned}$$

□

In a paper describing a similar technique, Grigori et al. [165], suggest another choice of $\tilde{\Lambda}$ which is:

$$\tilde{\lambda}_i = \begin{cases} 1 - (1-\lambda_i)/\varepsilon & \text{if} \quad i \le k \\ 0 & \text{otherwise} \end{cases} , \tag{6.27}$$

where ε is a parameter. Then the eigenvalues σ_i's are

$$\begin{cases} \varepsilon & \text{if} \quad i \le k \\ 1-\lambda_i & \text{otherwise} \end{cases} ,$$

Note that the first choice in (6.21) is a special case of (6.27) when $\varepsilon = 1$. Writing the transformed eigenvalues as

$$\{\varepsilon, 1-\lambda_{k+1}, 1-\lambda_{k+2}, \cdots, 1-\lambda_s\},$$

the authors stated that the resulting condition number is $\kappa = (1-\lambda_s)/\varepsilon$, with an implied assumption that $\varepsilon \le 1-\lambda_{k+1}$. In all cases, when $1-\lambda_{k+1} < \varepsilon \le 1-\lambda_s$, the spectral condition number is the same as above, i.e., equal to $(1-\lambda_s)/(1-\lambda_{k+1})$. On the other hand, when $0 \le \varepsilon \le 1-\lambda_{k+1}$, then the condition number is now $(1-\lambda_s)/\varepsilon$, and the best value will be reached again for $\varepsilon = 1-\lambda_{k+1}$, which leads to the same condition number as above.

In all cases, if we want the spectral condition number of the matrix $S\tilde{S}^{-1}$, which is $\kappa = (1-\lambda_s)/(1-\lambda_{k+1})$, to be bounded from above by a constant K, we can only guarantee this by having k large enough so that $1/(1-\lambda_{k+1}) \le K$, or equivalently, $\lambda_{k+1} \le 1 - 1/K$. In other words, we would have to select the rank k large enough such that

$$\lambda_{k+1} \le 1 - \frac{1}{K} . \tag{6.28}$$

Of course, the required rank k depends primarily on the eigenvalue decay of the λ_i's. In general, however, this means that the method will require a sufficient number of eigenvectors to be computed and that this number must be increased if we wish to decrease the spectral condition number to a given value. For problems arising from PDEs, it is expected that in order to keep the spectral condition number constant, k must have to be increased as the problem sizes increase.

6.5 Practical implementation

In this section, we will address the implementation details for computing and applying an SLR preconditioner.

6.5.1 Computing the low-rank approximations

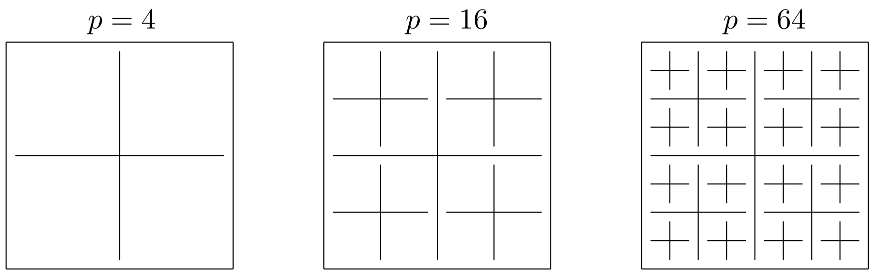
One of the key issues in setting up the preconditioner (6.14) is to extract a low-rank approximation to the matrix $C^{-1}E^TB^{-1}E$. Assuming that C is SPD, we use the Lanczos algorithm [118] on the matrix $L^{-1}E^TB^{-1}EL^{-T}$, where L is the Cholesky factor of C. In the Lanczos algorithm, we need to compute the matrix-vector product of the form $y = L^{-1}E^TB^{-1}EL^{-T}x$, which requires solves with the block diagonal matrix B and the triangular matrix L.

6.5.2 Solves with B and C

A solve with the matrix B amounts to p local and independent solves with the matrices B_i, $i = 1, \cdots, p$. These can be carried out efficiently either by a direct solver or by Krylov subspace methods with more traditional ILU preconditioners for example. On the other hand, the matrix C, associated with all the interface unknowns, often has some diagonal dominance properties for discretized PDEs problems, so that ILU-based methods can typically work well. However, as we will see, this method cannot scale well. For solving large problems, especially ones issued from 3-D PDEs, we need to have a large number of subdomains such that the local solves with B_i can be inexpensive. As a result, the number of interface unknowns will increase rapidly with the problem size and the number of subdomains.

For example, consider a 5-point stencil discretization of the 2-D Laplacian on a regular mesh of size $n_x \times n_x$ and assume a 2-D geometric partitioning is used. As shown in the leftmost side of the illustration in Figure 6.6, when we start with 4 partitions we have about $2n_x$ interface points ($2n_x - 1$ to be exact). Each time we partition further, halving each subdomain in each direction, we multiply the number of subdomains by a factor of 4, and add roughly $2^{k-1}n_x$ interface points in each direction at the k-th division ($k = 1$ corresponds to the initial partitioning on the left side of Figure 6.6. At the k-th division we would have $p = (2^k)^2$ subdomains and $1 + 2 + \cdots 2^{k-1} = 2^k - 1$ lines in each direction, i.e, $\approx 2^{k+1}n_x$ interface points, when k is large. So the number of interface points evolves like $2\sqrt{p}n_x = 2\sqrt{pN}$ where $N = n_x^2$ is the total number of points. For a 3-D mesh of size $n_x \times n_x \times n_x$, with a 3-D partitioning, the number of interface points is about $3(\sqrt{p}N)^{2/3}$, where $N = n_x^3$.

Figure 6.6: A 2-D finite difference mesh recursively partitioned into 64 subdomains



An alternative is to apply the SLR method recursively to C. This requires that the interface unknowns be ordered in a way that C has the same structure as that of A, i.e., that the leading block is block diagonal. This is a property that can be satisfied by the hierarchical interface decomposition (HID) method discussed in [166]. This idea essentially yields a multilevel scheme of the SLR method that was proposed in [167]. We will present this method in the next section.

6.6 Multilevel SLR preconditioners

An interesting class of multilevel DD methods, that exploits a hierarchy of interfaces, is the so-called *wirebasket* orderings [81, 168]. These techniques take advantages of the *cross-points* in the partitioned mesh to derive preconditioners with good convergence properties. This idea was further generalized in [166] for general sparse matrices that are not necessarily originating from PDEs, referred to as the Hierarchical Interface Decomposition (HID). In this section, we first review the concept of the HID method and then will see that the multilevel SLR method can be naturally incorporated into the HID framework.

6.6.1 Hierarchical interface decomposition

The HID method is defined through the notion of *connectors*. For a graph $\mathcal{G}$, we call a connector a (connected) subgraph of $\mathcal{G}$. Connectors are grouped into *levels*, labeled from 0 to m. So, a level is simply a set of connectors. The HID is defined as follows.

Definition 6.6.1 ([166]) *A set of levels labeled from* 0 *to* m *is a HID of a graph* $\mathcal{G}$*, if it satisfies the following conditions:*

1. *The connectors are disjoint and altogether they form a partition of* $\mathcal{G}$*,*
2. *Connectors of the same level are not adjacent,*
3. *Connectors at level* l *are separators for connectors at level* $l-1$*. More precisely, the removal of a connector of level* l *(from the subgraph of* $\mathcal{G}$ *consisting of the connectors of levels* l *and* $l-1$*) will disconnect at least two connector at level* $l-1$*.*

An HID ordering can be obtained in a number of ways, see [166] for an example via standard graph partitionings. It can also be obtained from the nested dissection (ND) algorithm [169]. Let $\mathcal{G} = (V, E)$ be the adjacency graph of matrix A. The basic idea of an ND ordering is to recursively partition the graph using *vertex separators*. Recall that a vertex separator S is a subset of V such that by removing S from $\mathcal{G}$ the remaining graph, $\mathcal{G} \setminus S$, has at least two connected components. In Figure 6.7, we show an example of the ND ordering with 4 levels (i.e., $m = 3$). In this example, the connector at level 3, indicated by the curve labeled 15 in Figure 6.7(a), is a vertex separator of $\mathcal{G}$ that separates the graph into two disconnected subgraphs. At level 2, we have two connectors that are the vertex separators of these two subgraphs. These two connectors are labeled 13 and 14. The same procedure is repeated for the 4 connectors at level 1. Finally, level 0 consists of the 8 connectors corresponding to the interior nodes of the 8 partitioned subdomains. The level information of these connectors can be represented by an HID tree shown in Figure 6.7(b). If we reorder the matrix A by the levels from 0 to 3, then the reordered matrix will have a structure shown in Figure 6.7(c). The desired block diagonal structures are obtained for each level i, $i = 0, 1, \dots m-1$. More formally, the reordered matrix has the following recursive form:

$$A_l = \begin{pmatrix} B_l & E_l \\ E_l^T & C_l \end{pmatrix} \quad \text{with} \quad A_{l+1} \equiv C_l \ , \quad \text{for} \quad l = 0, \dots m-1, \tag{6.29}$$

where each leading block B_l is block diagonal. In this example, B_0, B_1 and B_2 correspond to the connectors labeled 1-8, 9-12 and 13-14, respectively, while $A_3 = C_2$ is associated with the connector labeled 15. When the matrix is reordered in the form (6.29), for each level l, we can explore the low rank property as seen in the SLR method to obtain a preconditioner for A_l. We refer to this method as the Multilevel SLR (MSLR) approach. In the next section, we will discuss the multilevel scheme in this context.

