A class of linear solvers based on multilevel and supernodal factorization

Bu, Yiming

A Class of Linear Solvers Based on Multilevel and Supernodal Factorization

Yiming Bu

ISBN: 978-94-034-0727-2 (printed version)

ISBN: 978-94-034-0726-5 (electronic version)

Printed by: Ipskamp Printing, Enschede, The Netherlands.

university of
groningen

A Class of Linear Solvers Based on Multilevel and Supernodal Factorization

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. E. Sterken
and decision by the College of Deans.

This thesis will be defended in public on

Tuesday 3 July 2018 at 11.00 hours

by

Yiming Bu

born on 19 December 1986
in Hebei, China

Supervisor
Prof. A. Veldman

Co-supervisor
Dr. B. Carpentieri

Assessment committee
Prof. R. Bisseling
Prof. M. Ferronato
Prof. R. Verstappen

Contents

Acknowledgements

There is a proverb: “If you want to go quickly, go alone. If you want to go far, go together.” The way to a doctorate is a special life experience. All the people that I met along this experience have taught me many things, about science and about life. I appreciate that I went through this journey together with you.

First and foremost, I would like to thank my promotors, Arthur Veldman from University of Groningen and Tingzhu Huang from University of Electronic Science and Technology of China, for providing me the opportunity to carry out my doctoral studies. Your supervision teaches me what is scientific research and how to live a scientific life. I want to express my thanks through a Ubbo Emmius Scholarship.

I want to express my appreciation to Bruno Carpentieri, my supervisor as well as my great friend. You have good quality of professionalism and personality, and enlighten me both academically and in life. Thank you for the scientific and personal chats, and for being such a great supervisor and friend, which makes possible this thesis work.

I am grateful to Professors Rob Bisseling, Massimiliano Ferronato and Roel Verstappen, for being the members of the assessment committee of this thesis. Many thanks for your reviews, insightful comments and suggestions.

I feel fortunate and honored to have opportunity to share and discuss my research with great scientists, Miroslav Tůma and Michael Saunders. You have greatly encouraged and enlightened me on my future work.

Thanks to my fellow PhD students both in Groningen and in Chengdu, who have provided positive work environments and pleasant interactions: Yanfei Jing, Liang Li, Weiyan Song and Sourabh Kotnala. Many interesting and inspiring technical discussions have be done during the project.

I am thankful to the administrative staffs of University of Groningen and University of Electronic Science and Technology of China: Janieta de Jong-Schlukebir, Esmee Elshof, Ineke Schelhaas, Desiree Hansen, Ingrid Veltman, Annette Korringa, Liza van Eijck, Wenli Zhao, Yan Liu, Jiao Zhang, Puying Ye and Kunlong Li.

A special thanks to my paranymphs Astone and Sheng, for being by my side during the defence at University of Groningen. I am grateful to my dear friends Jiapan and Astone, Yanfang and Sheng, and Jinfeng. It must be our fate to study abroad, to meet in Groningen, and to be friends forever.

At the last but not the least, I want to express my immense gratitude to my parents, my sisters and brothers, for their encouragement and understanding. Special gratitude and love goes to my wife Yang, and my son Charles. You have been great support to me, a perfect partner and an adorable angel.

1 Introduction

1.1 Motivation and background

The solution of large and sparse linear systems is a critical component of modern science and engineering simulations, often accounting for up to 90% of the whole solution time. Iterative methods, namely the class of modern Krylov subspace methods, are often considered the method of choice for solving large-scale linear systems such as those arising from the discretization of partial differential equations on parallel computers. They are matrix-free and they can solve the memory bottlenecks of sparse direct methods which are known to scale poorly with the problem size. However, one practical difficulty of an efficient implementation of Krylov methods is that they lack the typical robustness of direct solvers. Preconditioning is a technique that can be used to reduce the number of iterations required to achieve convergence. In general terms, preconditioning can be defined as the science and art of transforming a problem that appears intractable into another whose solution can be approximated rapidly [98]. Nowadays it can be considered a crucial component of the linear systems solution.

One important class of preconditioning techniques for solving linear systems is represented by Incomplete LU decomposition (ILU) [73, 86]. ILU preconditioners have good robustness and typically exhibit fast convergence rate. Therefore, they are considered amongst the most reliable preconditioning techniques in a general setting. Well known theoretical results on the existence and the stability of the incomplete factorization can be proved for the class of M-matrices, and recent studies are involving more general matrices, both structured and unstructured. The quality of the factorization on difficult problems can be enhanced by using several techniques such as reordering, scaling, diagonal shifting, pivoting and condition estimators (see e.g. [18, 29, 43, 71, 89]). As a result of this active development, in the last years successful results are reported with ILU-type preconditioners in many areas that were of exclusive domain of direct methods, e.g., in circuits simulation, power system networks, chemical engineering plants modelling, graphs and other problems not governed by PDEs, or in areas where direct methods have been traditionally preferred, like in structural analysis, semiconductor device modelling and computational fluid dynamics applications (see e.g. [6, 14, 72, 87]). One problem with ILU-techniques is the severe degradation of performance observed on vector, parallel and GPU machines, mainly due to the sparse triangular solvers [70]. In some cases, reordering techniques may help to introduce nontrivial parallelism. However, parallel orderings may sometimes degrade the convergence rate, while more fill-in diminishes the overall parallelism of the solver [44].

On the other hand, sparse approximate inverse preconditioners approximate the inverse of the coefficient matrix of the linear system explicitly. They offer inherent parallelism compared to ILU methods as the application phase reduces to simply perform one or more sparse matrix-vector products, making them very appealing to solve large linear systems. Thus this class is receiving a renewed interest for implementation on massively parallel computers and hardware accelerators such as GPUs [26, 70, 74, 100]. Additionally, on certain indefinite problems with large nonsymmetric parts, the explicit approach can provide more numerical stability than ILU techniques (see e.g. [28, 35, 56]). In practice, however, some questions need to be addressed. The computed preconditioning matrix M could be singular, and the construction cost is typically much higher than for ILU-type methods, especially for sequential runs. The main issue is the selection of the non-zero pattern of M. The idea is to keep M reasonably sparse while trying to capture the 'large' entries of the inverse, which are expected to contribute the most to the quality of the preconditioner. On general problems it is difficult to determine the best structure for M in advance, and the computational and storage costs required to achieve the same rate of convergence of preconditioners given in implicit form may be prohibitive in practice.

In this thesis, we present an algebraic multilevel solver for preconditioning general nonsymmetric linear systems that attempts to combine characteristics of both approaches. Sparsity in the preconditioner is maximized by employing recursive combinatorial algorithms. Robustness is enhanced by combining the factorization with recently developed overlapping and compression strategies, and by using efficient local solvers. Our goal is to propose new preconditioning strategies and ideas that have a good deal of generality, so that they can improve parallelism and robustness of iterative solvers, while being economic to implement and use. We follow a purely algebraic approach, where the preconditioner is computed using only information available from the coefficient matrix of the linear system. Although not optimal for any specific problem, algebraic methods can be applied to different problems and to changes in the geometry by only tuning a few parameters.

We introduce a novel Algebraic Multilevel Explicit Solver (referred to as AMES), which is a hybrid recursive multilevel incomplete factorization preconditioner based on a distributed Schur complement formulation. The AMES method uses the nested dissection ordering to create a multilevel arrow structure, so that most nonzero entries of the original matrix are clustered into a few nonzero blocks. We assess the overall performance of AMES, and investigate the choice of parameters. Moreover, we also introduce a combination of the overlapping strategy with the AMES solver. The usage of the overlapping strategy preserves the multilevel block arrow structure of the original matrix, as well as helps improve the convergence rate and solving time by adding more information. The algorithm and performance of the combinatorial method are elaborated upon. We demonstrate that the proposed strategies can also be incorporated in other existing preconditioning and iterative solver packages in use today.

Based on the AMES algorithm, we propose an improved implicit variant, referred to as Algebraic Multilevel Implicit Solver (AMIS), also built upon the multilevel recursive nested dissection reordering. In AMIS we modify the solve phase and apply the preconditioner implicitly without using the explicit computation of the approximate inverse. This modification can save much of the recursive computation done in the AMES algorithm which is quite time consuming. Numerical experiments are presented to compare the overall performance of AMIS against AMES.

Finally, we propose a variable block variant of the AMIS solver (referred to as VBAMIS) to exploit the presence of dense components in the solution to the linear system. Sparse matrices arising from the discretization of partial differential equations often exhibit some small and dense nonzero blocks in the sparsity pattern, e.g., when several unknown quantities are associated with the same grid point. Such block orderings can be sometimes unravelled also on general unstructured matrices, by ordering consecutively rows and columns with a similar sparsity pattern, and treating some zero entries of the reordered matrix as nonzero elements. It is possible to take advantage of these structures in the design of AMIS for better numerical stability and higher efficiency on cache-based computer architectures. Compression techniques are used to discover dense blocks in the sparsity pattern of the coefficient matrix. Variable block compressed sparse row (VBCSR) storage formats and high-level BLAS (Basic Linear Algebra Subprograms) [40] routines are employed to achieve better cache performance on supercomputers.

The thesis is organized as follows. In Chapter 2 we review modern Krylov subspace methods and preconditioning techniques for solving large linear systems, with particular attention to Incomplete LU factorization (ILU) and sparse-approximate inverse-based preconditioners. In Chapter 3 we introduce the new solver AMES. We present the main computational steps of the method, and assess its performance for solving matrix problems arising from various applications against some state-of-the-art solvers. We also discuss different parameter settings of the new method and put forward suggestions of a reasonable parameter setting. In Chapter 4 we present a combination of an overlapping strategy with the AMES solver. We recall the theoretical background of overlapping, and present numerical experiments to illustrate the effect of overlapping. In Chapter 5 we propose an implicit formulation of AMES, referred to as AMIS. We also present comparative experiments on AMES and AMIS. Finally, in Chapter 6, we propose a variable block variant referred to as VBAMIS.

2 Iterative methods for sparse linear systems

Mathematical models of multiphysics applications often consist of systems of partial differential equations (PDEs), generally coupled with systems of ordinary differential equations. Current numerical methods and softwares to solve such models do not provide the required computational speed for practical applications. One reason for this is that limited use is made of recent developments in (parallel) numerical linear algebra algorithms. The numerical solution of PDEs is usually computed using standard finite-difference (FD) [67, 95], finite-element (FE) [91, 99], finite-volume (FV) [19, 20], discontinuous Galerkin (DG) [3, 36] or spectral element (SE) methods [66, 81]. The FE method is often used as it can encapsulate the complex geometry and small-scale details of the boundary. The spatial discretization of the underlying governing PDEs leads to very large and sparse linear systems that must be solved at each time step of the simulation, regardless of the type of numerical method employed. Sparse direct methods based on variants of Gaussian elimination are robust and numerically accurate algorithms, but they are too memory demanding for solving very large linear systems even on modern parallel computers. Iterative methods, namely modern Krylov subspace solvers, are based on matrix-vector (M-V) multiplications. They can solve the memory bottleneck of direct methods and present a viable alternative, provided they are accelerated with some form of preconditioning. In Sections 2.1 and 2.2, we propose a short review of modern Krylov subspace methods and preconditioning techniques for solving large linear systems as they form the basis of our development.

2.1 Krylov subspace methods

We denote our linear system to solve as

$$Ax = b, \tag{2.1}$$

where the coefficient matrix A is a large, sparse and unsymmetric $n \times n$ matrix, and b is the right-hand side vector.

There are two principal computational approaches for solving system (2.1), that are *direct methods* and *iterative methods*. Direct methods are based on variants of the Gaussian Elimination (GE) algorithm [42]. They are robust and predictable in terms of both accuracy and computing cost, but their $\mathcal{O}(n^2)$ memory complexity and $\mathcal{O}(n^3)$ algorithmic complexity, where n can be in the order of several millions of unknowns, are too demanding for solving practical applications. In this thesis we focus our attention on iterative methods. Iterative methods are matrix-free and hence they can solve the memory problems of direct methods. At each iteration, iterative methods improve the current approximation towards convergence. The computation requires M-V products with the coefficient matrix A plus vector operations, thus potentially reducing the heavy algorithmic and storage costs of sparse direct solvers on large problems.

The early development of iterative methods was based on the idea of splitting the coefficient matrix A into the sum of two matrices. The splitting $A = I - (I - A)$ produces the well-known Richardson iteration

$$x_i = b + (I - A)x_{i-1} = x_{i-1} + r_{i-1}, \tag{2.2}$$

where $r_{i-1} \equiv b - Ax_{i-1}$ is the $(i-1)$-th residual vector. Multiplying Equation (2.2) by $-A$ and adding b, we obtain

$$b - Ax_i = b - Ax_{i-1} - Ar_{i-1}$$

that can be written as

$$r_i = (I - A)r_{i-1} = (I - A)^i r_0 = P_i(A)r_0, \tag{2.3}$$

where $r_0 = b - Ax_0$ is the initial residual vector. From Equation (2.3), fast convergence is possible when $\|I - A\|_2 \ll 1$ [41]. In Equation (2.3) the residual r_i is expressed as a polynomial of A times r_0. For the standard Richardson iteration, the residual polynomial has the form $P_i(A) = (I - A)^i$.

The Richardson iteration can be generalized by using splitting of the form $A = K - (K - A)$, where the matrix K is an approximation of A and is easy to invert. The iteration scheme writes as follows

$$Kx_i = b + (K - A)x_{i-1} = Kx_{i-1} + r_{i-1},$$

$$x_i = x_{i-1} + K^{-1} r_{i-1}.$$

In general, the inverse matrix K^{-1} is never computed explicitly. In the above equations, if we define $B = K^{-1}A$ and $c = K^{-1}b$, then the splitting with K is equivalent with the standard Richardson method applied to the equivalent system $Bx = c$. Therefore, the matrix K^{-1} can be viewed as an operator that transforms the original coefficient matrix A into a new matrix which is close to I. This transformation is called preconditioning, which is the subject of this thesis, and will be introduced in Section 2.2.

Without any loss of generality, we can choose the initial guess to be $x_0 = 0$. From the Richardson iteration (2.2), it follows that

$$x_{i+1} = r_0 + r_1 + \ldots + r_i = \sum_{j=0}^{i} (I - A)^j r_0,$$

hence $x_{i+1} \in span\{r_0, Ar_0, A^2 r_0, \ldots, A^i r_0\} \equiv \mathcal{K}_{i+1}(A, r_0)$ [41]. Here the subspace $\mathcal{K}_i(A, r_0)$ is called the Krylov subspace of dimension i generated by A and r_0. This tells us that the Richardson iteration computes elements of Krylov subspaces of increasing dimension in the attempt to solve the linear system.

From the theory of Krylov subspace methods, the dimension m of the space $\mathcal{K}_m(A, r_0)$ where the exact solution lies depends on the distribution of the eigenvalues of A [57]. If A has k distinct eigenvalues denoted by $\lambda_1, \lambda_1, \ldots, \lambda_k$, then the minimal polynomial $P_m(t)$ of A can be constructed as follows

$$P_m(t) = \prod_{j=1}^{k} (t - \lambda_j)^{m_j}, \tag{2.4}$$

where λ_j has multiplicity m_j and $m = \sum_{j=1}^{k} m_j$. The minimal polynomial $P_m(t)$ of A is the unique monic polynomial of minimal degree such that $P_m(A) = 0$.

If we write Equation (2.4) into the form

$$P_m(t) = \sum_{j=0}^{m} a_j t^j,$$

then it follows that

$$P_m(A) = a_0 I + a_1 A + \ldots + a_m A^m = 0,$$

where the constant term $a_0 = \prod_{j=1}^{k} (\lambda_j)^{m_j}$. For nonsingular matrix A, $a_0 \neq 0$. Hence the inverse of matrix A can be expressed as

$$A^{-1} = -\frac{1}{a_0} \sum_{j=0}^{m-1} a_{j+1} A^j.$$

The above equation describes the solution $x = A^{-1}b$ as a linear combination of vectors from a Krylov subspace

$$x = -\sum_{j=0}^{m-1} \frac{\alpha_{j+1}}{\alpha_0} A^j b.$$

Since $x_0 = 0$, then we have

$$x = -\sum_{j=0}^{m-1} \frac{\alpha_{j+1}}{\alpha_0} A^j b = -\sum_{j=0}^{m-1} \frac{\alpha_{j+1}}{\alpha_0} A^j r_0,$$

hence the solution x lies in the Krylov subspace $\mathcal{K}_m(A, r_0)$ [41].

Generally, the Richardson method may suffer from slow convergence. Hence over the last three decades, many new faster, more powerful and robust Krylov methods that extract better approximations from the Krylov subspace than Richardson were developed for solving large nonsymmetric systems.

In most of our experiments we used the Generalized Minimum Residual Method (GMRES) [83]. Hence we briefly recall the GMRES method and the minimum residual approach in the first place.

2.1.1 The Generalized Minimum Residual Method (GMRES)

The minimum residual approach requires $\|b - Ax_k\|_2$ to be minimal over $\mathcal{K}_k(A, r_0)$. In the category of orthogonal methods, the Generalized Minimum Residual Method (GMRES) [83] is one of the most popular algorithms. GMRES is a projection method based on the Arnoldi process. Hence we first recall the Arnoldi method briefly as the prerequisites.

The Arnoldi method [4] described in Algorithm 2.1 is an orthogonal projection method onto the Krylov subspace $\mathcal{K}_m$. It generates an orthogonal basis $\{v_1, v_2, \ldots, v_m\}$ of the Krylov subspace $\mathcal{K}_m$.

Algorithm 2.1 *Arnoldi method.*

```
Choose a vector $v_1$ such that $\|v_1\|_2 = 1$
for $j = 1, 2, \ldots, m$ do
    for $i = 1, 2, \ldots, j$ do
        Compute $h_{i,j} = (Av_j, v_i)$
    end for
    Compute $w_j = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$
    $h_{j+1,j} = \|w_j\|_2$
    If $h_{j+1,j} = 0$, stop
    $v_{j+1} = w_j / h_{j+1,j}$
end for
```

Let V_m denote the $n \times m$ matrix with columns being $v_1, v_2, \ldots, v_m$. Also, let $H_{m+1,m}$ denote the upper $(m+1) \times m$ Hessenberg matrix with entries $h_{i,j}$, and H_m denote the square matrix with dimension m. According to Algorithm 2.1, the following relation holds

$$Av_j = \sum_{i=1}^{j+1} h_{i,j} v_i, j = 1, 2, \ldots, m.$$

This leads to

$$AV_m = V_{m+1} H_{m+1,m}. \tag{2.5}$$

Multiply the above equation by V_m^T from both sides, and we have

$$V_m^T A V_m = V_m^T V_{m+1} H_{m+1,m}.$$

With the orthonormality of $\{v_1, \ldots, v_m\}$, the product $V_m^T V_{m+1}$ produces an $m \times (m+1)$ matrix, whose upper $m \times m$ part is an identity matrix I_m and the $(m+1)$-th row is an empty row. Hence

$$V_m^T A V_m = H_m,$$

where H_m has the same entries $h_{i,j}$ as in $H_{m+1,m}$. The matrix H_m can be viewed as being obtained from $H_{m+1,m}$ by deleting the last row.

The first step of the projection process is to find a basis for the Krylov subspace. The Arnoldi process starts by computing orthonormal basis vectors $v_1, v_2, \ldots, v_m$

as a result of the modified Gram-Schmidt procedure. This results in matrix V_m with orthonormal columns that meets Equation (2.5).

GMRES computes the approximate solution by minimizing the residual norm over all the vectors in $x_0 + \mathcal{K}_m$. Define

$$J(y) = \|b - Ax_m\|_2 = \|b - A(x_0 + V_m y)\|_2, \tag{2.6}$$

then the approximate solution that minimizes (2.6) is computed by GMRES. Equation (2.5) leads to

$$\begin{aligned} b - Ax_m &= b - A(x_0 + V_m y) \\ &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1} H_{m+1,m} y \\ &= V_{m+1}(\beta e_1 - H_{m+1,m} y), \end{aligned}$$

where $\beta = \|r_0\|_2$ and v_1 is chosen so that $v_1 = r_0/\beta$. Since the columns of V_{m+1} are orthonormal vectors, then

$$J(y) \equiv \|b - A(x_0 + V_m y)\|_2 = \|\beta e_1 - H_m y\|_2. \tag{2.7}$$

By Equation (2.7), the approximation solution x_m can be computed as

$$x_m = x_0 + V_m y_m, \text{ where } y_m = argmin_y \|\beta e_1 - H_m y\|_2. \tag{2.8}$$

The minimizer y_m is inexpensive to compute since it can be calculated from an $(m+1) \times m$ least-squares problem which is typically small. To solve the least-squares problem $\|\beta e_1 - H_m y\|_2$, the Hessenberg matrix is often transformed into upper triangular form by using Givens rotations [83]. The rotation matrices are defined as

$$\Omega_i = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}$$

where c_i and s_i are chosen to meet $c_i^2 + s_i^2 = 1$ and to eliminate entry $h_{i+1,i}$ of the Hessenberg matrix. With the rotation operations with $\Omega_1, \Omega_2, \ldots, \Omega_m$, the

Hessenberg matrix H_m is transformed into the upper triangular matrix R_m. Set $Q_m = \Omega_m \Omega_{m-1} \dots \Omega_1$. Then $R_m = Q_m H_m$, and βe_1 is transformed into $g_m = Q_m \beta e_1$.

Since Q_m is unitary, the least-squares problems can be expressed as

$$min\|\beta e_1 - H_m y\|_2 = min\|g_m - R_m y\|_2. \tag{2.9}$$

The new least-squares problem (2.9) can be solved as $y_m = R_m^{-1} g_m$ (see Proposition 6.9 in [86] for detail).

The implementation of the GMRES method is illustrated in Algorithm 2.2.

Algorithm 2.2 *GMRES*

```
 1: Choose $x_0$; compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$ and $v_1 := r_0/\beta$.
 2: for $j = 1 : m$ do
 3:     Compute $w_j := Av_j$
 4:     for $i = 1 : j$ do
 5:         $h_{i,j} = (w_j, v_i)$
 6:         $w_j := w_j - h_{i,j} v_i$
 7:     end for
 8:     $h_{j+1,j} = \|w_j\|_2$.
 9:     if $h_{j+1,j} = 0$ then
10:         set $m := j$ and go to 14
11:     end if
12:     $v_{j+1} = w_j / h_{j+1,j}$
13: end for
14: Define the $(m+1) \times m$ Hessenberg matrix $H_m = \{h_{i,j}\}$, where $1 \le i \le m+1, 1 \le j \le m$.
15: Compute the minimizer $y_m$ of $\|\beta e_1 - H_m y\|_2$ and $x_m = x_0 + V_m y_m$.
```

As the dimension m increases, the computational and memory costs of each iteration would increase dramatically. One remedy could be to use a restart strategy, which leads to the restarted GMRES [45] method. In the restarted variant, the GMRES algorithm 2.2 is restarted every r steps, and the approximate solution x_r is used as initial guess for the next GMRES process until convergence is reached. But when the method restarts, the existing Krylov subspace is also destroyed and a new one is created from the ground up damaging convergence.

2.1.2 Other approaches

Although the GMRES method is mostly used in the experiments proposed in the following chapters, the preconditioners developed in this thesis are independent on the choice of the specific Krylov method. Occasionally, they will be combined with some other families of Krylov solvers. Thus, below we present a quick overview of other popular Krylov subspace methods for solving linear systems (2.1) [41].

The Ritz-Galerkin approach

The Ritz-Galerkin approach requires that $(b - Ax_k) \perp \mathcal{K}_k(A, r_0)$. This leads to popular and well-known orthogonal projection methods, such as the Full Orthogonalization Method (FOM) [86] for general non-symmetric matrices and the Conjugate Gradients (CG) [54] for symmetric, positive and definite (SPD) matrices.

For solving Equation (2.1), a *projection technique* could be an effective solving method. When the approximation $\tilde{x}$ is obtained from an m-dimensional subspace $\mathcal{K}_m$, the residual vector $b - A\tilde{x}$ must be constrained to be orthogonal to m linearly independent vectors, which form a basis of the m-dimensional subspace $\mathcal{L}$. The projection process is applied onto $\mathcal{K}$ and orthogonal to $\mathcal{L}$. The approximate solution $\tilde{x}$ is extracted by imposing the conditions that $\tilde{x}$ belongs to $\mathcal{K}$ and the residual vector $b - A\tilde{x}$ belongs to $\mathcal{L}^{\perp}$.

$$\text{Find } \tilde{x} \in \mathcal{K}, \text{ such that } b - A\tilde{x} \perp \mathcal{L}. \tag{2.10}$$

When the initial guess x_0 to the solution is given, the projection process (2.10) can be refined as

$$\text{Find } \tilde{x} \in x_0 + \mathcal{K}, \text{ such that } b - A\tilde{x} \perp \mathcal{L}.$$

In Section 2.1.1 we have recalled the Arnoldi method which is necessary in GMRES to compute the Krylov basis. If the initial vector v_1 in the Arnoldi method is chosen as $v_1 = r_0/\beta$, where $r_0 = b - Ax_0$ is the initial residual vector and $\beta = \|r_0\|_2$, then we have $r_0 = \beta v_1$. According to the definition of V_m, we have $V_m^T v_k = e_k$, where $1 \leq k \leq m$. Then the following equation holds

$$V_m^T r_0 = V_m^T(\beta v_1) = \beta e_1,$$

and the approximate solution is computed by

$$\begin{aligned} x_m &= x_0 + A^{-1} r_0 \\ &= x_0 + V_m (V_m^T A V_m)^{-1} V_m^T r_0, \\ &= x_0 + V_m H_m^{-1} (\beta e_1). \\ &= x_0 + V_m y_m, \end{aligned} \tag{2.11}$$

where $y_m = H_m^{-1}(\beta e_1)$.

Based on these derivations, the algorithm of Full Orthogonalization Method (FOM) is given in Algorithm 2.3.

Algorithm 2.3 *Full Orthogonalization Method (FOM).*

```
1:  Given an initial guess $x_0$, compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2, v_1 := r_0/\beta$
2:  Define $m \times m$ matrix $H_m = (h_{i,j})_{i,j=1,\dots,m}$; set $H_m = 0$
3:  for $j = 1, 2, \dots, m$ do
4:      Compute $w_j = Av_j$
5:      for $i = 1, 2, \dots, j$ do
6:          $h_{i,j} = (w_j, v_i)$
7:          $w_j = w_j - h_{i,j} v_i$
8:      end for
9:      Compute $h_{j+1,j} = \|w_j\|_2$.
10:     if $h_{j+1,j} = 0$ then
11:         Set $m := j$ and goto 15
12:     end if
13:     Compute $v_{j+1} = w_j / h_{j+1,j}$.
14: end for
15: Compute $y_m = H_m^{-1}(\beta e_1)$ and $x_m = x_0 + V_m y_m$
```

The Conjugate Gradient (CG) method is another effective Krylov method based on the Ritz-Galerkin approach. It is based on short-term vector recurrences, and may be considered the method of choice for solving large symmetric and positive definite linear systems. Since the focus of thesis is on the solution of nonsymmetric systems, we skip the derivation of the CG method.

The Petrov-Galerkin approach

The Petrov-Galerkin approach requires that $b - Ax_k$ is orthogonal to some other suitable k-dimensional subspace. This leads to bi-orthogonalization methods like the Bi-orthogonal version of CG (Bi-CG) and QMR [48] methods. Thereafter, hybrid variants of these approaches have been proposed, such as the Conjugate Gradient Squared (CGS) algorithm [93] and the Bi-conjugate Gradient Stabilized (BI-CGSTAB) method [39].