6.6.2 The multilevel framework

For each level l except the last one, suppose that A_l has the block factorization

$$A_l = \begin{pmatrix} B_l & E_l \\ E_l^T & A_{l+1} \end{pmatrix} = \begin{pmatrix} I & \\ E_l^T B_l^{-1} & I \end{pmatrix} \begin{pmatrix} B_l & E_l \\ & S_l \end{pmatrix}, \quad \text{with} \quad S_l = A_{l+1} - E_l^T B_l^{-1} E_l.$$

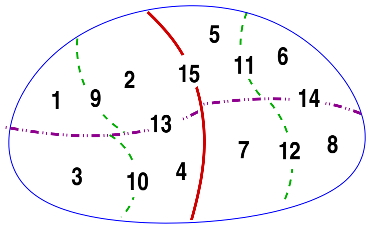
The SLR method with $\theta = 0$ in (6.26) is used for preconditioning A_l. In this method, the inverse of the Schur complement, S_l^{-1}, is approximated by

$$S_l^{-1} \approx A_{l+1}^{-1} + Z_l D_l Z_l^T, \quad \text{with} \quad D_l = \Lambda_l (1 - \Lambda_l)^{-1}, \tag{6.30}$$

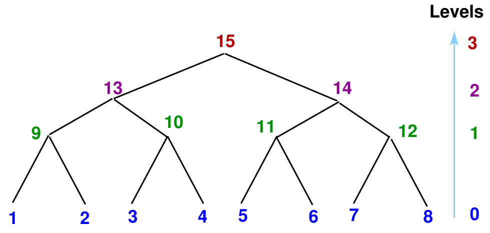
where Λ_l is the diagonal matrix that contains the largest eigenvalues of $C_l^{-1} E_l^T B_l^{-1} E_l$, and Z_l contains the corresponding eigenvectors as columns. For solves with C_l in (6.30), noting that $A_{l+1} = C_l$, we can use the same preconditioning as that for A_l. At the last level, an (incomplete) LU factorization is used for $A_{m-1} = C_{m-2}$. Therefore, an m-level SLR preconditioner can be defined as

$$M_l = \begin{cases} \begin{pmatrix} I & \\ E_l^T B_l^{-1} & I \end{pmatrix} \begin{pmatrix} B_l & E_l \\ & \tilde{S}_l \end{pmatrix}, \ \tilde{S}_l^{-1} = M_{l+1}^{-1} + Z_l D_l Z_l^T, & \text{if} \quad l < m-1, \\ L_{m-1} L_{m-1}^T, & \text{if} \quad l = m-1. \end{cases} \tag{6.31}$$

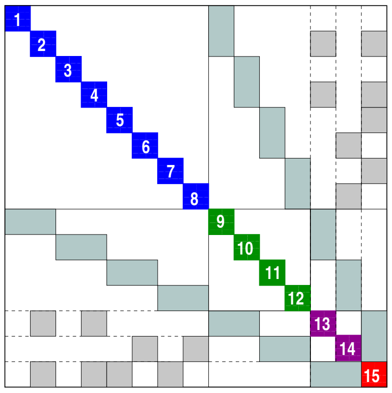
Figure 6.7: An illustration of the HID with 4 levels: the HID tree and the corresponding reordered matrix.



(a) 4-level HID



(b) HID tree of connectors



(c) Structure of the reordered matrix

The application of this preconditioner, $x_l = M_l^{-1} b_l$, can be computed by the recursive function, called `SLRSolve`, shown in Algorithm 9. In particular, $x = \texttt{SLRSolve}(0, m, b)$ performs the preconditioning operation of the MSLR preconditioner. At line 2 of this algorithm, linear systems with the matrix A_m at the last level are solved directly. Lines 7-8 correspond to the approximate inverse (6.30) where linear systems with S_l are solved approximately by applying C_l^{-1} and the low-rank correction.

6.6.3 Computing the low-rank approximations

The Lanczos algorithm can be used for computing the low-rank matrix $Z_l D_l Z_l^T$ in (6.30) as in the SLR method, while two new issues emerge associated with the multilevel framework. The first one is with respect to the order of computing these low-rank matrices at different levels. From (6.31), we can see that applying the preconditioner

Algorithm 9 Function $x_l = \texttt{SLRSolve}(l, m, b_l)$

```
 1: if $l = m - 1$ then
 2:     Solve $A_l x_l = b_l$
 3: else
 4:     Partition $A_l x_l = b_l$ corresponding to the HID:
        $\begin{pmatrix} B_l & E_l \\ E_l^T & A_{l+1} \end{pmatrix} \begin{pmatrix} u_l \\ y_l \end{pmatrix} = \begin{pmatrix} f_l \\ g_l \end{pmatrix}$
 5:     Solve $B_l z = f_l$
 6:     Compute $g_l' = g_l - E_l^T z$
 7:     Recursive call: $v = \texttt{SLRSolve}(l+1,\, m,\, g_l')$
 8:     Compute $y_l = v + Z_l D_l Z_l^T g_l'$
 9:     Solve $B_l u_l = f_l - E_l y_l$
10:     Return $x_l = (u_l^T, y_l^T)^T$
11: end if
```

M_l requires all the preconditioners M_j, for $j > l$. Therefore, the matrices Z_l and D_l must be computed in the order of $l = m-2, m-3, \ldots, 1$. The second issue is that matrix C_l will not be in a factored form except for the one at the last level. But we can still use the Lanczos algorithm to compute the largest eigenvalues and the corresponding eigenvectors for the matrix pair $(E_l^T B_l^{-1} E_l, C_l)$. For the Lanczos algorithm for matrix pairs, see, e.g. [154, §9.2.6]. This algorithm requires solving linear systems with $C_l = A_{l+1}$, which can be computed by performing a few steps of Krylov subspace methods with the available MSLR preconditioner M_{l+1}, or simple just by applying the preconditioner. When the later solution method is used, the eigenpairs computed are only approximations in this sense.

6.7 Extension to general nonsymmetric matrices

Consider the nonsymmetric equivalent of (6.1),

$$A = \begin{pmatrix} B & F \\ E^T & C \end{pmatrix} = \begin{pmatrix} I & \\ E^T B^{-1} & I \end{pmatrix} \begin{pmatrix} B & F \\ & S \end{pmatrix}, \quad \text{with} \quad S = C - E^T B^{-1} F.$$

Let $C = LU$ be the LU factorization of C, so that we have

$$S = L(I - L^{-1} E^T B^{-1} F U^{-1}) U \equiv L(I - H) U, \tag{6.32}$$

as in (6.2). Then, let the *complex* Schur decomposition of H be

$$H = L^{-1}E^TB^{-1}FU^{-1} = WRW^H, \tag{6.33}$$

where W is unitary and R is an upper triangular matrix that contains the eigenvalues of H, denoted by λ_i, on the diagonal. It follows by substituting (6.33) in (6.32) that

$$S = L(I - WRW^H)U = LW(I - R)W^HU, \tag{6.34}$$

and the inverse of S is then

$$S^{-1} = U^{-1}[W(I - R)^{-1}W^H]L^{-1}.$$

Let $\tilde{R}$ be a triangular matrix, that is an approximation to R, with the diagonal entries $\tilde{R}_{i,i} = \tilde{\lambda}_i$, $i = 1, 2, \ldots, s$. Then, an approximate inverse of S can be obtained from

$$\tilde{S}^{-1} = U^{-1}W(I - \tilde{R})^{-1}W^HL^{-1} = U^{-1}\left[I + W\left((I - \tilde{R})^{-1} - I\right)W^H\right]L^{-1}. \tag{6.35}$$

Analogously to Proposition 6.4.1, the following proposition shows the eigenvalues of $S\tilde{S}^{-1}$.

Proposition 6.7.1 *Let S and $\tilde{S}^{-1}$ be given by (6.34) and (6.35) respectively. The Schur decomposition of $L^{-1}S\tilde{S}^{-1}L$ is given by*

$$L^{-1}S\tilde{S}^{-1}L = W\Sigma W^H, \tag{6.36}$$

where $\Sigma = (I - R)(I - \tilde{R})^{-1}$ is an upper triangular matrix that has $\sigma_1, \ldots, \sigma_s$ on the diagonal with σ_i defined by

$$\sigma_i = \frac{1 - \lambda_i}{1 - \tilde{\lambda}_i}\ , \quad i = 1, \ldots, s. \tag{6.37}$$

Proof. By multiplying (6.34) with (6.35), we have

$$S\tilde{S}^{-1} = LW(I - R)W^HUU^{-1}W(I - \tilde{R})^{-1}W^HL^{-1} = LW(I - R)(I - \tilde{R})^{-1}W^HL^{-1}.$$

Therefore,

$$L^{-1}S\tilde{S}^{-1}L = W(I - R)(I - \tilde{R})^{-1}W^H = W\Sigma W^H,$$

with $\sigma_i = (1 - R_{ii})/(1 - \tilde{R}_{ii}) = (1 - \lambda_i)/(1 - \tilde{\lambda}_i)$. $\square$

Clearly, the eigenvalues of $S\tilde{S}^{-1}$ are σ_i, $i = 1, \dots, s$. As before, we let the first k diagonal entries of $\tilde{R}$ match those of R and all the others equal to a constant θ, i.e.,

$$\begin{cases} \tilde{\lambda}_i = \lambda_i & i = 1, \dots, k \\ \tilde{\lambda}_i = \theta & i = k+1, \dots, s \end{cases} . \tag{6.38}$$

With this, the eigenvalues of $S\tilde{S}^{-1}$ become

$$\begin{cases} 1 & i = 1, \dots, k \\ (1-\lambda_i)/(1-\theta) & i = k+1, \dots, s \end{cases} . \tag{6.39}$$

In practice we need not to compute the entire Schur decomposition of H. Only the $k \times k$ leading principal submatrix of R and the first k Schur vectors are needed. This is shown in the following proposition, where we denote by $S_{k,\theta}^{-1}$ the resulting approximate inverse of S.