The Biconjugate Gradient (BiCG) method was first proposed by Lanczos [68]. BiCG can be interpreted as the nonsymmetric variant of CG. The difference is that A^T is used in the BiCG recurrence instead of A. Below we present the BiCG algorithm.

Algorithm 2.4 *The BiCG method.*

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$
Choose $r_0^*$ (for example, $r_0^* = r_0$)
Let $p_0 = r_0$, $p_0^* = r_0^*$
for $i = 0, 1, 2, \ldots$ do
    $\alpha_i = (r_i, r_i^*)/(Ap_i, p_i^*)$
    $x_{i+1} = x_i + \alpha_i p_i$
    $r_{i+1} = r_i - \alpha_i Ap_i$
    $r_{i+1}^* = r_i^* - \alpha_i A^T p_i^*$
    $\beta_i = (r_{i+1}, r_{i+1}^*)/(r_i, r_i^*)$
    $p_{i+1} = r_{i+1} + \beta_i p_i$
    $p_{i+1}^* = r_{i+1}^* + \beta_i p_i^*$
    Check convergence; continue if necessary
end for
```

BiCG not only solves the original system $Ax = b$, but also solves the dual linear system $A^T x^* = b^*$ implicitly. The residual vectors r_i in CG cannot be made orthogonal with short recurrences, which makes it not suitable for nonsymmetric systems. In contrast, BiCG replaces the orthogonal sequence of residuals r_i by two mutually orthogonal sequences, r_i and r_i^*. These two sequences of vectors are based on A and A^T respectively. The two sequences are made bi-orthogonal rather than orthogonalizing each sequence. BiCG obtains approximations from the Krylov subspace by means of the bi-orthogonal basis approach, which enables BiCG to solve nonsymmetric systems.

The Conjugate Gradient Squared (CGS) method proposed by Sonneveld [93] aims to obtain a faster convergence rate with the same computational costs, yet without involving A^T as in BiCG. In BiCG it is necessary to compute r_i^*. However, the CGS method avoids the construction of r_i^* and the computation with A^T.

This results in the CGS algorithm given below. The computational costs of each iteration are almost the same for BiCG and CGS, but CGS has an advantage of avoiding the computation with respect to the r_i^* and A^T. Hence CGS can be considered as the transpose-free variant of BiCG. In general, CGS is expected to exhibit faster convergence than BiCG with roughly the same computational and memory cost. When solving SPD problems, BiCG and CGS produce the same x_i and r_i as CG and do not break down. For the unsymmetric cases, there is no thorough report on convergence analysis yet. But numerical tests show that with an appropriate preconditioning strategy, CGS could be an effective solver for the unsymmetric problems.

Algorithm 2.5 *The CGS method.*

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$, choose arbitrary $r_0^*$
Choose $r_0$ arbitrarily
Let $p_0 = u_0 = r_0$
for $i = 0, 1, 2, \ldots$ do
    $\alpha_i = (r_i, r_0^*)/(Ap_i, r_0^*)$
    $q_i = u_i - \alpha_i Ap_i$
    $x_{i+1} = x_i + \alpha_i(u_i + q_i)$
    $r_{i+1} = r_i - \alpha_i A(u_i + q_i)$
    $\beta_i = (r_{i+1}, r_0^*)/(r_i, r_0^*)$
    $u_{i+1} = r_{i+1} + \beta_i q_i$
    $p_{i+1} = u_{i+1} + \beta_i(q_i + \beta_i p_i)$
    Check convergence; continue if necessary
end for
```

Recently, Sonneveld and van Gijzen have developed a family of efficient methods, called IDR(s), for solving large nonsymmetric systems of linear equations [94]. These methods are based on the induced dimension reduction (IDR) method proposed by Sonneveld in 1980 [101]. They generate residuals that are forced to be in a sequence of nested subspaces. It is shown that the IDR methods can be interpreted as a Petrov-Galerkin process over particularly chosen block Krylov subspaces. The IDR methods can fully exploit the implicitly imposed orthogonality and is quite effective for solving many problems. Subsequently, Simoncini and Szyld present a new version of IDR, which is called Ritz-IDR method [92]. The Ritz-IDR method could overcome the difficulties encountered when using a small block size for building the block subspace. The Ritz-IDR method has been verified to work well on convection-diffusion problems with definite spectrum.

The minimum error approach

The minimum error approach requires $\|x - x_k\|_2$ to be minimal over $A^T\mathcal{K}_k(A^T, r_0)$. It is not so obvious that we could minimize the error. However, the special form of the subspace makes this possible. Methods of this category include Conjugate Gradient on the Normal equations, CGNR ("Residual") and CGNE ("Error").

Given a linear system of equations $Ax = b$ with nonsymmetric A, the normal equation is defined as

$$A^TAx = A^Tb. \tag{2.12}$$

or

$$(AA^T)y = b, \text{ with } x = A^Ty. \tag{2.13}$$

The CGNR and CGNE methods are constructed by applying the CG method to the normal equations. CGNR solves Equation (2.12), and CGNE solves Equation (2.13). For nonsymmetric and nonsingular coefficient matrix A, the normal equations matrices AA^T and A^TA are symmetric and positive definite, and CG can be applied. However, in this case the convergence rate of CG depends on the square of the condition number of A, which makes CG converge slowly. The LSQR algorithm proposed by Paige and Saunders [77] applies the Lanczos process to the *damped least squares* problem

$$min\left\|\begin{pmatrix} A \\ \lambda I \end{pmatrix} x - \begin{pmatrix} b \\ 0 \end{pmatrix}\right\|_2, \tag{2.14}$$

where λ is an arbitrary scalar. The solution to (2.14) satisfies the symmetric system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix}\begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix},$$

where $r = b - Ax$. The LSQR method is analytically equal to the CG method, but possesses better robustness.

2.1.3 Concluding remarks

For a given problem, choosing an appropriate method is not an easy task. Important selection criteria are the storage costs, the availability of A^{-1}, the computational costs of the M-V products, `SAXPY`s and inner products. Table 2.1 lists the operations performed per iteration and storage costs of the Krylov subspace methods introduced above.

Method	Inner Product	`SAXPY`	M-V Product	Storage costs
BiCG	2	5	1/1	matrix + $10n$
CGS	2	6	2	matrix + $11n$
GMRES	$i+1$	$i+1$	1	matrix + $(i+5)n$

Table 2.1: Operational and storage costs for the methods in the i-th iteration. Here "1/1" means 1 multiplication with the matrix and 1 with its transpose.

There is a possibility that one Krylov subspace converges slowly and no good approximate solution can be found or such an approximate solution cannot be obtained easily. In this case it is recommended to use a preconditioner to converge faster.

2.2 Preconditioning

Iterative methods require M-V products involving the coefficient matrix of the linear system plus vector operations, which can potentially solve the memory bottlenecks of sparse direct solvers. However, as we discussed at the end of Section 2.1, there are many occasions where iterative methods converge slowly or even fail to converge. Under this circumstance, we aim at computing a matrix M such that the solution to the equivalent system $MAx = Mb$ or $AMy = b$, can be obtained in a smaller number of iterations of a Krylov method. This idea is called *preconditioning*. The matrix M, which is called the *preconditioner*, should approximate the inverse of the coefficient matrix A.

It is widely recognized that preconditioning is a critical ingredient in the development of efficient solvers for modern computational science and engineering applications. In practice, with the availability of a high quality preconditioner, the choice of the Krylov subspace accelerator is often not so critical, see e.g. conclusions in [51,86]. Some preconditioning approaches often work well only for a narrow class of problems. In this thesis we propose new preconditioning methods for solving general nonsymmetric linear systems. We pursue a purely algebraic approach where the preconditioner is computed from the coefficient matrix of the linear system. Although not optimal for any specific problem, algebraic methods are universally applicable. They can be adapted to different operators and to changes in the geometry by tuning only a few parameters, and are suited for solving irregular problems defined on unstructured meshes.

A good preconditioner M should satisfy the following properties

1. M is a good approximation to A^{-1} in some sense.

2. The cost of the construction of M is not prohibitive.

3. The new preconditioned system is much easier to solve than the original one.

At every iteration step, the operation $y = MAx$ for left preconditioning is computed in two steps: we first compute $z = Ax$ and then we obtain y as $y = Mz$. When solving the preconditioned system

$$MAx = Mb$$

by a Krylov subspace method, the approximate solution x_i at step i belongs to the Krylov subspace

$$span\{Mr_0, (MA)(Mr_0), \ldots, (MA)^{i-1}(Mr_0)\}. \tag{2.15}$$

By a suitable choice of M, we may expect that the dimension of the preconditioned Krylov space where the exact solution lies is much smaller compared to the unpreconditioned one, so that convergence may be faster.

There are three different ways to implement a preconditioner M, as described below, leading to different convergence behaviors in general.

1. **Left-preconditioning.** Apply the iterative method to the transformed linear system
$$MAx = Mb. \tag{2.16}$$
All residual vectors and norms computed by a Krylov method correspond in this case to the preconditioned residuals $M(b - Ax_k)$ rather than the original residuals $b - Ax_k$. This may have an impact on the stopping criteria based on the residual norms.

2. **Right-preconditioning.** Apply the iterative method to the transformed system

$$AMy = b, \; x = My. \tag{2.17}$$

Right-preconditioning has the advantage that it only affects the operator and not the right-hand side. Therefore the preconditioned and unpreconditioned residual vectors are the same.

3. **Split (Two-sided) preconditioning.** The preconditioning process can be written in the form

$$M_1AM_2z = M_1b, \; x = M_2z. \tag{2.18}$$

Then we apply the iterative method to (2.18). This form of preconditioning may be useful especially for preconditioners that come in factored form. It can be seen as a compromise between left- and right-preconditioning. This form may be also used to obtain a (near) symmetric operator for situations where M cannot be used for the definition of an inner product (as described under left-preconditioning).

The convergence rate is determined by the approximation degree of M to A^{-1}. In one extreme case, if M is equal to A^{-1}, the convergence will be reached in one step. But in this case the construction of the preconditioner is equivalent to solving the original problem. In general, the preconditioner M should be easy to construct and to apply, and the total computing time for solving the preconditioned system be less than those for the original one.

2.2.1 Explicit and implicit form of preconditioning

Most of the existing preconditioning methods can be divided into *implicit* and *explicit* form. A preconditioner of *implicit* form is defined by a nonsingular matrix $M \approx A^{-1}$ and requires to solve a linear system at each step of an iterative method. One of the most important examples in this class is Incomplete LU decomposition (ILU), where M is implicitly defined by $M = \overline{LU}$. Here $\overline{L}$ and $\overline{U}$ are triangular matrices that approximate the exact L and U factors of A^{-1}, according to some dropping strategy adopted in the Gaussian elimination algorithm. Well known theoretical results on the existence and stability of the factorization can be proved for the class of M-matrices [73], and recent studies involve more general matrices, both structured and unstructured. Many techniques can help improve the quality of the factorization for solving more general problems, such as reordering, scaling, diagonal shifting, pivoting and condition estimators [18, 29, 43, 71, 89]. Successful computational experience using ILU-type preconditioners have been reported in many areas, such as circuit simulation, power system networks, chemical engineering, structural analysis, semiconductor device modelling and computational fluid dynamics [6, 14, 72, 87]. However, one problem is that the sparse triangular solvers can lead to a severe degradation of performance of ILU-techniques on highly parallel and GPU machines [70]. For some cases, reordering techniques may help to introduce nontrivial parallelism. But in many cases parallel orderings may degrade the convergence rate, while more fill-ins diminish the parallelism of the solver [44].

On the contrary, *explicit* preconditioning directly approximates A^{-1}, so that the preconditioning operation reduces to forming one (or more) sparse M-V product. Consequently, the application of the preconditioner may be easier to parallelize. Some of these explicit techniques can also perform the construction phase in parallel [30, 55, 58, 78, 79]. On certain indefinite problems, these methods have provided better results than ILU techniques (see e.g. [28, 35, 56]), representing an efficient alternative in the solution of difficult applications. In practice, however, some questions need to be addressed. First of all the resulting matrix M could be singular, so that the new transformed linear system is no longer equivalent to the original one. In the second place, explicit techniques usually require more CPU-time to compute the preconditioner than ILU-type methods. The main issue is the selection of the nonzero pattern of M. The idea is to keep M reasonably sparse while trying to capture the large entries of the inverse, which are expected to contribute the most to the quality of the preconditioner. Generally speaking, it is difficult to determine the best nonzero pattern for M and it is often necessary to increase the density of the computed sparse approximate inverse in order to have the same rate of convergence as attained by implicit preconditioners. This may lead to prohibitive computational and storage costs.

In the sections below we describe more extensively the popular classes of Incomplete LU factorization (ILU) and sparse-approximate-inverse-based preconditioners, which are the prerequisites for understanding our research.

2.2.2 Incomplete LU factorization

An ILU factorization can be derived by performing the standard Gaussian elimination process and choosing a proper sparsity pattern for the nonzero entries of the approximate triangular factors of A. Let A_1 be the matrix obtained from the first step of the Gaussian elimination process applied to A.

$$A_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}.$$

We define the user-selected nonzero pattern P of a target matrix M as

$$P = \{(i,j) \in [1,n]^2 \; s.t. \; m_{ij} \neq 0\},$$

and the nonzero pattern of a given matrix A as

$$NZ(A) = \{(i,j) \in [1,n]^2 \; s.t. \; a_{ij} \neq 0\}.$$

The nonzero pattern P can be selected in advance, and typical options are in general $P = NZ(A), P = (NZ(A))^2, \dots$. Some small entries outside of the main diagonal of A_1 can be 0 according to the selected nonzero pattern P. The resulting matrix is denoted by $\tilde{A}_1$.