Proposition 6.7.2 *Let $H = WRW^H$ be the complex Schur decomposition of H. R_k is the $k \times k$ leading principal submatrix of R and W_k have the first k Schur vectors as columns. With $\tilde{R}$ defined by*

$$\tilde{R} = \begin{pmatrix} R_k & \\ & \theta I \end{pmatrix},$$

the following expression for $S_{k,\theta}^{-1}$ holds

$$S_{k,\theta}^{-1} = \frac{1}{1-\theta} C^{-1} \; + \; U^{-1} W_k \left[(I - R_k)^{-1} - (1-\theta)^{-1} I \right] W_k^H L^{-1}. \tag{6.40}$$

Proof. The proof is a straightforward extension of that of Proposition 6.4.1. Let $W = [W_k, \bar{W}]$. From (6.35), we have

$$\begin{aligned} S_{k,\theta}^{-1} &= C^{-1} + U^{-1}[W_k, \bar{W}] \begin{pmatrix} (I-R_k)^{-1} - I & \\ & [(1-\theta)^{-1} - 1]I \end{pmatrix} [W_k, \bar{W}]^T L^{-1} \\ &= C^{-1} + U^{-1} \left(W_k[(I-R_k)^{-1} - I]W_k^T + \theta(1-\theta)^{-1}(I - W_k W_k^T) \right) L^{-1} \\ &= C^{-1} + U^{-1} \left(W_k[(I-R_k)^{-1} - (1-\theta)^{-1}I]W_k^T + \theta(1-\theta)^{-1} I \right) L^{-1} \\ &= \frac{1}{1-\theta} C^{-1} + U^{-1} W_k[(I-R_k)^{-1} - (1-\theta)^{-1}I]W_k^T L^{-1}. \end{aligned}$$

□

6.7.1 Computing the low-rank approximations

We use the Arnoldi process [20] to compute the low-rank matrices in (6.40). When the low-rank approximation is associated with the extreme eigenvalues of the matrix H, i.e., the eigenvalues on the periphery of the spectrum, this computation can be efficient. Performing m, $m > k$, steps of the Arnoldi process on H, leads to the standard Krylov factorization equations:

$$\begin{aligned} HU_m &= U_m H_m + h_{m+1,m} u_{m+1} e_m^T, \\ U_m^T H U_m &= H_m, \end{aligned}$$

where u_i, for $i = 1, \ldots, m+1$, are orthonormal and $U_m = [u_1, \ldots, u_m]$. The eigenvalues of H_m are good estimates of the extreme eigenvalues of H. Let the complex Schur decomposition of H_m be

$$Q^H H_m Q = T. \tag{6.41}$$

Furthermore, the k eigenvalues that we want to deflate can be ordered to the first k entries on the diagonal of T [170,171]. This ordering method is provided by the LAPACK [172] subroutine `DTRSEN`. Thus, the low-rank matrices in (6.40) can be approximated by

$$R_k \approx T_{1:k,1:k} \quad \text{and} \quad W_k \approx U_m Q_{:,1:k} \ .$$

When used in finite precision arithmetic, the basis vectors computed in the Arnoldi procedure with the Gram-Schmidt (G-S) orthogonalization process will lose their orthogonality [173] at some point. A simple remedy is to perform a reorthogonalization. As indicated in [174, 175], only one reorthogonalization step with the classical G-S algorithm is needed to preserve the orthogonality to the machine precision level. Note that the orthogonality issue of the Arnoldi vectors typically does not have a big impact in the GMRES method, for which the modified G-S performs well. On the other hand, when computing the low-rank correction in the SLR preconditioner, we cannot forego reorthogonalization in the Arnoldi process.

The results shown in section also apply to the symmetric case when both A and C are indefinite. The extension to complex non-Hermitian matrices is straightforward.

6.8 Numerical experiments

The experiments were conducted on a machine at Minnesota Supercomputing Institute, equipped with two Intel Xeon X5560 processors (8 MB Cache, 2.8 GHz, 4-core) and 24 GB of memory. An implementation of the SLR preconditioner was written in C/C++, and the code was compiled by the Intel C compiler with the -O2 optimization level. BLAS and LAPACK routines from Intel Math Kernel Library were used to enhance the performance on multiple cores. Thread-level parallelism was realized by OpenMP [27].

For symmetric linear systems, the accelerators used were the CG method for the SPD cases, and GMRES(40) for the indefinite cases. For the nonsymmetric cases, GMRES(40) was used. Three types of preconditioning methods were tested in our experiments: 1) ILU-type preconditioners with threshold dropping that include the incomplete Cholesky factorization (ICT), the incomplete LDL factorization (ILDLT) and ILUTP, 2) the restricted additive Schwarz (RAS) method [83] (with one-level overlapping) and 3) the SLR method and the MSLR method. For the RAS method, we used ILU factorizations as the local solvers. Moreover, since the RAS preconditioner is nonsymmetric even for a symmetric matrix, GMRES(40) was used with it.

For all the problems, we used the graph partitioner `PartGraphRecursive` from Metis [76, 161] to partition the domains. The time for the graph partitioning will not be included in the time of building the preconditioners. For each subdomain i, B_i was reordered by the approximate minimum degree (AMD) ordering [2, 110, 111] and ILU was used as the local solver. Prior to computing the ILU preconditioners, the AMD ordering was applied to the original matrix. In the SLR method, matrix C was factored by ILU, while in the MSLR method, only the C at the last level was factored. In the Lanczos/Arnoldi procedure, we set the maximum number of steps as five times the number of requested eigenvalues.

Based on the experimental results, we can state that in general, building an SLR preconditioner, especially when using larger ranks, requires much more time than an ICT/ILDLT preconditioner or an RAS preconditioner with a similar storage. Nevertheless, experimental results indicated that the SLR preconditioner is more robust and can also achieve great time savings in the iterative phase. In the cases of systems with a large number of right-hand sides, expensive but effective preconditioners may be justified because their cost is amortized. In this section, we first report on the results of solving symmetric linear systems from a 2-D/3-D PDE on regular meshes. Then, we report the performance of the SLR preconditioner for solving complex non-Hermitian linear systems from 2-D and 3-D Helmholtz equations. Last, we will show the results for solving a sequence of general sparse symmetric linear systems. Except for the Helmholtz problems, the iterations were stopped whenever the residual norm had been reduced by 8 orders of magnitude or the maximum number of iterations allowed, which is 300, was exceeded. The results are summarized in Tables 6.2 - 6.7 and 6.9, where all times are reported in seconds.

6.8.1 2-D/3-D model problems

We consider 2-D and 3-D PDE problems,

$$\begin{aligned} -\Delta u - cu &= f \ \text{ in } \Omega, \\ u &= \phi(x) \text{ on } \partial\Omega, \end{aligned} \tag{6.42}$$

where $\Omega = (0,1)^2$ and $\Omega = (0,1)^3$ are the domains, and $\partial\Omega$ is the boundary. We take the 5-point or 7-point centered difference approximation on the regular meshes.

To begin with, we examine the required ranks of the SLR method in order to bound the spectral condition number of the matrix $S\tilde{S}^{-1}$ by a constant K. Recall from (6.28) that this requires that the $(k+1)$-st largest eigenvalue, λ_{k+1}, of the matrix $C^{-1}E^TB^{-1}E$ be less than $1-1/K$. The results for 2-D/3-D Laplacians are shown in Table 6.1, where 8 subdomains were used for the 2-D case and it was 32 for the 3-D case. From there we can see that for the 2-D problems, the required rank is about doubled when the step-size is reduced by half, while for the 3-D cases, the rank needs to be increased by a factor of roughly 3.5.

In the next set of experiments, we solve (6.42) with $c = 0$, so that the coefficient matrices are SPD and we use the SLR preconditioner with the CG method. Numerical experiments were carried out to compare the performance of the SLR preconditioner with the ICT and the RAS preconditioners. The results are shown in Table 6.2. The sizes of the grids, the fill-ratios (fill), the numbers of iterations (its), the time for building the preconditioners (p-t) and the time for iterations (i-t) are tabulated. For the SLR

Table 6.1: The required ranks of the SLR method for bounding $\kappa(S\tilde{S}^{-1})$ by K for 2-D/3-D Laplacians.

Grid	rank $K \approx 33$	Grid	rank $K \approx 12$
128^2	3	25^3	1
256^2	8	40^3	4
512^2	20	64^3	12
1024^2	42	100^3	42

preconditioners, the number of subdomains (nd) and the rank (rk) are also listed. The fill-ratios of the three preconditioners were controlled to be roughly equal. For all the cases tested here and in the following sections, the RAS method always used the same numbers of subdomains as did the SLR method. The ICT factorizations were used for the solves with the matrices B and C in the SLR method. As shown in Table 6.2, we tested the problems on three 2-D grids and three 3-D grids of increasing sizes, where for the RAS method and the SLR method, the domain was partitioned into 32, 64 and 128 subdomains respectively, and the ranks 16 or 32 were used in the SLR preconditioners.

Table 6.2: Performance of the ICT, the RAS and the SLR preconditioners for solving SPD linear systems from the 2-D/3-D PDE with CG.

Grid	ICT-CG				RAS-GMRES				SLR-CG					
	fill	p-t	its	i-t	fill	p-t	its	i-t	nd	rk	fill	p-t	its	i-t
256^2	4.5	.074	51	.239	4.5	.088	129	.281	32	16	4.3	.090	67	.145
512^2	4.6	.299	97	1.93	4.8	.356	259	2.34	64	32	4.9	.650	103	1.01
1024^2	5.4	1.44	149	14.2	6.2	1.94	F	-	128	32	5.7	5.23	175	7.95
40^3	4.4	.125	25	.152	4.5	.145	36	.101	32	16	4.0	.182	31	.104
64^3	6.8	.976	32	1.24	6.2	.912	49	.622	64	32	6.3	1.52	38	.633
100^3	7.3	4.05	47	7.52	6.1	3.48	82	4.29	128	32	6.5	5.50	67	4.48

Compared with the ICT and the RAS preconditioners, building an SLR preconditioner required more CPU time (up to 4 times more for the largest 2-D case). For these problems, the SLR-CG method achieved convergence in slightly more iterations than those with the ICT preconditioner, but SLR still achieved performance gains in terms of significantly reduced iteration times. The CPU time for building an SLR preconditioner is typically dominated by the cost of the Lanczos algorithm. Furthermore, this cost is actually governed by the cost of the solves with B_i's and C, which are required at each iteration. Moreover, when the rank k used is large, the cost of reorthogonalization will also become significant. Some simple thread-level parallelism has been exploited using OpenMP for the solves with the B_i's, which can be performed independently. The multi-threaded MKL routines also helped speedup the vector operations in the reorthogonalizations. We point out that there is room for substantial improvements in the performance of these computations. In particular they are very suitable for the SIMD type parallel machines such as computers equipped with GPUs or with the Intel Xeon Phi processors. These features have not yet been implemented in the current code.

Next, we consider solving the symmetric indefinite problems by setting $c > 0$ in (6.42), which corresponds to shifting the discretized negative Laplacian (a positive definite matrix) by subtracting sI with a certain $s > 0$. In this set of experiments, we solve the 2-D problems with $s = 0.01$ and the 3-D problems with $s = 0.05$. The SLR method is compared with ILDLT and RAS with GMRES(40).

Table 6.3: Performance of the ILDLT, the RAS and the SLR preconditioners for solving symmetric indefinite linear systems from the 2-D/3-D PDE with GMRES(40).

Grid	ILDLT-GMRES				RAS-GMRES				SLR-GMRES					
	fill	p-t	its	i-t	fill	p-t	its	i-t	nd	rk	fill	p-t	its	i-t
256^2	6.5	.125	F	-	6.3	.134	F	-	8	32	6.4	.213	33	.125
512^2	8.4	.702	F	-	8.4	.721	F	-	16	64	7.6	2.06	93	1.50
1024^2	12.6	5.14	F	-	19.4	21.6	F	-	8	128	10.8	24.5	50	4.81
40^3	6.7	.249	54	.540	6.7	.254	99	.300	64	32	6.7	.490	23	.123
64^3	9.0	1.39	F	-	11.8	2.16	F	-	128	64	9.1	3.94	45	1.16
100^3	14.7	10.9	F	-	11.7	14.5	F	-	128	180	14.6	62.9	88	13.9

Results are shown in Table 6.3. For most problems, the ILDLT/GMRES and the RAS/GMRES method failed even with high fill-ratios. In contrast, the SLR method appears to be more effective, achieving convergence for all cases, and great savings in the iteration time. In contrast with the SPD case, a few difficulties were encountered. For the 2-D problems, an SLR preconditioner with a large number of subdomains (say, 64 or 128) often failed to converge. As a result the sizes of the subdomains were still quite large and factoring the matrices B_i's was quite expensive in terms of both the CPU time and the memory requirement. Furthermore, for both the 2-D and 3-D problems, approximations of higher ranks were required compared with those used in the SPD cases. This only increased the memory requirement slightly, but it significantly increased the CPU time required by the Lanczos algorithm. An example is the largest 3-D problem in Table 6.3, where a rank of 180 was used.

Finally in this section, we compare the performance between the SLR and the MSLR methods on the SPD cases for the 3-D model problem. We report the results in Table 6.4. From the result for the largest problem, we can see the advantages of the MSLR method. Compared with the SLR method, it required not only less CPU time to construct the preconditioner but slightly fewer iterations and less iteration time to converge. More results of the MSLR preconditioner for solving indefinite model problems and systems with general matrices can be found in [167].

Table 6.4: Performance of the SLR and the MSLR preconditioners for solving SPD linear systems from the 2-D/3-D PDE with CG.

Grid	MSLR-CG						SLR-CG				
	nlev	rk	fill	p-t	its	i-t	rk	fill	p-t	its	i-t
32^3	7	16	4.13	0.10	17	0.03	16	4.18	0.15	16	0.03
64^3	10	16	4.07	0.85	35	0.47	16	4.14	1.05	37	0.61
128^3	13	16	4.16	10.18	66	8.21	16	4.17	10.80	78	10.9

6.8.2 2-D/3-D Helmholtz problems

This section discusses the performance of the SLR preconditioner for solving problems from a 2-D/3-D Helmholtz equation of the form

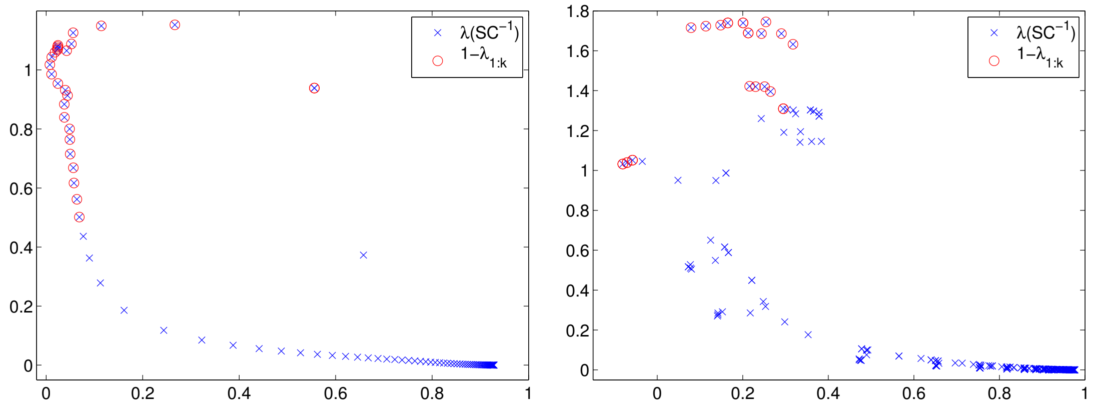
$$\left(-\Delta - \frac{\omega^2}{v(x)^2}\right) u(x,\omega) = s(x,\omega) \tag{6.43}$$

where Δ is the Laplacian operator, ω is the angular frequency, $v(x)$ is the velocity field, and $u(x,\omega)$ is called the time-harmonic wave field solution to the external forcing term

$s(x,\omega)$. The domain of interest is the unit box, $(0,1)^d$ with $d = 2, 3$, where we take the 5-point or the 7-point centered difference approximation with regular meshes. The PML boundary condition is used at all sides. The resulting coefficient matrices are *complex non-Hermitian*. In (6.43), we assume that the mean of $v(x)$ is equal to 1. Then, $\omega/(2\pi)$ is the wave number and $\lambda = 2\pi/\omega$ is the wavelength. When the sampling rate of q points per wavelength is used, the number of mesh points for each dimension is $N = q\omega/(2\pi)$ and the coefficient matrix is of the order $n = N^d$, which is a negative Laplacian shifted by $-sI$ with $s \approx \omega^2/N^2 = (2\pi/q)^2$.

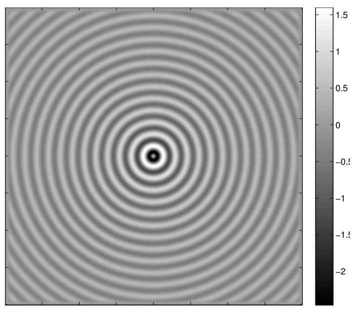
To begin with, we examine the eigenvalues of the preconditioned Schur complement with the SLR preconditioner. Figure 6.8 shows the eigenvalues of SC^{-1} and the eigenvalues corresponding to the low-rank correction, $\lambda_{1:k}$. Observe that in both cases, many eigenvalues of SC^{-1} are clustered around 1 and that most are far away from the origin. Thus, we chose the eigenvalues that are far from 1 and the corresponding Schur vectors for the low-rank correction. These eigenvalues will be deflated to 1, while the remaining ones will be scaled by a scalar $(1-\theta)^{-1}$. Recall that the eigenvalues of SC^{-1} are $1-\lambda_i$, where λ_i are the eigenvalues of the matrix H in (6.33). Therefore, the eigenvalues of SC^{-1} far from 1 correspond to the eigenvalues of H that are large in magnitude. These eigenvalues and the Schur vectors can be efficiently computed by the Arnoldi process.