We use $\tilde{A}_1(\mathcal{I}, \mathcal{J})$ to denote the submatrix of $\tilde{A}_1$, with entries a_{ij} of $\tilde{A}_1$ such that $i \in \mathcal{I}$ and $j \in \mathcal{J}$ with $\mathcal{I}$ and $\mathcal{J}$ subsets of $[1,n]$. Here we define $\mathcal{I}_i = \{i, i+1, \dots, n\}$. At each step, the Gaussian elimination process is repeated on the matrix $\tilde{A}_1(\mathcal{I}_i, \mathcal{I}_i)$, $i = 2, 3, \dots, n$, until the incomplete factorization of A is obtained. Note that the entries of $\tilde{A}_i$ are updated at each elimination step.

Below we describe the idea behind the general ILU factorization which is denoted as ILU_P. Here P represents the selected nonzero pattern as introduced above.

The Gaussian elimination procedure contains three loops that can be implemented in several ways. Algorithm 2.6 computes the outer product update and can be viewed as the KIJ version of Gaussian elimination. Similarly, there are other variants of Gaussian elimination implementations such as the IKJ variant, that can be incorporated with the ILU_P preconditioner [86].

If the nonzero pattern P is chosen to be precisely the nonzero pattern of A and no fill-in is allowed, then we have the so-called ILU(0) preconditioner. In the ILU(0) preconditioner, the triangular factors have the same pattern as the lower and upper triangular parts of A. We denote by a_{i*} the i-th row of A, by $a_{ij}, 1 \leq i,j \leq n$ the entries of A.

Algorithm 2.6 *General Static ILU factorization.*

```
for $k = 1 : n - 1$ do
    for $i = k + 1 : n$ and if $(i, k) \in P$ do
        $a_{ik} := a_{ik}/a_{kk}$
        for $j = k + 1 : n$ and if $(i, j) \in P$ do
            $a_{ij} := a_{ij} - a_{ik} \times a_{kj}$
        end for
    end for
end for
```

Algorithm 2.7 *IKJ variant of general ILU factorization: ILU(0).*

```
for $i = 2 : n$ do
    for $k = 1 : i - 1$ and if $(i, k) \in P$ do
        $a_{ik} := a_{ik}/a_{kk}$
        for $j = k + 1 : n$ and if $(i, j) \in P$ do
            $a_{ij} := a_{ij} - a_{ik}a_{kj}$
        end for
    end for
end for
```

Observe Figure 2.1 for example. In Figure 2.1 we denote nonzeros by the little squares. Matrix A is illustrated in the bottom left part. L and U are triangular factors so that LU approximates A^{-1}. At the top of Figure 2.1 we show the matrix L possessing the same structure as the lower part of A, and the matrix U possessing the same structure as the upper part of A. If we execute the product LU, we would obtain the matrix possessing the pattern shown in the bottom right part of Figure 2.1. Here we see that the product LU has more nonzeros than A. The extra nonzeros are denoted by hollow blocks and are called *fill-ins*. If we ignore these fill-ins at the cost of some approximation, then we can find L and U so that the entries of $A - LU$ are equal to zero in the positions of $NZ(A)$. The standard ILU(0) implementation is described in Algorithm 2.7. The selected nonzero pattern P is chosen to be equal to $NZ(A)$.

If we drop too many fill-ins, then the ILU(0) factorization will not be accurate enough to converge fast. By employing a hierarchy of ILU factorizations, we can keep more fill-ins and obtain more accurate and reliable preconditioners. This hierarchical ILU factorization is denoted by ILU(p), where the positive integer p represents the so-called *level of fill-in*.

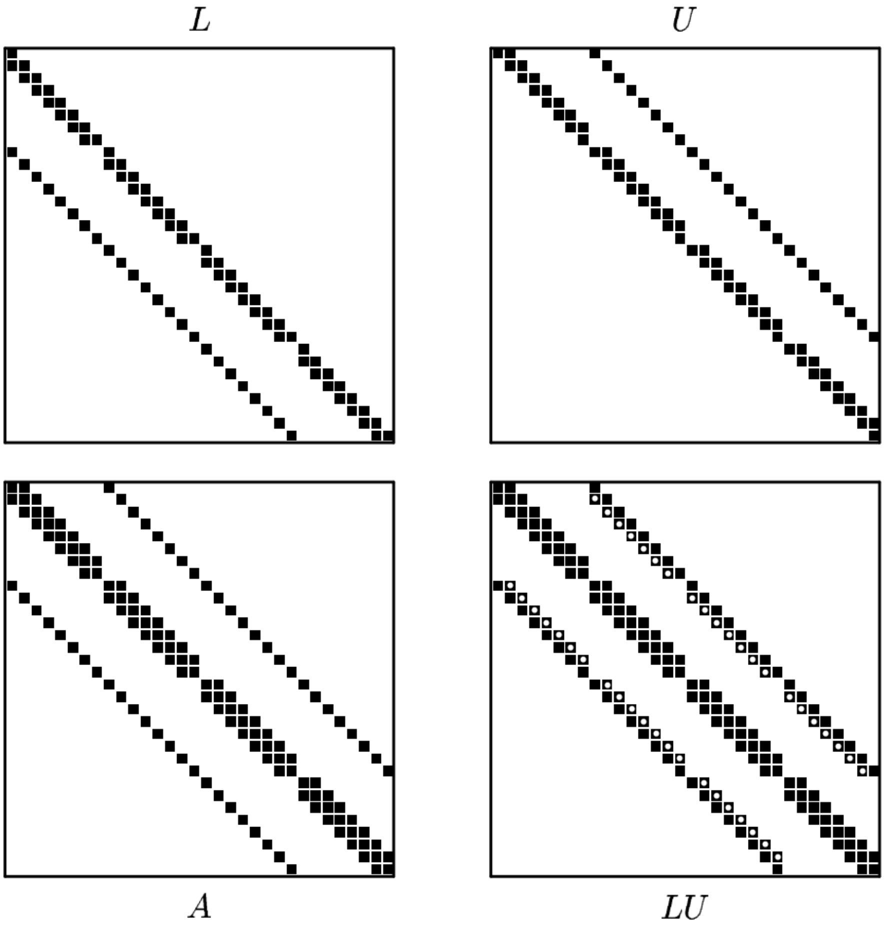


Figure 2.1: The ILU(0) factorization for a five-point matrix.

We continue to use the example of ILU(0) factorization shown in Figure 2.1 to elaborate ILU(1) factorization. Different from ILU(0), the ILU(1) factorization chooses $NZ(LU)$ to be the nonzero pattern of the triangular factors L_1 and U_1, with L and U resulting from the ILU(0) factorization applied to A. The pattern $NZ(LU)$ is shown at the bottom right part of Figure 2.1.

The nonzero patterns of L_1 and U_1 are illustrated at the top of Figure 2.2. The new product L_1U_1 is shown at the bottom right part of the figure, that has more

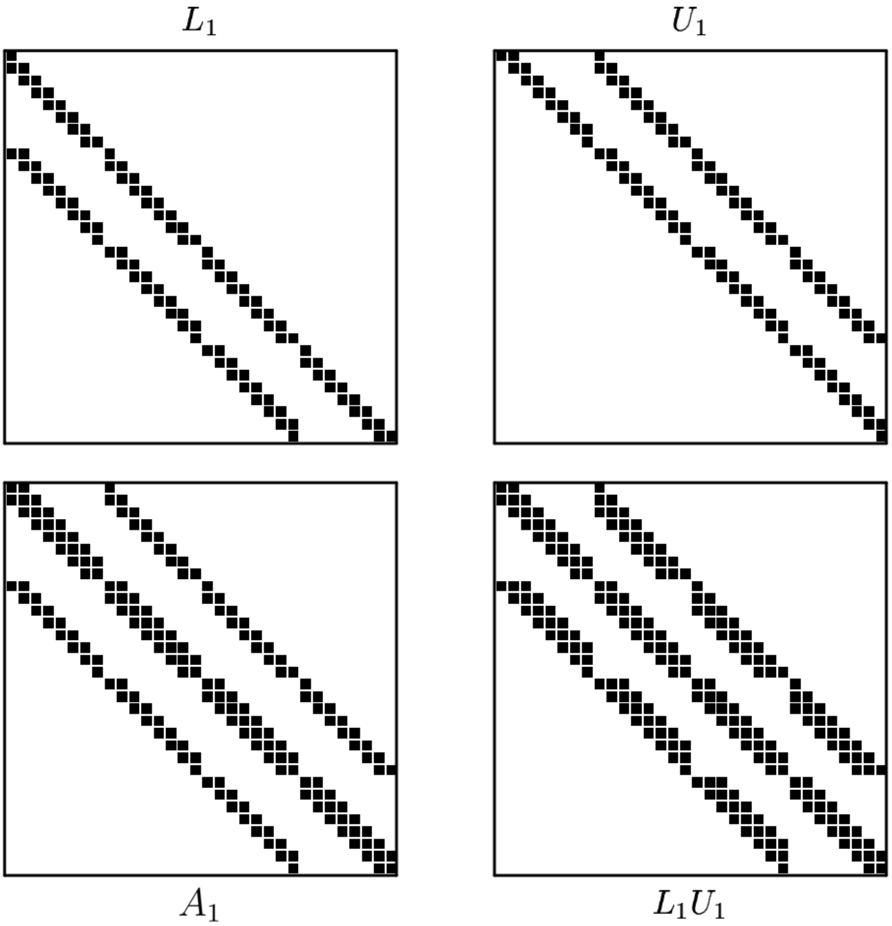


Figure 2.2: The ILU(1) factorization.

fill-ins than A_1.

Based on the concept of *levels of fill-in*, a hierarchy of ILU preconditioners can be obtained. A level of fill-in $lev_{i,j}$ is assigned to each entry a_{ij} that results from the Gaussian elimination process, and dropping is performed based on the values of $lev_{i,j}$. The initial level of fill-in of an entry a_{ij} is defined as follow:

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

During the ILU factorization, a_{ij} is modified at each step, and its level of fill-in $lev_{i,j}$ is updated accordingly:

$$lev_{i,j} = min\{lev_{i,j}, lev_{i,k} + lev_{k,j}\}. \tag{2.19}$$

In ILU(p), all fill-ins whose level of fill-in is greater than p are dropped. Then the nonzero pattern for ILU(p) is defined as

$$P_p = \{(i,j)|lev_{i,j} \leq p\}.$$

When $p = 0$, ILU(p) equals to the ILU(0) factorization as defined before. As p increases, accuracy grows while the factorization cost also increases. In general, ILU(1) is good enough for most cases. The accuracy of ILU(1) improves to a large extent compared to ILU(0), at moderate computational and memory cost. Below is the ILU(p) algorithm.

Algorithm 2.8 *ILU(p).*

```
Define $lev(a_{ij}) = 0$ for all the nonzero entries $a_{ij}$
**for** $i = 2 : n$ **do**
    **for** $k = 1 : i - 1$ and $lev_{i,k} \leq p$ **do**
        Compute $a_{ik} = a_{ik}/a_{kk}$
        $a_{i*} = a_{i*} - a_{ik}a_{k*}$
        Update $lev_{i,j}$ according to Equation (2.19)
    **end for**
**end for**
```

However, ILU(p) may drop large entries or keep small entries due to the static pattern selection strategy. This would lead to inaccurate factorization and result in slow convergence rate or even incorrect results. The problem can be partially remedied by using a magnitude-based dropping strategy, rather than a location-based dropping strategy. In this case the nonzero pattern P is determined dynamically.

By incorporating a set of rules for dropping small entries, a general ILU factorization with threshold (denoted as ILUT) can be obtained. In ILUT, the dropping strategy implemented in the Gaussian elimination process is based on their magnitude instead of their positions. Below is a short description of the ILUT algorithm. Here a_{i*} denotes the i-th row of A, and w_k denotes the k-th entry of the auxiliary vector w.

Algorithm 2.9 *ILUT.*

```
1:  for $i = 1 : n$ do
2:      $w = a_{i*}$
3:      for $k = 1 : i - 1$ and $w_k \neq 0$ do
4:          $w_k = w_k / a_{kk}$
5:          Apply a dropping rule to $w_k$
6:          if $w_k \neq 0$ then
7:              $w = w - w_k \times u_{k*}$
8:          end if
9:      end for
10:     Apply a dropping rule to row $w$
11:     $l_{i,j} = w_j$ for $j = 1, \ldots, i - 1$
12:     $u_{i,j} = w_j$ for $j = i, \ldots, n$
13:     $w = 0$
14: end for
```

1. In line 5, the element w_k is dropped if $w_k < \tau_i$, the relative tolerance obtained by multiplying τ by the original norm of a_{i*} .

2. In line 10, first drop the elements whose magnitude are below the relative tolerance τ_i in row w. Then except for the p largest elements in the L part, the p largest elements in the U part of the row and the diagonal element, all the rest entries are dropped.

Here p can be used to reduce the memory costs, and τ is used to control the computational costs. The preconditioner implemented in Algorithm 2.9 is referred to in the literature as ILUT(p, τ).

Multielimination ILU preconditioner

In the ILU preconditioners introduced above, the dropped entries are discarded according to the location or to the magnitude. The resulting factors are often ill-conditioned and the convergence rate tends to deteriorate [87]. To solve this issue, multilevel strategies are often incorporated in ILU factorizations. Multielimination ILU preconditioner (ILUM) for general sparse matrices is one of the pioneering methods in this category [84].

ILUM relies on the fact that at a given stage of Gaussian elimination, many rows can be eliminated simultaneously because of the matrix sparsity. The set that consists of such rows is called an *independent set*. Once an independent set is computed, the unknowns associated with it can be eliminated simultaneously. After the elimination, a smaller reduced system can be obtained. The elimination is applied approximately and recursively a few times, until the reduced system is small enough to be solved by an iterative solver.

We will use the following terminology. Let $G = (V, E)$ denote the adjacency graph of the matrix A. Here G is a directed graph, $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices, and E is the set of edges (v_i, v_j), where $v_i, v_j \in V$ and (v_i, v_j) denotes an edge from vertex v_i to v_j. The n vertices in V represent the n unknowns, and the edges in E represent the binary relations in the linear system. When $a_{ij} \neq 0$, there exists an edge $(v_i, v_j) \in E$.