Figure 6.8: The eigenvalues of SC^{-1} and the k eigenvalues associated with the low-rank correction for 2-D and 3-D Helmholtz problems on 100^2 (left) and 20^3 (right) meshes.



We first tested the performance of the SLR preconditioner compared with the ILUTP preconditioner for solving the 2-D Helmholtz problem discretized with $q = 16$ points per wavelength. The corresponding shift is about $-0.15I$. GMRES(40) was used with the relative residual tolerance set to be 10^{-5}. For the SLR preconditioner, $S_{k,\theta}^{-1}$, in (6.40), we set $\theta = \lambda_{k+1}$. The results are shown in Table 6.5. The picture on the left shows the solution with $\omega/(2\pi) = 25$. We tested the problem on 6 meshes of increasing sizes, where the wave number $\omega/(2\pi)$ is proportional to N. For all the problems, convergence was achieved with the SLR method using higher ranks for larger problems, whereas the ILUTP preconditioner failed for the 3 large problems.

Table 6.5: Results of the SLR and ILUTP preconditioners for solving 2-D Helmholtz problems with GMRES(40). 16 points per wavelength. Left: the solution with $\frac{\omega}{2\pi} = 25$.

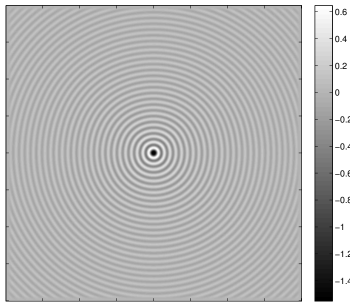


$\omega/(2\pi)$	q	$n = N^2$	SLR				ILUTP	
			nd	rk	fill	its	fill	its
6.25	16	100^2	4	32	3.3	27	3.3	55
12.5	16	200^2	8	64	6.8	28	6.7	36
25	16	400^2	8	64	8.7	61	8.6	270
31.25	16	500^2	8	128	9.6	44	9.7	F
37.5	16	600^2	8	200	9.6	39	10.1	F
43.75	16	700^2	8	250	10.1	46	11.7	F

Next we consider 2-D Helmholtz problems with $q = 8$. This set of problems is harder than the previous one, as indicated by the higher wave numbers and the larger shift, which is about $-0.62I$, for the coefficient matrix. The results are shown in Table 6.6. The picture on the left shows the solution with $\omega/(2\pi) = 50$. Compared with the cases with $q = 16$, higher ranks were used and more iterations were required for convergence. For these problems, the SLR method still outperformed the ILUTP preconditioner. It succeeded for all the cases and required fewer iterations for the smallest case where the ILUTP method also worked.

Next, we solve 3-D Helmholtz problems on 6 cubes with $q = 8$. The results are shown in Table 6.7. The picture shows the solution with $\frac{\omega}{2\pi} = 10$. Compared with the 2-D problems, larger numbers of subdomains were used in order to keep the cost of factoring B_i inexpensive, and the ranks used were smaller. The SLR method showed a performance that is superior to that of the ILUTP preconditioner. It led to convergence

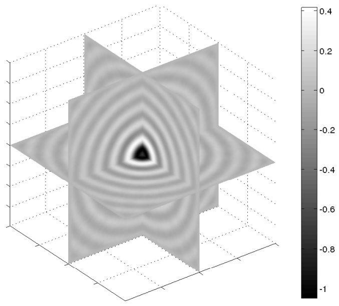
Table 6.6: Results of the SLR and ILUTP preconditioners for solving 2-D Helmholtz problems with GMRES(40). 8 points per wavelength. Left: the solution with $\frac{\omega}{2\pi} = 50$.



$\omega/(2\pi)$	q	$n = N^2$	SLR				ILUTP	
			nd	rk	fill	its	fill	its
12.5	8	100^2	4	64	4.9	10	4.8	58
25	8	200^2	8	128	6.6	21	6.9	F
50	8	400^2	8	128	8.3	100	8.2	F
62.5	8	500^2	8	256	9.2	84	9.3	F
75	8	600^2	8	400	10.3	153	12.2	F
87.5	8	700^2	8	500	10.6	236	10.7	F

for all the cases and required fewer iterations than ILUTP when both worked.

Table 6.7: Results of the SLR and ILUTP preconditioners for solving 3-D Helmholtz problems with GMRES(40). 8 points per wavelength. Left: the solution with $\frac{\omega}{2\pi} = 10$.



$\omega/(2\pi)$	q	$n = N^3$	SLR				ILUTP	
			nd	rk	fill	its	fill	its
2.5	8	20^3	4	32	3.4	28	3.3	33
3.75	8	30^3	8	32	4.1	49	3.9	60
5	8	40^3	16	32	4.5	88	4.5	125
6.25	8	50^3	16	32	8.5	85	8.7	145
7.5	8	60^3	16	128	11.6	88	12.2	F
10	8	80^3	16	128	15.6	176	16.6	F

6.8.3 General symmetric matrices

We selected 15 symmetric matrices from the University of Florida sparse matrix collection [26]. Among these matrices, 10 matrices are SPD and the other 5 matrices are symmetric indefinite. Table 6.8 lists the name, the order (N), the number of nonzeros (NNZ), the positive definiteness, and a short description for each matrix. If a right-hand side is not provided, an artificial one was created by $b = Ae$, where e is a random vector of unit 2-norm.

Table 6.9 shows the performance of the three preconditioning methods. The CG method and the GMRES method achieved convergence for all the cases with the SLR

Table 6.8: Names, orders (N), numbers of nonzeros (NNZ), positive definiteness and short descriptions of the test matrices.

Matrix	N	NNZ	SPD	Description
Williams/cant	62,451	4,007,383	yes	FEM cantilever
UTEP/dubcova2	65,025	1,030,225	yes	2-D/3-D PDE problem
UTEP/dubcova3	146,689	3,636,643	yes	2-D/3-D PDE problem
Rothberg/cfd1	70,656	1,825,580	yes	CFD problem
Rothberg/cfd2	123,440	3,085,406	yes	CFD problem
Schmid/thermal1	82,654	574,458	yes	thermal problem
Schmid/thermal2	1,228,045	8,580,313	yes	thermal problem
Wissgott/parabolic_fem	525,825	3,674,625	yes	CFD problem
CEMW/tmt_sym	726,713	5,080,961	yes	electromagnetics problem
McRae/ecology2	999,999	4,995,991	yes	landscape ecology problem
Lin/Lin	256,000	1,766,400	no	structural problem
Cote/vibrobox	12,328	301,700	no	vibroacoustic problem
Cunningham/qa8fk	66,127	1,660,579	no	3-D acoustics problem
Koutsovasilis/F2	71,505	5,294,285	no	structural problem
GHS_indef/helm2d03	392,257	2,741,935	no	2-D Helmholtz problem

preconditioner, whereas for many cases, they failed to converge with the ICT/ILDLT and the RAS preconditioners. Similar to the experiment results for the PDE problems, the SLR preconditioner often required more CPU time to build than the other two counterparts when the memory requirements of these methods are similar. In the iteration phase, the SLR preconditioner required fewer iterations for most of the problems, and achieved significant CPU time savings for almost all the cases where the other two methods also worked (the exception is `qa8fk`, for which the RAS method gave the best iteration time).

6.9 Summary and discussion

This chapter has presented a preconditioning method, named SLR, based on Schur complement techniques with low-rank corrections for solving general sparse linear systems. Like the MLR and DDLR preconditioners presented in the previous chapters, low-rank approximations are used to build the preconditioner. The SLR method computes an approximate inverse of the Schur complement by exploiting some decay property of eigenvalues. Then, the preconditioner for the original matrix is followed by the Schur

Table 6.9: Performance of the ICT/ILDLT, the RAS and the SLR preconditioners for solving general symmetric linear systems.

Matrix	ICT/ILDLT				RAS				SLR					
	fill	p-t	its	i-t	fill	p-t	its	i-t	nd	rk	fill	p-t	its	i-t
cant	4.7	3.87	150	9.34	5.9	6.25	F	-	32	90	4.9	5.58	82	1.92
dubcova2	2.7	.300	47	.492	2.8	.489	60	.223	16	32	2.8	.280	19	.080
dubcova3	2.2	1.01	46	1.44	2.1	1.46	59	.654	16	32	1.8	.677	19	.212
cfd1	6.9	2.89	295	11.9	8.3	3.04	F	-	32	32	6.9	2.13	64	1.07
cfd2	9.9	13.5	F	-	8.9	7.88	F	-	32	80	8.8	7.62	178	5.75
thermal1	5.1	.227	68	.711	5.0	.348	F	-	16	32	5.0	.277	59	.231
thermal2	6.9	5.10	178	39.3	7.1	8.46	F	-	64	90	6.6	14.8	184	15.0
para_fem	6.1	2.04	58	4.68	6.3	3.17	236	6.11	32	80	6.9	6.05	86	3.03
tmt_sym	6.0	1.85	122	11.6	6.2	3.67	F	-	64	80	5.9	6.61	127	5.23
ecology2	8.4	2.64	142	18.5	9.5	4.78	F	-	32	96	8.0	12.3	90	5.58
Lin	11	1.93	F	-	19	4.61	F	-	64	64	9.9	3.78	73	1.75
vibrobox	6.0	.738	F	-	7.0	.513	F	-	4	64	3.8	.437	226	.619
qa8fk	4.2	.789	22	.507	4.6	1.14	35	.273	16	64	4.5	1.94	28	.309
F2	5.1	9.66	F	-	5.4	9.43	F	-	8	80	3.9	6.25	72	2.14
helm2d03	14	14.4	F	-	11	7.20	F	-	16	128	11	11.9	63	2.63

complement approaches in any standard domain decomposition framework. Experimental results indicated that in terms of iteration time, the proposed preconditioner can be an efficient alternative to the ones based on the ILU-type methods for SPD systems. Moreover, the SLR preconditioner appears to be more robust than the ILU-based methods for indefinite problems. The results for the shifted Laplacian matrices and the Helmholtz problems support this finding. Building an SLR preconditioner can be time consuming. The mitigating factors pointed out in Section 4.12 for the MLR method also apply here.

Chapter 7

Conclusion

This thesis has presented several preconditioning approaches combined with Krylov subspace methods that can be accelerated by the current many-core processors for solving sparse linear systems. Significant performance enhancement has been achieved with appropriate preconditioners. On the other hand, incomplete LU (ILU) type preconditioners that are among the most reliable methods in general settings yield exceedingly poor performance on these new platforms due to the serial nature of the computations.

A class of new preconditioning techniques based on low-rank approximations has been presented along with their implementations and the performance comparisons with standard preconditioners. These approaches are proposed primarily as means to bypass the difficulties encountered by ILU preconditioners. The first approach was presented in Chapter 4. It uses a divide-and-conquer paradigm: an approximate inverse of the matrix corresponding to a whole domain can be computed by the sum of the inverses of the two matrices associated with the bipartitioned subdomains and a low-rank correction term. The second method presented in Chapter 5 uses a more general setting of domain decomposition (DD) so that it can be easily adopted in the general framework of distributed sparse linear systems. Instead of solving the reduce system that involves the global Schur complement as in traditional parallel ILU methods, this method computes an approximate inverse based on a low-rank property. The third method that we presented in Chapter 6 utilizes Schur complement approaches with standard DD, which tries to approximate the inverse of the Schur complement directly by exploiting some decay property of eigenvalues. Experiment results indicated that these preconditioning methods appear to be more robust than the traditional ILU-based methods for indefinite problems. Although these methods have not been implemented and tested on the targeted computing environments, i.e., the ones equipped with many-core accelerators, the efficiency in the iteration phase has been seen from the experiment results on multi-core CPUs. The advantages and disadvantages of these methods were discussed in Section 4.12.

The advent of advanced supercomputers, especially the ones with the recent massively parallel processors, lead to a growing demand for algorithms with high degrees of parallelism. A metric of computational cost by floating point operation (FLOP) counts becomes less and less informative on the real performance of sparse matrix computations on these machines. Very often, one may consider trading the volume of computations with the speed. Moreover, huge performance gaps have been reported [96, 176, 177] between the throughput of dense matrix computations, which can yield performance quite close to the peak performance of modern processors, and that of sparse matrix computations. This indeed demands that efficient algorithms should have less "irregular" computations—computations in the sparse mode. The presented preconditioning methods based on low-rank approximations are superior to the traditional ILU-type methods in both aspects, so that they can show better performance.

A recent trend in computing preconditioners is to exploit some sort of low-rank properties [129–137]. Instead of merely seeking the "nonzero sparsity" as in the traditional ILU-type or approximate inverse type methods by dropping elements based on their magnitudes, these methods also exploit a "rank sparsity", i.e., representing certain matrices by approximations of very low ranks. These preconditioning methods can exhibit higher robustness and accuracy on indefinite problems. The exciting results of the sweeping preconditioner [131] by Engquist and Ying on Helmholtz problems will prompt our further exploration of low-rank approximation based preconditioners. Compared with the sweeping preconditioner, which is still quite restrictive to a certain kind of problems and the types of discretizations and boundary conditions, our presented approaches are more general, derived and computed in a pure algebraic way.

By concluding this thesis, this line of work on preconditioning methods with the considerations of both parallel efficiency and robustness for indefinite problems is by no means finished. Parallel preconditioning methods presented in Chapter 3 have not been implemented and tested on a more recently emerged many-core accelerator, the Intel Xeon Phi processor. The SIMD parallelism paradigm of this platform is similar to that of Graphic Processing Units (GPUs) but the programming model is quite different. Performance evaluations on these processors remain in the future work.

For computing and applying ILU preconditioners on GPUs, a recent novel approach was presented in [178]. The results shown therein have led to the reconsideration by the author of the current view of ILU-type methods running on GPUs.

For the presented low-rank approximation based preconditioners, several problems are still open, most of which have been disseminated in the "Summary and Discussion" sections of Chapters 4 to 6. A few directions of the future work are mentioned as follows. The first ongoing work is the implementations of these methods with the acceleration by the many-core processors. Next, since the constructions of these preconditioners are very time-consuming, more efficient algorithms than the currently used Lanczos/Arnoldi procedures need to be exploited. Finally, in Section 4.10, a simple way of using tensor representations and low-rank approximations by tensors in order to reduce memory requirement has been presented. So far, the use of tensors is restricted to 2-D and 3-D model problems. For general matrices, this use is still not clear. As some low-rank properties have been seen to exist in higher dimensions, tensors may be of more and greater use.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[2] T. A. Davis. *Direct Methods for Sparse Linear Systems.* SIAM, 2006, http://epubs.siam.org/doi/pdf/10.1137/1.9780898718881.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Inc., New York, NY, USA, 1986.

[4] H. A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems.* Cambridge University Press, 2003. Cambridge Books Online.

[5] J. W. Ruge and K. Stüben. *Algebraic Multigrid*, chapter 4, pages 73–130. SIAM, 1987, http://epubs.siam.org/doi/pdf/10.1137/1.9781611971057.ch4.

[6] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

[7] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[8] E. Chow and P. S. Vassilevski. Multilevel block factorizations in generalized hierarchical bases. *Numerical Linear Algebra with Applications*, 10(1-2):105–127, 2003.

[9] Y. Notay. Aggregation-based algebraic multilevel preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 27(4):998–1018, 2006, http://dx.doi.org/10.1137/04061129X.

[10] D. Osei-Kuffuor, R. Li, and Y. Saad. Matrix reordering using multilevel graph coarsening for ILU preconditioning. *SIAM Journal on Scientific Computing*, 37(1):A391–A419, 2015, http://dx.doi.org/10.1137/130936610.

[11] Y. Notay and P. S. Vassilevski. Recursive Krylov-based multigrid cycles. *Numerical Linear Algebra with Applications*, 15(5):473–487, 2008.

[12] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition.* SIAM, Philadelpha, PA, 2003.

[13] M. Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comput. Phys.*, 182(2):418–477, November 2002.

[14] L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra.* SIAM, Philadelpha, PA, 1997.

[15] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.

[16] M. Benzi, C. D. Meyer, and Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17:1135–1149, 1996.

[17] M. Benzi and M. Tůma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10(5-6):385–400, 2003.

[18] E. Chow and Y. Saad. Approximate inverse techniques for block-partitioned matrices. *SIAM Journal on Scientific Computing*, 18(6):1657–1675, 1997, http://epubs.siam.org/doi/pdf/10.1137/S1064827595281575.

[19] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63:443–466, 2013.

[20] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Q. Appl. Math*, 9(17):17–29, 1951.

[21] R. B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995, http://dx.doi.org/10.1137/S0895479893253975.

[22] N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen. How fast are nonsymmetric matrix iterations? *SIAM Journal on Matrix Analysis and Applications*, 13(3):778–795, 1992, http://dx.doi.org/10.1137/0613049.

[23] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976, http://dx.doi.org/10.1137/0205021.

[24] D. Hysom and A. Pothen. Level-based incomplete LU factorization: Graph model and algorithms. Technical Report UCRL-JC-150789, Lawrence Livermore National Laboratory, 2002.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[26] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[27] OpenMP Architecture Review Board. OpenMP application program interface version 3.1, July 2011.

[28] Y. Saad. ILUT: a dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[29] M. Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338(13):201–218, 2001.

[30] N. Li, Y. Saad, and E. Chow. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing*, 25(2):716–728, 2003, http://dx.doi.org/10.1137/S1064827502405094.

[31] C. Lin and J. Moré. Incomplete Cholesky factorizations with limited memory. *SIAM Journal on Scientific Computing*, 21(1):24–45, 1999, http://dx.doi.org/10.1137/S1064827597327334.

[32] M. Jones and P. Plassmann. An improved incomplete Cholesky factorization. *ACM Trans. Math. Softw.*, 21(1):5–17, March 1995.