Definition 1. *An independent set S is a subset of V such that*

$$\forall v_i, v_j \in S, (v_i, v_j) \notin E.$$

This corresponds to a set of nodes that are not coupled.

A maximal independent set is an independent set that can not be augmented, i.e., for any $v \in V$, $S \bigcup \{v\}$ is not an independent set. In this thesis the term independent set is used to refer to the maximal independent set.

During the multilevel Gaussian elimination process, the matrix A is preliminarily reordered into a 2 × 2 block form by a permutation matrix P

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}, \tag{2.20}$$

where D is diagonal. The permutation is executed so that the unknowns contained in the independent set are listed first, followed by the remaining unknowns. The following block LU factorization is performed on (2.20)

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ ED^{-1} & I \end{pmatrix} \times \begin{pmatrix} D & F \\ 0 & A_1 \end{pmatrix}.$$

Here $A_1 = C - ED^{-1}F$ is the Schur complement as well as the coefficient matrix of the reduced system. The same permutation can be applied recursively on A_1

and the consecutively reduced systems, until the last system is small enough to be easily solved by a standard method. After the last level is reached, the last-level system can be solved efficiently by a Krylov subspace or a direct method. Then through backward iterations, the solution for the original system is obtained. With the multilevel strategy, the solution of a big problem is decomposed into the solution of a sequence of smaller problems.

Block versions of the Multielimination and Multilevel ILU preconditioner

ILUM is an effective multilevel solver that has a high degree of parallelism, but it also has some drawbacks. First, during the reduction process, the diagonal entries of matrix D could be small in magnitude and this may lead to an inaccurate factorization. Second, when the coefficient matrix A is relatively dense, then the size of the independent set is typically small and the Schur complement is large, and the convergence rate deteriorates. In order to avoid these problems, the ILUM factorization can be generalized to block versions of multilevel ILU factorization (BILUM) [90] that exploit block structures in the matrix. The idea of exploiting the block structure within the coefficient matrix inspired us to construct the block multilevel solver, presented in Chapter 6.

Consider a partitioning of the set of vertices V of the adjacency graph of A with disjoint subsets V_i of vertices, that is

$$V_i \bigcap V_j = \emptyset, \text{ if } i \neq j.$$

A quotient graph can be constructed where each of these subsets is viewed as a super-vertex. In a quotient graph, vertex V_i is called to be adjacent to vertex V_j if there is a vertex in V_i which is sharing an edge with a vertex in V_j. Then a block independent set can be defined as follows.

Definition 2. *Let $V_1, V_2, \ldots, V_m$ be a group of disjoint subsets of V. The set $\bar{V}$ is called a block independent set if any two subsets V_i and V_j in $\bar{V}$ are not adjacent in the quotient graph.*

For simplicity, the block independent set ordering is initially described using a constant block size equal to 2. This can be generalized to any size. There are different blocking strategies to couple a node with its neighbors to form a 2×2 block. One way is to check the absolute values of the neighbors of this node, and couple the node with the neighbor which has the largest absolute value. This blocking strategy leads to nonsingular 2 × 2 diagonal blocks, and the inversion of these blocks is stable. Upon permutation, we obtain the reordered system

$$PAP^T = \begin{pmatrix} D & F \\ E & C \end{pmatrix}. \tag{2.21}$$

Equation (2.21) possesses the same structure as that of Equation (2.20), and the remaining factorization steps are also similar to ILUM. The submatrix D is a block diagonal matrix in BILUM framework, and the inverse of D can be explicitly computed by inverting the small blocks exactly. The forward and backward substitutions are similar to those in ILUM with the diagonal matrix D being a block diagonal matrix.

The Algebraic Recursive Multilevel Solver (ARMS)

The variant of the ILUM and BILUM methods that we consider in this thesis is called ARMS. The ARMS solver proposed in [89] is a general preconditioning method based on a multilevel partial solution approach. It can be viewed as a generalization of ILUM and BILUM. The main framework of ARMS is similar to that of BILUM. For the processing phase, the block size can be larger than two, and the block ILU factorization is applied recursively to the reduced system at each level, until the final system is small enough to be solved with a standard method; the solving phase consists of the solution of the last-level system and consecutive backward substitutions.

The ARMS solver uses block independent sets as in BILUM. One major implementation difference between the two methods lies in the calculation of the factorization and of the Schur complement at each level. One standard approach for computing the factorization (2.22) is to use Gaussian elimination and produce the factorization in the form

$$\begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} L & 0 \\ G & I \end{pmatrix} \times \begin{pmatrix} U & W \\ 0 & A_1 \end{pmatrix}. \tag{2.22}$$

Here D is factorized as $D \approx \bar{L}\bar{U}$ with $\bar{L} \approx L$ and $\bar{U} \approx U$, $G = E\bar{U}^{-1}$ and $W = \bar{L}^{-1}F$, and $A_1 = C - ED^{-1}F$ is the Schur complement with respect to C. However, in the ARMS implementation, after the incomplete factors of D, $\bar{L}$ and $\bar{U}$, are computed, the approximation $\bar{W} \approx \bar{L}^{-1}F$ is also computed. Then the approximation $\bar{G} \approx E\bar{U}^{-1}$ and approximate Schur complement $\bar{A}_1$ are obtained. These computations are iterated until the last level is reached. The blocks $\bar{W}$ and

$\bar{G}$ are stored temporarily to compute the Schur complement, and then discarded afterwards. Then the costs of $\bar{W}$ and $\bar{G}$ can be saved, and the accuracy of the factorization of D is guaranteed.

Inverse-based Multilevel Incomplete LU factorization

This recently developed preconditioner is implemented in the popular ILUPACK (abbreviation for Incomplete LU factorization PACKage) software package developed by Bollhöfer and Saad [16] for solving large sparse linear systems. The ILUPACK software is mainly built on multilevel incomplete factorization methods combined with Krylov subspace methods. The multilevel approach of ILUPACK is summarized below.

- The coefficient matrix A is first scaled by diagonal matrices D_1 and D_2

  $$D_1 A D_2 = \bar{A},$$

  and then permuted by matrices P and P^T

  $$P\bar{A}P^T = \hat{A}.$$

  Scaling and permutation can be viewed as preprocessing steps before the factorization step to make the original system more amenable to the iterative solution. These techniques include methods for reducing the fill-in and improving the diagonal dominance, like the maximum weight matching [15], the multiple minimum degree ordering [2] and the nested dissection [50].

- The matrix $\hat{A}$ is approximately factorized as

  $$\hat{A} \approx LDU,$$

  with L and U being unit lower and unit upper triangular factors, and D being a diagonal matrix.

- Apply the above procedures to the reduced system with coefficient matrix $\bar{A} = S$, where $S = C - ED^{-1}F$ is the Schur complement. Recursively apply these procedures onto the Schur complement until S is small or dense enough to be efficiently factorized by level 3 BLAS operations.

- Compute the last-level system with Krylov subspace methods and obtain the solution to the original system by a hierarchy of backward substitutions.

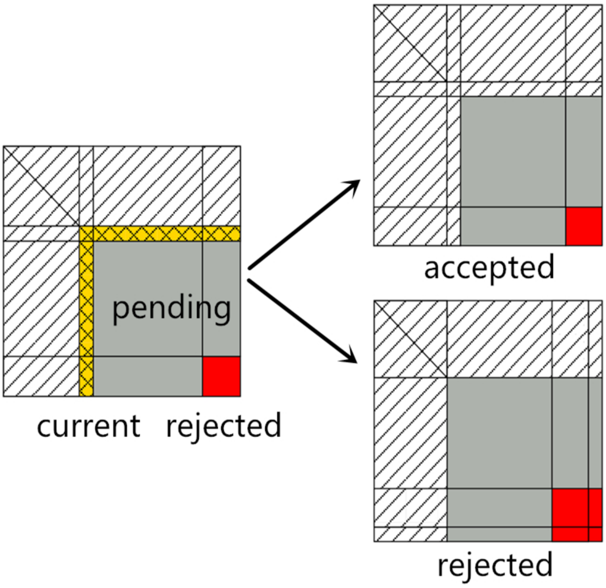


Figure 2.3: Pivoting in ILUPACK.

To limit the possibly large norm of the inverse triangular factors L^{-1} and U^{-1} which may lead to ill-conditioned triangular solvers, a pivoting strategy is incorporated during the factorization process. The pivoting can be expressed as

$$P\hat{A}P^T = \begin{pmatrix} B & E \\ F & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix} \begin{pmatrix} D_B U_B & D_B U_E \\ 0 & S \end{pmatrix} + R,$$

where R is the error matrix and S is the Schur complement. The pivoting strategy estimates the norm of the inverse factors. If the norm of one row in U^{-1} or one column in L^{-1} exceeds a pre-determined bound τ, then this row or column is rejected, otherwise it is accepted. All the accepted rows and columns continue the approximate factorization process. The rejected rows are permuted to the lower end of the matrix, and the rejected columns are permuted to the right end of the matrix. The pivoting process is illustrated in Figure 2.3.

The inverse-based approach of ILUPACK is based on subtle relations between ILU methods and sparse approximate inverse methods. Bollhöfer and Saad have investigated the relations between approximate inverses and related ILU methods [17]. These useful relationships can be applied to generate more robust variants of ILU methods [13]. In ILUPACK, the approximate inverse triangular factors are computed instead of the incomplete factors. The information contained in the inverse factors improves the robustness of the factorization process. Hence the inverse-based approach and the incomplete factorization process are combined in ILUPACK and efficiently implemented. We thoroughly use ILUPACK in our development and comparison.

2.2.3 Sparse-approximate-inverse-based preconditioners

Many popular general-purpose preconditioners, such as preconditioners based on incomplete factorizations of A, have good robustness and good convergence rate. However they require sparse triangular factors to be performed at each iteration, which are highly sequential operations. Hence it is difficult to obtain good parallel scalability using incomplete factorization preconditioners on parallel computers. Therefore in this thesis we also revisit the class of approximate inverse factorization preconditioners, as their constructions reduce to solving independent linear systems which can be performed concurrently. Before introducing our methods, we would like to recall the main underlying ideas.

Sparse approximate inverse preconditioning possesses natural parallelism since only sparse M-V products are needed at each step of an iterative method. This class of methods often succeeds in solving complicated problems while ILU preconditioning fails, and has attracted considerable research interest in the last few decades [1,7,9,33,46,53,64,86]. Sparse approximate inverse preconditioning aims at explicitly computing a sparse approximation of the inverse $M \approx A^{-1}$, or of its triangular factors $M = M_1M_2 \approx A^{-1}$. In the former case, it can be implemented as left-preconditioning (2.16) or right-preconditioning (2.17), in the latter case as split preconditioning (2.18). Then a Krylov subspace method is used to solve the preconditioned system.

A good preconditioner M should be as sparse as possible, and the construction of M should be efficient. Therefore, sparse approximate inverse preconditioning is based on the implicit assumption that A^{-1} can be effectively approximated by a sparse matrix, which means that most entries of A^{-1} are small. To save computational and memory costs, dropping strategies are applied during the preconditioning construction. Some dropping strategies are location-based or pattern-based, that is entries at chosen locations will be dropped; some dropping strategies are size-based or threshold-based, that is the entries whose size meet a certain condition will be dropped. Although the relation between the size of the dropped entries and the convergence rate is still not clearly understood, basically dropping small elements is a better choice, which tends to produce a better quality preconditioner [86].

To reduce the total execution time, the computation of M and the M-V product My should be computed in parallel. Sparse approximate inverse methods are clearly superior to Incomplete LU factorizations in this respect, as they do not require sequential triangular solvers. A lot of work has been devoted to developing preconditioning methods that naturally possess parallelism. Among these methods, sparse approximate inverse methods have received an increasing attention. We have incorporated some of the underlying ideas into our multilevel framework presented in the coming chapters, attempting to improve the performance of the original methods. Below we briefly introduce the popular classes of SParse Approximate Inverse (SPAI), Approximate INVerse (AINV) and Factorized Sparse Approximate Inverse (FSAI).

SParse Approximate Inverse (SPAI)

One idea for constructing an approximate inverse preconditioner is to find a sparse matrix M such that $||AM - I||$ is small for some selected norm. The SPAI method [53] presents the inverse in factored form, which avoids the singularity of M. Direct or iterative methods can be both used to minimize $||AM - I||_F$ [1].

The SPAI method uses the Frobenius norm to minimize $||AM - I||$, that is it solves

$$||AM - I||_F^2 = \sum_{k=1}^{n} ||(AM - I)e_k||_2^2. \tag{2.23}$$

Since each column of M is independent, solving Equation (2.23) amounts to solving n independent least squares problems

$$\min_{m_k} ||Am_k - e_k||_2, \; k = 1, 2, \ldots, n, \tag{2.24}$$

where m_k is the i-th column of M, and $e_k = (0, \ldots, 0, 1, 0, \ldots, 0)^T$. Below we solve the kth least squares problem as an example. Let $\mathcal{J}$ be the set of indices j such that $m_k(j) \neq 0$. $\overline{m_k}$ denotes the reduced vector of unknowns $m_k(\mathcal{J})$. To eliminate the zero rows in the submatrix $A(., \mathcal{J})$, let $\mathcal{I}$ be the set of indices i such that $A(i, \mathcal{J})$ is not all zero, and the resulting submatrix $A(\mathcal{I}, \mathcal{J})$ is denoted by $\overline{A}$. Analogously, $e_k(\mathcal{I})$ is denoted by $\overline{e_k}$. Then solving Equation (2.24) for m_k

amounts to solving the extracted least squares problem

$$\min_{\overline{m_k}} \|\overline{A}\overline{m_k} - \overline{e_k}\|_2. \tag{2.25}$$

When A and M are sparse matrices, the least squares problem (2.25) is much smaller than the corresponding original least squares problem in (2.24) and it can be easily solved. The computation is very similar in the case of left preconditioning.