[33] N. Li and Y. Saad. Crout versions of ILU factorization with pivoting for sparse symmetric matrices. *Electronic Transactions on Numerical Analysis*, 20:75–85, 2005.

[34] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):pp. 148–162, 1977.

[35] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of computation*, 34(150):473–497, April 1980.

[36] Y. Robert. Regular incomplete factorizations of real positive definite matrices. *Linear Algebra and its Applications*, 48(0):105–117, 1982.

[37] R. S. Varga, E. B. Saff, and V. Mehrmann. Incomplete factorizations of matrices and connections with H-matrices. *SIAM Journal on Numerical Analysis*, 17(6):787–793, 1980, http://dx.doi.org/10.1137/0717066.

[38] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics*, 26(1):43–65, 1978.

[39] Y. Saad. Preconditioned Krylov subspace methods for CFD applications. In *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, pages 179–195. Wiley, 1995.

[40] H. A. van der Vorst. Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from PDE-problems. *Journal of Computational Physics*, 44(1):1–19, 1981.

[41] O. Axelsson and L. Kolotilina. Diagonally compensated reduction and related preconditioning methods. *Numerical Linear Algebra with Applications*, 1(2):155–177, 1994.

[42] M. A. Ajiz and A. Jennings. A robust incomplete Choleski-conjugate gradient algorithm. *International Journal for Numerical Methods in Engineering*, 20(5):949–966, 1984.

[43] H. C. Elman. A stability analysis of incomplete lu factorizations. *Mathematics of Computation*, 47(175):pp. 191–217, 1986.

[44] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414, 1997.

[45] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT Numerical Mathematics*, 29(4):635–657, 1989.

[46] S. T. Barnard, A. Pothen, and H. Simon. A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334, 1995.

[47] R. Bridson and W. Tang. A structural diagnosis of some IC orderings. *SIAM Journal on Scientific Computing*, 22(5):1527–1532, 2001, http://dx.doi.org/10.1137/S1064827599353841.

[48] M. Benzi, D. Szyld, and A. van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing*, 20(5):1652–1670, 1999, http://dx.doi.org/10.1137/S1064827597326845.

[49] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001, http://dx.doi.org/10.1137/S0895479899358443.

[50] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, July 1999.

[51] Y. Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *Journal of Computational and Applied Mathematics*, 24(1-2):89–105, 1988.

[52] O. Axelsson and M. Larin. An algebraic multilevel iteration method for finite element matrices. *Journal of Computational and Applied Mathematics*, 89(1):135–153, 1998.

[53] O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods, II. *SIAM Journal on Numerical Analysis*, 27(6):1569–1590, 1990, http://dx.doi.org/10.1137/0727092.

[54] E.F.F. Botta and F.W. Wubs. *MRILU: It's the Preconditioning that Counts.* Tech. Report, W-9703, Department of Mathematics, University of Groningen, The Netherlands, 1997.

[55] A. van der Ploeg, E.F.F. Botta, and F.W. Wubs. Nested grids ILU-decomposition (NGILU). *Journal of Computational and Applied Mathematics*, 66(12):515–526, 1996. Proceedings of the Sixth International Congress on Computational and Applied Mathematics.

[56] J. Zhang. A grid-based multilevel incomplete LU factorization preconditioning technique for general sparse matrices. *Appl. Math. Comput.*, 124(1):95–115, November 2001.

[57] J. Zhang. A multilevel dual reordering strategy for robust incomplete LU factorization of indefinite matrices. *SIAM Journal on Matrix Analysis and Applications*, 22(3):925–947, 2001, http://dx.doi.org/10.1137/S0895479899354251.

[58] J. Zhang. A class of multilevel recursive incomplete LU preconditioning techniques. *Korean Journal of Computational and Applied Mathematics*, 8(2):213–234, 2001.

[59] Y. Saad and J. Zhang. BILUM: Block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(6):2103–2121, 1999, http://dx.doi.org/10.1137/S106482759732753X.

[60] Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17(4):830–847, 1996, http://dx.doi.org/10.1137/0917054.

[61] Y. Saad and J. Zhang. BILUTM: A domain-based multilevel block ilut preconditioner for general sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 21(1):279–299, 1999, http://dx.doi.org/10.1137/S0895479898341268.

[62] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9, 2002.

[63] Y. Notay. A multilevel block incomplete factorization preconditioning. *Applied Numerical Mathematics*, 31(2):209–225, 1999.

[64] Y. Notay. Robust parameter-free algebraic multilevel preconditioning. *Numerical Linear Algebra with Applications*, 9(6-7):409–428, 2002.

[65] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973, http://dx.doi.org/10.1137/0710032.

[66] S. MacLachlan and Y. Saad. A greedy strategy for coarse-grid selection. *SIAM Journal on Scientific Computing*, 29(5):1825–1853, 2007, http://dx.doi.org/10.1137/060654062.

[67] Y. Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM Journal on Scientific Computing*, 27(3):1032–1057, 2005, http://dx.doi.org/10.1137/030602733.

[68] S. MacLachlan and Y. Saad. Greedy coarsening strategies for nonsymmetric problems. *SIAM Journal on Scientific Computing*, 29(5):2115–2143, 2007, http://dx.doi.org/10.1137/060660928.

[69] Y. Notay and Z. Ould Amar. Incomplete factorization preconditioning may lead to multigrid like speed of convergence. *Advanced Mathematics: Computation and Applications, AS Alekseev and NS Bakhvalov, eds., NCC Publisher, Novosibirsk, Russia*, pages 435–446, 1996.

[70] A. Reusken. A multigrid method based on incomplete Gaussian elimination. *Numerical Linear Algebra with Applications*, 3(5):369–390, 1996.

[71] C. Wagner, W. Kinzelbach, and G. Wittum. Schur-complement multigrid. *Numerische Mathematik*, 75(4):523–545, 1997.

[72] R. E. Bank and R. K. Smith. The incomplete factorization multigraph algorithm. *SIAM Journal on Scientific Computing*, 20(4):1349–1364, 1999, http://dx.doi.org/10.1137/S1064827597319520.

[73] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid.* Academic Press, 1st edition, December 2000.

[74] Ü. V. Çatalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[75] B. Hendrickson and R. Leland. *The Chaco User's Guide Version 2.* Sandia National Laboratories, Albuquerque NM, 1994.

[76] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput*, 20(1):359–392, 1998, http://dx.doi.org/10.1137/S1064827595287997.

[77] T. G. Kolda. Partitioning sparse rectangular matrices for parallel processing. *Lecture Notes in Computer Science*, 1457:68–79, 1998.

[78] F. Pellegrini. SCOTCH *and* LIBSCOTCH *5.1 User's Guide.* INRIA Bordeaux Sud-Ouest, IPB & LaBRI, UMR CNRS 5800, 2010.

[79] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11:430–452, 1990.

[80] M. Dryja and O. Widlund. *An additive variant of the Schwarz alternating method for the case of many subregions.* New York: Courant Institute of Mathematical Sciences, New York University, 1987.

[81] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, NY, USA, 1996.

[82] X. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 3(3):221–237, 1996.

[83] X. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, 1999, http://dx.doi.org/10.1137/S106482759732678X.

[84] S. Ma and Y. Saad. Distributed ILU(0) and SOR preconditioners for unstructured sparse linear systems. Technical Report 94-027, Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN, 1994.

[85] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2001, http://dx.doi.org/10.1137/S1064827500376193.

[86] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. In *Supercomputing, ACM/IEEE 1997 Conference*, page 28, Nov 1997.

[87] M. Magolu monga Made and H. A. van der Vorst. Parallel incomplete factorizations with pseudo-overlapped subdomains. *Parallel Computing*, 27(8):989–1008, 2001.

[88] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10(5-6):485–509, 2003.

[89] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(4):1337–1356, 1999, http://dx.doi.org/10.1137/S1064827597328996.

[90] Y. Saad and M. Sosonkina. Enhanced parallel multicolor preconditioning techniques for linear systems. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM: Philadelphia, PA, 1999.

[91] Z. Li and Y. Saad. SchurRAS: A restricted version of the overlapping Schur complement preconditioner. *SIAM Journal on Scientific Computing*, 27(5):1787–1801, 2006, http://dx.doi.org/10.1137/040608350.

[92] NVIDIA. *NVIDIA CUDA C Programming Guide 7.0*, March 2015.

[93] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22:917–924, 2003.

[94] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, July 2003.

[95] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical report, IBM Research, 2008.

[96] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[97] F. Vázquez, E. M. Garzon, J. A. Martinez, and J. J. Fernandez. The sparse matrix vector product on GPUs. Technical report, Dept of Computer Architecture and Electronics, University of Almeria, 2009.

[98] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM SIGPLAN Notices*, 45:115–126, January 2010.

[99] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 107–118, New York, NY, USA, 2014. ACM.

[100] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In Koen Bertels, Nikitas Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures,*

Modeling, and Simulation, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer Berlin Heidelberg, 2009.

[101] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In Yale Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2010.

[102] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. *Microprocess. Microsyst.*, 36(2):65–77, March 2012.

[103] B. Su and K. Keutzer. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012. ACM.

[104] J. L. Greathouse and M. Daga. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, Piscataway, NJ, USA, 2014. IEEE Press.

[105] S. Williams, N. Bell, J. W. Choi, M. Garland, L. Oliker, and R. Vuduc. *Scientific Computing with Multicore and Accelerators*, chapter 5, pages 83–109. CRC Press, 2010.