On the one hand, preconditioning methods based on Frobenius norm minimization have high potential for parallelism. The minimization in the Frobenius norm leads to the independent calculation of each column of M, and Equation (2.24) can be solved in parallel. On the other hand, the solution process can be much simplified by solving the reduced least squares problems. Research reveals that the SPAI algorithm could produce a sparse and effective preconditioner [30].

The choice of the sparsity pattern of M is usually an important issue to compute an effective and cheap sparse approximate inverse preconditioner. A very sparse pattern yields a preconditioner that is easy to construct, but may lead to a slow convergence rate. A large sparsity pattern may enable us to converge faster, but it also results in higher computational costs. The idea is to keep M reasonably sparse while still being able to capture the large entries of the exact inverse of A, which are supposed to contribute most to the quality of the preconditioner computed. The sparsity pattern of M can be *static* or prescribed a *priori*. A common choice is to select for M the same nonzero pattern of A. This strategy can be effective especially if A has some degree of diagonal dominance. However, this approach may not be robust for solving general sparse problems since A^{-1} may have large entries outside of the sparsity pattern of A. One possible remedy is to choose the sparsity pattern of M to be that of A^k where k is a positive integer and $k \geq 2$ [33]. This approach computes more of the large nonzero entries and often results in better performance, but the computational and storage costs of the preconditioner tend to grow rapidly with larger k. The sparsity pattern of M can also be selected dynamically, starting with a simple initial guess like a diagonal pattern. Then the sparsity pattern is augmented as the algorithm proceeds, until some accuracy criteria are satisfied.

Approximate INVerse (AINV)

The AINV method proposed in [10, 11] is a method for computing sparse approximate inverse preconditioners in factorized form. The resulting factorized sparse approximate inverse is used as an explicit preconditioner for Krylov subspace methods.

AINV computes a split factorization $M = M_1M_2$ of the form (2.18). The triangular factors in the AINV algorithm are computed based on two sets of A-biconjugate vectors $\{z_i\}_{i=1}^n$ and $\{w_i\}_{i=1}^n$, i.e. $w_i^TAz_j = 0$ if and only if $i \neq j$. Then introducing the matrices $Z = [z_1, z_2, \ldots, z_n]$, and $W = [w_1, w_2, \ldots, w_n]$, the relation

$$W^TAZ = D = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix}$$

holds, where $p_i = w_i^TAz_i \neq 0$, and the inverse is equal to

$$A^{-1} = ZD^{-1}W^T = \sum_{i=1}^{n} \frac{z_iw_i^T}{p_i}.$$

Observe that A^{-1} is known if two complete sets of A-biconjugate vectors are known. The two sets of A-biconjugate vectors are computed by means of a (two-sided) Gram-Schmidt orthogonalization process with respect to the bilinear form associated with A. In exact arithmetic this process can be completed without breakdowns if and only if A admits an LU factorization, that is if all the leading principal minors of A are nonzeros [11]. Sparsity is preserved during the process by discarding elements having magnitude smaller than a given positive threshold. This strategy has proved to be robust and effective especially for unstructured matrices.

For SPD problems, there exists a stabilized variant of AINV which is breakdown free [7,62]. When $A_{n\times n}$ is an SPD matrix, it does not need to be stored explicitly. This is economical memory-wise cost and useful when A is just known as an implicit operator. After Z and D are obtained, the solution vector can be calculated as

$$x = A^{-1}b = ZD^{-1}Z^Tb = \sum_{i=1}^{n}(\frac{z_i^Tb}{p_i})z_i,$$

where $p_i = z_i^TAz_i$. For general problems, some care must be taken to avoid breakdowns due to divisions by zero.

During the AINV process, Z and W tend to be dense, which makes the costs unacceptable. Since many of the entries in the inverse factors of a sparse matrix are small in absolute value, sparsity can be preserved by dropping these small fill-ins arising during the computation of the vectors of Z and W.

The AINV method does not require the sparsity pattern to be known in advance. It uses adaptive techniques that dynamically identify the best structure for the inverse factors.

The AINV preconditioner is typically robust in applications and it results in good convergence rates of a Krylov subspace method at low set-up costs [11]. One drawback of AINV is that the orthogonalization steps for generating the preconditioner is highly sequential. This makes it difficult to implement AINV in parallel. Graph techniques can be used to obtain an effective parallelization [9].

Factorized Sparse Approximate Inverse (FSAI)

In this subsection we briefly review the FSAI preconditioner proposed by Kolotilina and Yeremin in a series of papers [63–65, 103].

Let

$$Ax = b \tag{2.26}$$

be an $n \times n$ general linear system with nonsingular, possibly indefinite and unsymmetric matrix $A = \{a_{ij}\} \in \mathbb{R}^{n \times n}$, and vectors $x, b \in \mathbb{R}^n$. Consider the left preconditioned system of the form

$$MAx = Mb,$$

where $M \approx A^{-1}$ is nonsingular. If the sparsity pattern $S \subseteq \{(i,j) : 1 \leq i \neq j \leq n\}$ for $M = \{m_{ij}\}$ is prescribed in advance, we can set

$$m_{ij} = 0 \text{ if } (i,j) \in S,$$

and determine the nonzero values m_{ij} for $(i,j) \notin S$ by minimizing the nonnegative quadratic functional

$$\|I - MA\|_W^2 = tr\left[(I - MA)\, W \,(I - MA)^T\right] \tag{2.27}$$

with a positive definite, possibly nonsymmetric weight matrix W [64]. A nonsymmetric matrix A is said to be positive definite if $(A + A^T)/2$ is positive definite, and hence we have

$$\left(MA\left(W + W^T\right)A^T\right)_{ij} = \left(\left(W + W^T\right)A^T\right)_{ij}, (i,j) \notin S. \tag{2.28}$$

Then Equation (2.28) decouples into n independent linear systems with symmetric positive definite coefficient matrices. We can compute nonzero entries in each row of M from corresponding independent linear system. Unfortunately, (2.28) for $W \neq A^{-1}$ generally yields an unsymmetric approximate inverse M. Hence we assume that A admits for a triangular decomposition

$$A = LU$$

and we precondition system (2.26) as

$$M_L A M_U y = M_L b$$

with $M_L \approx L^{-1}$ and $M_U \approx U^{-1}$, clearly preserving symmetry and/or positive definiteness of A.

The FSAI preconditioner is based on incomplete inverse factorizations of A^{-1}, and constructs M_L and M_U by first selecting some triangular sparsity patterns S_L and S_U such that

$$\{(i,j) : i < j\} \subseteq S_L \subseteq \{(i,j) : i \neq j\} \tag{2.29}$$

and

$$\{(i,j) : i > j\} \subseteq S_U \subseteq \{(i,j) : i \neq j\}\,, \tag{2.30}$$

and then computing nonzero entries of M_L and M_U to satisfy the conditions

$$\begin{array}{ll} (M_L)_{ij} = 0, & (i,j) \in S_L, \\ (M_U)_{ij} = 0, & (i,j) \in S_U, \\ (M_L A)_{ij} = \delta_{ij}, & (i,j) \notin S_L, \\ (A M_U)_{ij} = \delta_{ij}, & (i,j) \notin S_U. \end{array} \tag{2.31}$$

The preconditioned matrix has the form

$$M_L A M_U D^{-1}, \tag{2.32}$$

where

$$D = diag\,(M_L A M_U)\,. \tag{2.33}$$

It is easy to see that the nonzero entries of the i-th row of M_L can be determined by solving linear algebraic equations with coefficient matrix $A^{(i)}$, where $A^{(i)}$ denotes the principal submatrix of A having entries a_{jk} with $j,k \in \{\ell : (i,\ell) \notin S_L\}$ [64]. A similar assertion also holds for the nonzero entries of the jth column of the matrix M_U. The entries of the matrices M_L and M_U are uniquely determined whenever all the submatrices $A^{(i)}$, $i = 1, \ldots, n$, are nonsingular. From Equation (2.31), we see that the factors L and U are actually not needed for the construction of M_L and M_U. Clearly, there is wide scope for parallelism in this approach. For symmetric positive definite (SPD) linear systems, the resulting preconditioner is *quasi-optimal* as a minimizer of the Frobenius norm of the corresponding residual matrix over all lower triangular preconditioning matrices of a fixed sparsity pattern [103]. In addition, it is always well defined and can be computed in a stable way if A is SPD. Convergence results are proved for M-matrices, H-matrices, and block H-matrices [64]. For more general matrices, however, it may fail to exist.

The main issue is the selection of the nonzero patterns S_L and S_U. The idea is to keep the inverse factors M_L and M_U reasonably sparse while trying to capture the "large" entries of L^{-1} and U^{-1}, which are expected to contribute the most to the quality of the preconditioner. For arbitrary matrices, especially those having highly irregular structure, it is unclear how to determine a nearly optimal nonzero pattern in advance for the inverse factors. The simple strategy to take S_L equal to the sparsity pattern of the lower triangular part of the symmetrized matrix $A + A^T$, and to set $S_U = S_L^T$, can sometimes produce preconditioners of low quality (this will also be confirmed by some of our numerical experiments reported in this thesis). In order to improve the quality of the FSAI preconditioner, one could use the pattern of the lower triangular part of the matrix $(A+A^T)^p$, where p is a positive integer [33, 34, 102]. The larger p, the better in general the approximations to the triangular factors of A, and the higher the quality of the computed preconditioner. But larger p may rapidly increase the density of the pattern. The serial costs of constructing, storing and applying the FSAI method can be very high if its nonzero pattern is sufficiently dense.

Recently, blocking and adaptive variants have been proposed [47, 58, 60], which increase the robustness and efficiency as well as maintain the parallelism. The Block FSAI preconditioner (BFSAI) [58] combines the FSAI method with an incomplete block Jacobi algorithm to solve SPD linear systems of equations. The idea of BFSAI is to find a preconditioner $M = G^T G$, where $G \in S_L$, to minimize the Frobenius norm

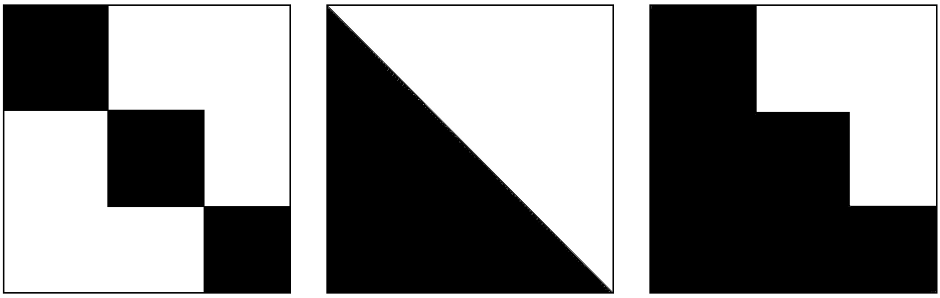
$$\|I - GL\|_F,$$

where L is the lower triangular factor of A such that $A = LL^T$, and S_L is the nonzero pattern of L. In BFSAI, the block preconditioner $M_A = F^T F$, where

$F \in S_{BL}$, is employed to minimize the Frobenius norm

$$\|D - FL\|_F,$$

where D is a block-diagonal matrix with a block nonzero pattern S_{BD}, and $S_{BL} = S_{BD} \cup S_L$. The block nonzero patterns are described in Figure 2.4. The inner preconditioner $M_A = F^T F$ is used to reduce the interaction between the diagonal blocks of A, which results in FAF^T's nice property of parallel implementation. To improve the convergence rate, FAF^T is preconditioned again. With a number of blocks at least equal to the number of available processors, each block of FAF^T is preconditioned separately with a sequential preconditioner. The outer preconditioner $J^T J$ of FAF^T is computed by an Incomplete Cholesky (IC) decomposition for improving the conditioning index of each block. Then the final BFSAI preconditioner is



(a) S_{BD} (b) S_L (c) S_{BL}

Figure 2.4: Schematic of the nonzero patterns S_{BD}, S_L and S_{BL}.

$$M = W^T W = (JF)^T JF,$$

and the preconditioned matrix is

$$JFA(JF)^T = WAW^T.$$

The BFSAI preconditioner is a parallelizable hybrid of FSAI and ILU, which coincides with FSAI when the number of blocks is equal to the dimension of the matrix, and reduces to IC if only one block is applied.

As an approximate inverse method based on the Frobenius norm minimization, the BFSAI preconditioner aims to compute a preconditioner to minimize the distance between the preconditioned matrix and a block diagonal matrix with Frobenius norm. Among the minimization process, the nonzero pattern is chosen beforehand by the user, which is usually a difficulty for a general problem. To overcome the limitation of BFSAI, the Adaptive BFSAI (ABF) preconditioner proposed in [60] generates the nonzero pattern automatically. The adaptive procedure captures the most significant nonzero entries of the first powers of A, which is based on the minimization of an upper bound to the Kaporin conditioning number of the preconditioned matrix. The Kaporin number of an SPD matrix A is defined as

$$\beta(A) = \frac{trA}{n\ \det(A)^{1/n}}.$$

The adaptive computing of the nonzero pattern proceeds iteratively until a user-specified criterion on the upper bound variation is satisfied. The adaptive strategy for computing the nonzero pattern captures the most significant entries of powers of A larger than 3 without computing all their entries, which reduces the number of iterations dramatically and keeps the density of the preconditioner at a low level.

In [47], the BFSAI preconditioner for SPD is further expanded to solving unsymmetric linear systems of equations, which is called Unsymmetric Block FSAI (UBF). In the UBF algorithm, the preconditioner for the unsymmetric matrix

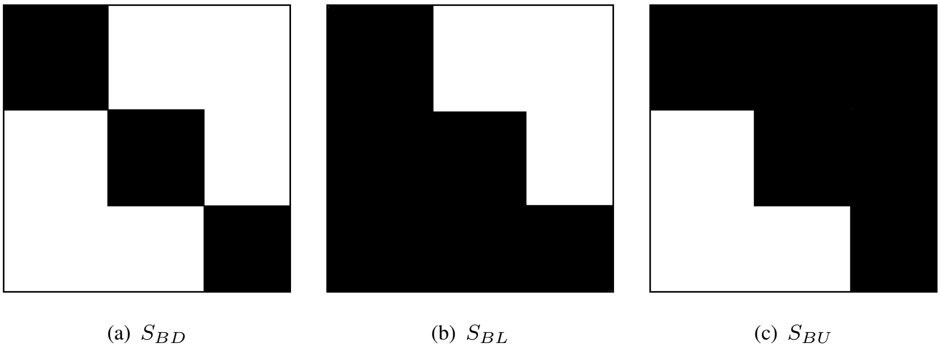

(a) S_{BD} (b) S_{BL} (c) S_{BU}

Figure 2.5: Schematic of the nonzero patterns S_{BD}, S_{BL} and S_{BU}.

A is computed as $M = F_U F_L$, where the block preconditioner $F_U \in S_{BU}$ and $F_L \in S_{BL}$. Here the factors F_L and F_U can be computed independently by splitting to different processors in parallel. Similar to block FSAI, the aim of F_L and F_U is to resemble the preconditioned matrix $F_L A F_U$ to a block diagonal matrix. Then the block diagonal matrix $F_L A F_U$ can be preconditioned by a block diagonal preconditioner in parallel, where each diagonal block is preconditioned separately by a sequential preconditioner.

2.3 Concluding remarks

In this chapter, we recalled the basic concepts underlying the development of modern Krylov subspace methods for solving large linear systems. By repeatedly performing M-V products involving the coefficient matrix A plus vector operations, Krylov subspace methods can significantly reduce the memory bottlenecks of direct solvers. Krylov subspace methods are iterative in nature and economical in computation. Therefore they are often considered the methods of choice for solving large linear systems such as those arising from the discretization of partial differential equations on modern computers. We have briefly highlighted several standard Krylov subspace methods including BiCG, CGS and GMRES. We have also proposed a comparison of the computational and storage costs of these methods. The GMRES method is mostly used in our algorithms, and some other Krylov methods are also applicable.

One practical problem is that iterative solvers lack the typical robustness of direct solvers. Preconditioning techniques are necessary for enhancing their robustness. We have described preconditioning techniques of both explicit and implicit form, with particular attention to Incomplete LU factorization (ILU) and sparse-approximate-inverse-based methods. In many problems, ILU preconditioners work well. But as the size of the problem increases, the convergence rate tends to deteriorate. Multilevel and block variants such as ILUM and BILUM improve standard methods and result in faster convergence rate. This motivates us to construct preconditioners with multilevel and block strategies. We have also discussed recent developments in the field of preconditioning methods, including ARMS and ILUPACK. These state-of-the-art methods have been well proved to have good performance on solving general linear systems. Hence we also incorporate ILUPACK in our new preconditioners, and compare the combinatorial methods to ARMS and ILUPACK. Preconditioners based on sparse approximate inverse are computationally attractive due to their natural parallelization and good numerical stability properties. We have briefly revisited the SPAI, AINV and FSAI methods, including recent block variants. The performance of these methods can also be improved by combining them with our multilevel strategy. In the coming chapters we present the development of a new class of linear solvers to enhance the robustness of multilevel preconditioning. We focused in particular on approximate inverse preconditioners, as their constructions reduce to solving independent linear systems, and this task can be performed concurrently. The proposed methods are based on multilevel and supernodal factorization for solving general linear systems, making them very good candidates to solve very large linear systems on massively parallel computers.

3 Distributed Schur complement preconditioning[1]

To solve the increasingly complicated and enormous linear systems from various applications, some aspects of the existing preconditioning methods need to be improved. Some of the existing methods are problem-dependent. There exists no such a universal preconditioning technique that is applicable to every problem. Although it is unlikely to compute a preconditioner that is the most effective for all applications, it is still highly desirable to develop general-purpose techniques that can be efficient for a large group of problems. Some of the existing preconditioning methods can be improved with respect to inherent parallelism, robustness, efficiency and cost. The design of the preconditioner is somehow a matter of both art and science. The matrix factorization involved in our preconditioners is based on the reordered matrices themselves and does not require any priori information of the problem or physical domain. Our goal is to propose black box solvers for sparse matrices with a general structure. The new preconditioning strategies and ideas have a good deal of generality, so that they can improve parallelism and robustness of iterative solvers, while being economic to implement and use.

In this chapter we introduce an algebraic recursive multilevel incomplete factorization preconditioner based on a distributed Schur complement formulation for solving general linear systems

$$Ax = b. \tag{3.1}$$

The novelty of the proposed method is to combine hybrid factorization techniques of both implicit and explicit type, and to use recursive combinatorial algorithms and multilevel mechanisms as an attempt to maximize sparsity and reduce costs in the factorization. Assuming that the coefficient matrix A of system (3.1) admits the factorization $A = LU$, with L a unit lower and U an upper triangular matrix, our

[1] This chapter is based on the published article [22, 24].

method approximates the inverse factors L^{-1} and U^{-1}. The strategies investigated in this chapter can be also incorporated in other existing preconditioning and iterative solvers packages. Our experiments will show that the proposed methods can compete well and can sometimes outperform other existing state-of-the-art approaches for solving linear systems. As it will be shown in the following chapters, some of the ideas proposed here can be used also in the context of direct methods.

This chapter is organized as follows. First we describe the proposed Algebraic Multilevel Explicit Solver (referred to as AMES) in Section 3.1. In Section 3.2 we assess the overall performance of AMES by showing several numerical experiments on realistic matrix problems. We compare the effect of AMES against other state-of-the-art solvers like ARMS and ILUPACK.

3.1 AMES: a hybrid recursive multilevel incomplete factorization preconditioner for general linear systems

In this section we describe AMES, an algebraic multilevel solver of explicit type that can be seen as a multilevel generalization of the FSAI factorization described in Section 2.2.3. The main novelty of the AMES solver is to use multilevel combinatorial algorithms to enforce sparsity in the approximate inverse factors. It is easier to describe the AMES method by using graph notation, dividing the solution of the linear system $Ax = b$ in five distinct phases [22, 24, 42]:

1. a *scale phase*, where the coefficient matrix A is scaled by rows and columns so that the largest entry of the scaled matrix has magnitude smaller than one;

2. a *preorder phase*, where the structure of A is used to compute a suitable ordering that maximizes sparsity in the approximate inverse factors;

3. an *analysis phase*, where the sparsity preserving ordering is analyzed and an efficient data structure is generated for the factorization;

4. a *factorization phase*, where the nonzero entries of the preconditioner are computed;

5. a *solve phase*, where all the data structures are accessed for solving the linear system.

Below we describe each phase separately.

3.1.1 Scale phase

We initially scale the system of equation $Ax = b$ by rows and columns as

$$D_1^{1/2}Ay = D_1^{1/2}b, \ \ y = D_2^{1/2}x, \tag{3.2}$$

where the $n \times n$ diagonal scaling matrices D_1 and D_2 have the form

$$D_1(i,j) = \begin{cases} \frac{1}{\max\limits_i |a_{ij}|} & , \text{ if } i = j \\ \\ 0 & , \text{ if } i \neq j \end{cases}, \quad D_2(i,j) = \begin{cases} \frac{1}{\max\limits_j |a_{ij}|} & , \text{ if } i = j \\ \\ 0 & , \text{ if } i \neq j \end{cases}.$$

For simplicity, in our thesis we still refer to the scaled system (3.2) as $Ax = b$.

3.1.2 Preorder phase

We use standard notation of graph theory to describe this computational step. We denote by $\mathcal{G}(\tilde{A})$ the undirected graph associated with the matrix

$$\tilde{A} = \begin{cases} A, & \text{if } A \text{ is symmetric,} \\ A + A^T, & \text{if } A \text{ is nonsymmetric.} \end{cases}$$

First, $\mathcal{G}(\tilde{A})$ is partitioned into p non-overlapping subgraphs $\mathcal{G}_i$ of roughly equal size by using the multilevel graph partitioning routines `metis` and `submetis`, which are available in the Metis package [61]. For each partition $\mathcal{G}_i$ we distinguish two disjoint sets of nodes (or vertices): *interior nodes* that are connected only to nodes in the same partition, and *interface nodes* that straddle between two different partitions; the set of interior nodes of $\mathcal{G}_i$ form a so called *separable* or *independent cluster* (see Figure 3.1).

Upon renumbering the vertices of $\mathcal{G}$ one cluster after another, followed by the interface nodes as last, and permuting A according to this new ordering, a block bordered linear system is obtained, with coefficient matrix of the form

$$\tilde{A} = P^TAP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} B_1 & O & O & O & F_1 \\ O & B_2 & O & O & F_2 \\ O & O & \ddots & O & \vdots \\ O & O & O & B_p & F_p \\ E_1 & E_2 & \cdots & E_p & C \end{pmatrix}. \tag{3.3}$$

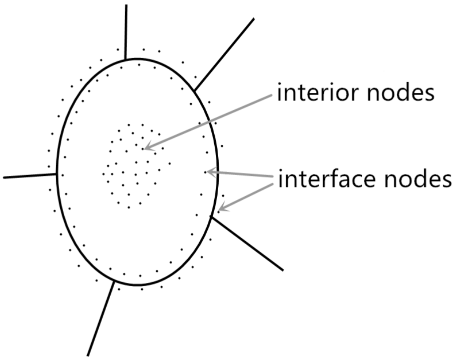
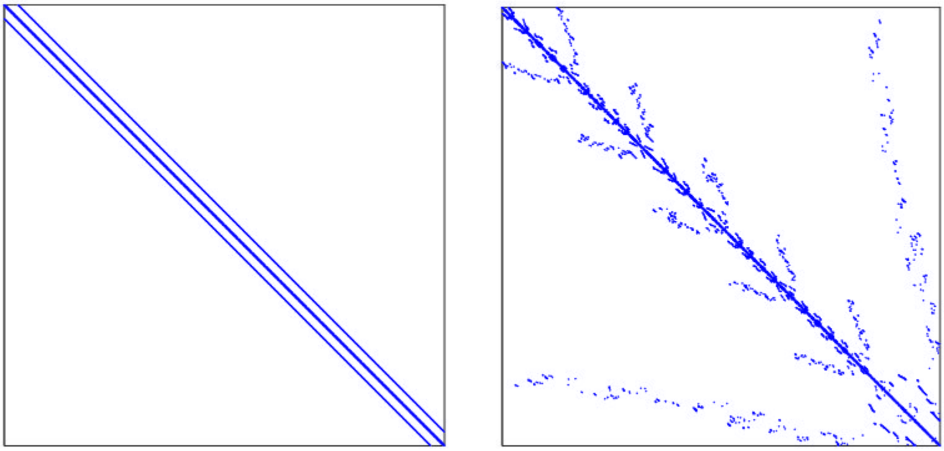


Figure 3.1: The classification of nodes in a local partition $\mathcal{G}_i$.

By abuse of notation, we still call $\mathcal{G}$ the renumbered graph. Here we use the multilevel graph partitioning routines `metis` and `submetis` in the Metis package [61] to carry out the nested dissection process. Nested dissection splits the problem into smaller subproblems, all having similar structures to the original one. Then the dissection process is repeated recursively on each of the subgraph until a desirable block-size is reached. By using the Metis tools, we can find block independent sets and split the graph (problem) into p separate subgraphs (subproblems) with no coupling. Then by combining the solutions to the subproblems, we obtain the solution to the original problem. Upon the nested dissection permutation, a sparse matrix can be reordered in the block downward arrow structure illustrated in Equation (3.3). The diagonal blocks B_i correspond to the interior nodes of $\mathcal{G}_i$, the blocks E_i and F_i correspond to the interface nodes of $\mathcal{G}_i$, the block C is associated to the mutual interactions between the interface nodes, and the rest are zero blocks. In our multilevel scheme, we apply the same block downward arrow structure to the diagonal blocks B_i of $\tilde{A}$. The procedure is repeated recursively until a maximum number of levels is reached, or until the last-level blocks are sufficiently small to be easily factorized. As an example, in Figure 3.2(b) we show the structure of the sparse matrix `rdb2048` from the SuiteSparse (formerly the University of Florida sparse matrix collection) [37] after three reordering levels.

To reduce factorization costs, a similar permutation is applied to the Schur



(a) The original structure of the rdb2048 matrix. (b) The structure of rdb2048 after permutation.

Figure 3.2: Structure of the multilevel inverse-based factorization for the matrix rdb2048.

complement matrix $S = C - EB^{-1}F$ as follows

$$\tilde{S} = \begin{pmatrix} B_{S1} & & & & F_{S1} \\ & B_{S2} & & & F_{S2} \\ & & \ddots & & \vdots \\ & & & B_{Sp} & F_{Sp} \\ E_{S1} & E_{S2} & \cdots & E_{Sp} & C_S \end{pmatrix}. \tag{3.4}$$

Although the Schur complement tends to get fairly dense, it typically preserves a good deal of sparsity that can be exploited in the design of the preconditioner.

3.1.3 Analysis phase

In the analysis phase, a suitable data structure for storing the linear system is defined, allocated and initialized. We use a tree structure to store the block bordered form (3.3) of $\tilde{A}$. The root is the whole graph $\mathcal{G}$, and the leaves at each level are the independent clusters of each subgraph. Each node of the tree corresponds to one partition $\mathcal{G}_i$ of $\mathcal{G}(\tilde{A})$, or equivalently to one block B_i of matrix $\tilde{A}$. The information stored at each node are the entries of the off-diagonal blocks E and F of B_i's father, and those of the block C of B_i after its permutation, except at the last level of the tree where we store the entire block B_i.