[106] NVIDIA. *CUSPARSE Library User Guide 7.0*, March 2014.

[107] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2013. Version 0.4.0.

[108] Joseph W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990, http://dx.doi.org/10.1137/0611010.

[109] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, 31(1):1–19, 1989.

[110] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.

[111] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: An approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):381–388, 2004.

[112] NVIDIA. *CUBLAS Library User Guide 7.0*, March 2014.

[113] R. Karp. Reducibility among combinatorial problems. In R. Miller, J. Thatcher, and J. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[114] W. Gautschi. On generating orthogonal polynomials. *SIAM Journal on Scientific and Statistical Computing*, 3(3):289–317, 1982, http://dx.doi.org/10.1137/0903018.

[115] P. J. Davis. *Interpolation and Approximation.* Blaisdell, Waltham, MA, 1963.

[116] Y. Saad. Iterative solution of indefinite symmetric linear systems by methods using orthogonal polynomials over two disjoint intervals. *SIAM Journal on Numerical Analysis*, 20(4):784–811, 1983, http://dx.doi.org/10.1137/0720052.

[117] J. Erhel, F. Guyomarc'H, and Y. Saad. Least-squares polynomial filters for ill-conditioned linear systems. Technical Report umsi-2001-32, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.

[118] C. 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*, 45:255–282, 1950.

[119] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky. Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration. *Phys. Rev. E*, 74:066704, Dec 2006.

[120] B. N. Parlett. *The Symmetric Eigenvalue Problem.* SIAM, Philadephia, PA, 1998.

[121] M. J. Grote and T. Huckle. Parallel preconditionings with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18:838–853, 1997.

[122] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998, http://epubs.siam.org/doi/pdf/10.1137/S1064827594270415.

[123] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati. FSAIPACK: a software package for high performance Factored Sparse Approximate Inverse preconditioning. *ACM Transactions on Mathematical Software*, 41(2), 2014.

[124] M. Lukash, K. Rupp, and S. Selberherr. Sparse approximate inverse preconditioners for iterative solvers on GPUs. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 13:1–13:8, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[125] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 583–592, Feb 2010.

[126] R. Farina, S. Cuomo, P. De Michele, and M. Chinnici. Inverse preconditioning techniques on a GPUs architecture in global ocean models. In *Proceedings of the 16th WSEAS International Conference on Applied Mathematics*, pages 15–20, 2011.

[127] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012, http://dx.doi.org/10.1137/110838844.

[128] L. Wang, X. Hu, J. Cohen, and J. Xu. A parallel auxiliary grid algebraic multigrid method for graphic processing units. *SIAM Journal on Scientific Computing*, 35(3):C263–C283, 2013, http://dx.doi.org/10.1137/120894452.

[129] W. Hackbusch. A sparse matrix arithmetic based on h-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[130] W. Hackbusch and B. N. Khoromskij. A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems. *Computing*, 64:21–47, 2000.

[131] B. Engquist and L. Ying. Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation. *Communications on Pure and Applied Mathematics*, 64(5):697–735, 2011.

[132] S. Le Borne. $\mathcal{H}$-matrices for convection-diffusion problems with constant convection. *Computing*, 70(3):261–274, July 2003.

[133] S. Le Borne and L. Grasedyck. $\mathcal{H}$-matrix preconditioners in convection-dominated problems. *SIAM Journal on Matrix Analysis and Applications*, 27(4):1172–1183, 2006, http://epubs.siam.org/doi/pdf/10.1137/040615845.

[134] S. Ambikasaran and E. Darve. An $\mathcal{O}(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013.

[135] S. Wang, M. V. de Hoop, and J. Xia. On 3D modeling of seismic wave propagation via a structured parallel multifrontal direct helmholtz solver. *Geophysical Prospecting*, 59(5):857–873, 2011.

[136] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[137] J. Xia and M. Gu. Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices. *SIAM J. MATRIX ANAL. APPL.*, 31(5):2899–2920, 2010.

[138] R. Li and Y. Saad. Divide and conquer low-rank preconditioners for symmetric matrices. *SIAM Journal on Scientific Computing*, 35(4):A2069–A2095, 2013, http://dx.doi.org/10.1137/120872735.

[139] J. Tang and Y. Saad. Domain-decomposition-type methods for computing the diagonal of a matrix inverse. *SIAM Journal on Scientific Computing*, 33(5):2823–2847, 2011, http://epubs.siam.org/doi/pdf/10.1137/100799939.

[140] G. H. Golub and C. F. Van Loan. *Matrix Computations, 4th edition.* Johns Hopkins University Press, Baltimore, MD, 4th edition, 2013.

[141] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide.* SIAM, Philadelphia, PA, USA, 2000.

[142] J. F. Grcar. *Analyses of the Lanczos algorithm and of the approximation problem in richardson's method.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1981. AAI8203472.

[143] B. N. Parlett and D. S. Scott. The Lanczos algorithm with selective orthogonalization. *Mathematics of Computation*, 33(145):pp. 217–238, 1979.

[144] H. D. Simon. The Lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, 42(165):pp. 115–142, 1984.

[145] H. Fang and Y. Saad. A filtered Lanczos procedure for extreme and interior eigenvalue problems. *SIAM Journal on Scientific Computing*, 34(4):A2220–A2246, 2012, http://epubs.siam.org/doi/pdf/10.1137/110836535.

[146] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000, http://dx.doi.org/10.1137/S0895479896305696.

[147] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009, http://dx.doi.org/10.1137/07070111X.

[148] B. Savas and L. Eldén. Handwritten digit classification using higher order singular value decomposition. *Pattern Recognition*, 40(3):993–1003, 2007.

[149] J. Chen and Y. Saad. On the tensor SVD and the optimal low rank orthogonal approximation of tensors. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1709–1734, 2009, http://dx.doi.org/10.1137/070711621.

[150] T. Kolda. Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081, Sandia National Laboratories, April 2006.

[151] M. Dryja and O. Widlund. Additive Schwarz methods for elliptic finite element problems in three dimensions. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 3–18. SIAM, 1991.

[152] R. Li and Y. Saad. Low-rank correction methods for algebraic domain decomposition preconditioners. `arXiv:1505.04341 [cs.NA]`, 2015, http://arxiv.org/abs/1505.04341.

[153] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[154] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* SIAM, 2011, http://epubs.siam.org/doi/pdf/10.1137/1.9781611970739.

[155] A. S. Householder. *Theory of Matrices in Numerical Analysis.* Blaisdell Pub. Co., Johnson, CO, 1964.

[156] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[157] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2014.

[158] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[159] Y. Saad and M. Sosonkina. pARMS: A package for solving general sparse linear systems on parallel computers. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Waniewski, editors, *Parallel Processing and Applied*

Mathematics, volume 2328 of *Lecture Notes in Computer Science*, pages 446–457. Springer Berlin Heidelberg, 2002.

[160] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993, http://dx.doi.org/10.1137/0914028.

[161] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[162] N. Halko, P. Martinsson, and J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011, http://epubs.siam.org/doi/pdf/10.1137/090771806.

[163] R. Li, Y. Xi, and Y. Saad. Schur complement based domain decomposition preconditioners with low-rank corrections. `arXiv:1505.04340 [cs.NA]`, 2015, http://arxiv.org/abs/1505.04340.

[164] A.W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and its applications*. Academic Pr, New York, 1979.

[165] L. Grigori, F. Nataf, and S. Yousef. Robust algebraic Schur complement preconditioners based on low rank corrections. Rapport de recherche RR-8557, INRIA, 2014.

[166] P. Hénon and Y. Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM Journal on Scientific Computing*, 28(6):2266–2293, 2006, http://dx.doi.org/10.1137/040608258.

[167] Y. Xi, R. Li, and Y. Saad. An algebraic multilevel preconditioner with low-rank corrections for general sparse symmetric matrices. Technical Report ys-2014-5, Dept. Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 2014.

[168] B. Smith. Domain decomposition algorithms for the partial differential equations of linear elasticity. Technical report, New York, 1990.

[169] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973, http://dx.doi.org/10.1137/0710032.

[170] Z. Bai and J. W. Demmel. On swapping diagonal blocks in real Schur form. *Linear Algebra and its Applications*, 186(0):75–95, 1993.

[171] G. W. Stewart. Algorithm 506: Hqr3 and exchng: Fortran subroutines for calculating and ordering the eigenvalues of a real upper Hessenberg matrix. *ACM Trans. Math. Softw.*, 2(3):275–280, September 1976.

[172] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, third edition, 1999.

[173] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik*, 101(1):87–100, 2005.

[174] L. Giraud, J. Langou, and M. Rozloznik. The loss of orthogonality in the Gram-Schmidt orthogonalization process. *Computers & Mathematics with Applications*, 50(7):1069–1075, 2005. Numerical Methods and Computational Mechanics.

[175] D. Watkins. *The Matrix Eigenvalue Problem.* SIAM, 2007, http://epubs.siam.org/doi/pdf/10.1137/1.9780898717808.

[176] J. Dongarra and M. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013-4744, Sandia National Laboratories, June 2013.

[177] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVIDIA NVR-2008-004.

[178] E. Chow and A. Patel. Fine-grained parallel incomplete lu factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015, http://dx.doi.org/10.1137/140968896.