In order to take advantage of the large proportion of zero entries, we employ sparse storage formats, such as the *Compressed Sparse Row* (CSR) and the *Compressed Sparse Column* (CSC) formats to store the matrix blocks. The CSR format is one of the most popular storage formats for storing sparse matrices. The CSR data structure stores sparse matrices using three arrays AA, JA and IA. Suppose the matrix has nnz nonzero entries and its dimension is n. AA contains the nonzero entries of the matrix row by row. JA contains the column indices of the entries stored in AA. The lengths of AA and and JA are both nnz. IA contains the pointers to the first entries of each row in AA and JA. The i-th element of IA is the position in AA and JA where the i-th row starts. The length of IA is $n+1$ and $IA(n+1) = nnz + 1$.

Take the following matrix for example.

$$A = \begin{pmatrix} 6 & 3 & 0 & 0 & 9 \\ 0 & 1 & 0 & 0 & 8 \\ 2 & 0 & 9 & 3 & 0 \\ 0 & 7 & 0 & 1 & 0 \\ 2 & 5 & 0 & 0 & 2 \end{pmatrix},$$

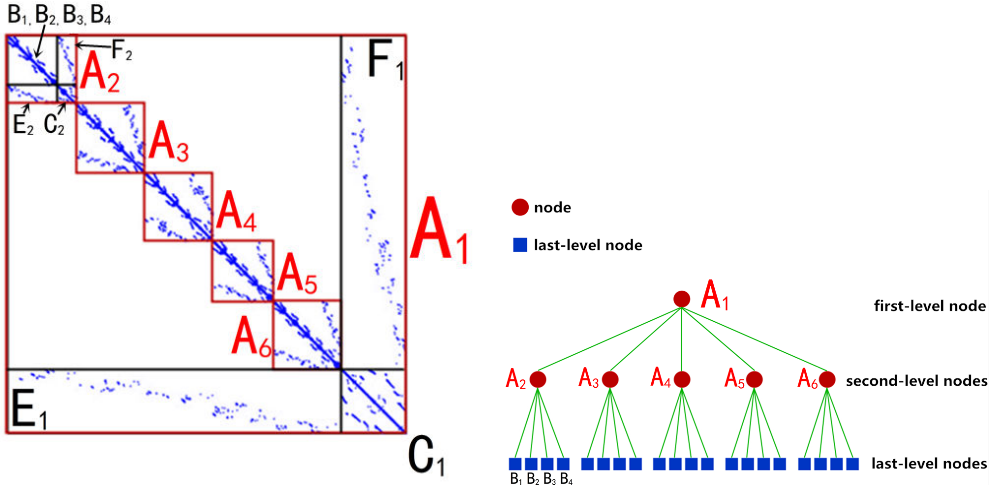
This matrix can be stored by the following CSR arrays

$$\begin{aligned} AA &= (6, 3, 9, 1, 8, 2, 9, 3, 7, 1, 2, 5, 2), \\ JA &= (1, 2, 5, 2, 5, 1, 3, 4, 2, 4, 1, 2, 5), \\ IA &= (1, 4, 6, 9, 11, 14). \end{aligned}$$

Similar to CSR, CSC stores the entries, indices and pointers of the nonzero entries of the sparse matrices in column-wise manner. The above matrix can be stored with CSC arrays as

$$\begin{aligned} AA &= (6, 2, 2, 3, 1, 7, 5, 9, 3, 1, 9, 8, 2), \\ JA &= (1, 3, 5, 1, 2, 4, 5, 3, 3, 4, 1, 2, 5), \\ IA &= (1, 4, 8, 9, 11, 14). \end{aligned}$$

In Figure 3.3(a), we describe the structure of matrix `rdb2048` after the nested dissection permutation, and in Figure 3.3(b) we show the tree data structure. In this example the matrix `rdb2048` is partitioned into three levels. The first-level



(a) The structure of `rdb2048`.

(b) The tree graph of `rdb2048`.

Figure 3.3: Structure and tree graph of matrix `rdb2048` after the nested dissection permutation.

block A_1 is partitioned into five second-level blocks A_2, A_3, A_4, A_5 and A_6. As an example of the second-level blocks, A_2 is partitioned into four blocks B_1, B_2, B_3 and B_4, which are last-level blocks. At the first level, we store blocks E_1, F_1 and C_1; at the second level, we store blocks $E_2 \sim E_6$, $F_2 \sim F_6$ and $C_2 \sim C_6$; at the third level (last level), we store all the blocks B. All the blocks E_i, C_i and the last-level blocks B_i are stored in CSR format, and blocks F_i are stored in CSC format.

3.1.4 Factorization phase

The approximate inverse factors $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ of $\tilde{A}$ can be written in the following form

$$\tilde{L}^{-1} \approx \begin{pmatrix} U_1^{-1} & & & & W_1 \\ & U_2^{-1} & & & W_2 \\ & & \ddots & & \vdots \\ & & & U_p^{-1} & W_p \\ & & & & U_S^{-1} \end{pmatrix}, \; \tilde{U}^{-1} \approx \begin{pmatrix} L_1^{-1} & & & & \\ & L_2^{-1} & & & \\ & & \ddots & & \\ & & & L_p^{-1} & \\ G_1 & G_2 & \cdots & G_p & L_S^{-1} \end{pmatrix} \tag{3.5}$$

where $B_i = L_i U_i$, and L_S, U_S are the triangular factors of the Schur complement matrix

$$S = C - \sum_{i=1}^{p} E_i B_i^{-1} F_i. \tag{3.6}$$

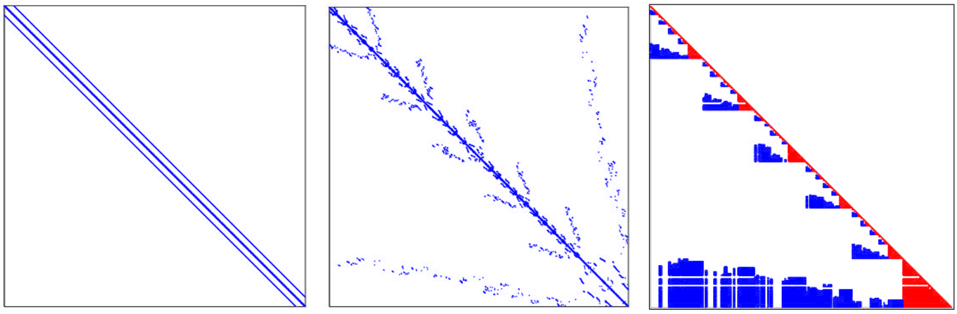
By computing the matrix-matrix production $A \times \tilde{A}^{-1} = I$ with Equation (3.5), that is

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \times (\tilde{L}^{-1} \times \tilde{U}^{-1}) = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix},$$

we can obtain the auxiliary matrices

$$W_i = -U_i^{-1} L_i^{-1} F_i U_S^{-1}, G_i = -L_S^{-1} E_i U_i^{-1} L_i^{-1} \tag{3.7}$$

Some fill-in may occur in $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ during the factorization, but only within the nonzero blocks. This two-level reordering scheme was used in the context of factorized approximate inverse methods for the parallelization of the AINV preconditioner in [9]. However, differently from [9], we apply the arrow structure (3.3) recursively to the diagonal blocks and to the first-level Schur complement as well, so that most of the nonzero entries of the original matrix are clustered into a few nonzero blocks. This effectively maximizes sparsity in the inverse factors and consequently reduces the factorization costs. The multilevel factorization algorithm requires to invert only the last-level blocks and the small Schur complements at each reordering level; the blocks W_i, G_i do not need to be assembled explicitly, as they may be applied using Equation (3.7). For the `rdb2048` problem, in Figure 3.4(c) we display in red the actual extra storage required by the exact multilevel inverse factorization in addition to matrix A; these represent only 34% of the total number of nonzeros of A which is already very sparse. From the knowledge of the red entries, the blue ones can be retrieved from Equation (3.7), using the off-diagonal blocks of A. We also permute the large Schur complement at the first level into a block bordered structure, until we reach a maximal number of levels or a given minimal size. The last-level matrix is inverted inexactly. An inexact solver is also used to factorize the last-level blocks B_i in (3.6).



(a) The original structure of `rdb2048`.

(b) The structure of `rdb2048` after permutation.

(c) The structure of the inverse factor. The entries actually stored are displayed in red.

Figure 3.4: Structure of the multilevel inverse-based factorization for the matrix `rdb2048`.

3.1.5 Solve phase

In the solve phase, the multilevel factorization is applied at every iteration step of a Krylov method for solving the linear system. Notice that the inverse factorization of $\tilde{A}$ may be written as

$$(PAP^T)^{-1} = \begin{pmatrix} U^{-1} & W \\ 0 & U_S^{-1} \end{pmatrix} \times \begin{pmatrix} L^{-1} & 0 \\ G & L_S^{-1} \end{pmatrix} \tag{3.8}$$

where $W = -U^{-1}L^{-1}FU_S^{-1}$, $G = -L_S^{-1}EU^{-1}L^{-1}$, and L_S, U_S are the inverse factors of the Schur complement matrix $S = C - EB^{-1}F$.

From Equation. (3.8), we obtain the following expression for the exact inverse

$$\begin{pmatrix} B^{-1} + B^{-1}FS^{-1}EB^{-1} & -B^{-1}FS^{-1} \\ -S^{-1}EB^{-1} & S^{-1} \end{pmatrix}. \tag{3.9}$$

We can derive preconditioners from Equation. (3.9) by computing approximate solvers $\tilde{B}^{-1}$ for B and $\tilde{S}^{-1}$ for S, that are cheap to compute, apply and store due to sparsity. Hence the preconditioner matrix M will have the form

$$M = \begin{pmatrix} \tilde{B}^{-1} + \tilde{B}^{-1}F\tilde{S}^{-1}E\tilde{B}^{-1} & -\tilde{B}^{-1}F\tilde{S}^{-1} \\ -\tilde{S}^{-1}E\tilde{B}^{-1} & \tilde{S}^{-1} \end{pmatrix}, \tag{3.10}$$

and the preconditioning operation $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = M \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$ writes as Algorithm 3.1. Notice that Algorithm 3.1 is called recursively at lines 1-3, as $\tilde{B}$ and $\tilde{S}$ also have a block bordered structure upon permutation.

Algorithm 3.1 *The preconditioning operation in the AMES solver.*

```
1: $p_1 = \tilde{B}^{-1}b_1$
2: $[p_2, p_3] = \tilde{S}^{-1}[E \cdot p_1, b_2]$
3: $[p_4, p_5] = \tilde{B}^{-1}[F \cdot p_2, F \cdot p_3]$
4: $x_1 = p_1 + p_4 - p_5$
5: $x_2 = p_3 - p_2$
```

3.2 Numerical experiments with AMES

In this section we present the results of our numerical experiments to illustrate the performance of the AMES preconditioner, also against other state-of-the-art methods and software for solving general linear systems. The selected matrix problems are extracted from the public-domain matrix repository available at the SuiteSparse [37], and arise from various application fields. We present a summary of the characteristics of each linear system in Table 3.1. We applied AMES as a preconditioner for the Generalized Minimal Residual (GMRES) method by Saad and Schultz [83]. In all our runs we started the iterative solution from the zero vector and we stopped it when either the initial residual was reduced by twelve orders of magnitude or when no convergence was achieved after 5000 M-V products. To limit memory costs, we restarted the GMRES algorithm every 500 iterations. The right-hand side b of the linear system was chosen so that the solution is the vector of all ones, that is $b = Ae$ with $e = (1, \ldots, 1)^T$. In each run we recorded the following performance measures:

Matrix problem	n	Field	$nnz(A)$
orsirr_1	1,030	Oil reservoir simulation	6,858
1138_bus	1,138	Bus Power System	4,054
bcsstk27	1,224	BCS Structural Engineering Matrix	28,675
epb0	1,794	Plate-fin heat exchanger	7,764
cz20468	20,468	Closest Point Method	206,076
raefsky3	21,200	Fluid Structure Interaction	1,488,768
ABACUS_shell_ud	23,412	ABAQUS benchmark	218,484
sme3Db	29,067	3D structural mechanics problem	2,081,063
viscoplastic2	32,769	FEM discretization	381,326
cz40948	40,948	Closest Point Method	412,148
rma10	46,835	3D CFD Model	2,374,001
finan512	74,752	Portfolio optim	596,992
helm2d03	392,257	Helmholtz eq. on a unit square	2,741,935
parabolic_fem	525,825	Parabolic FEM	3,674,625

Table 3.1: Set and characteristics of the test matrix problems.

1. the density ratio $\frac{nnz(M_L+M_U)}{nnz(A)}$, that is the ratio between the number of nonzeros in the preconditioner matrix $M = M_U M_L$ versus the number of nonzeros in the coefficient matrix A;

2. the number of iterations Its required to reduce the initial residual by 12 orders of magnitude starting from the zero vector;

3. the CPU time cost in seconds for completing the preorder phase (denoted by t_p), for constructing the approximate inverse factorization (t_f), and for solving the linear system (t_s). Symbol "-" means that the corresponding phase does not apply to the given run. For example, some of the preconditioners used for the comparison against our method do not have a preorder phase.

The codes were developed in Fortran 95. The experiments were run in double precision floating point arithmetic on a PC equipped with an Intel(R) Core(TM)2 Duo CPU E8400, 3.00 GHz of frequency, 4 GB of RAM and 6144 KB of cache memory. In the coming sections we study the effect of using different parameter settings in the AMES solver, and we illustrate the overall performance on the selected matrix problems.

3.2.1 Performance of the multilevel mechanism

The AMES method can be seen as a multilevel generalization of factorized approximate inverse techniques such as the FSAI preconditioner by Kolotilina and Yeremin, and the AINV preconditioner by Benzi and Tůma. Therefore, first we present some comparison between these methods, to show the benefit of the multilevel mechanism introduced in our AMES. The results are reported in Table 3.2. For these runs, we considered four matrix problems from Table 3.1, that are `orsirr_1`, `1138_bus`, `bcsstk27` and `epb0`. In our AMES solver, we inverted the last-level blocks using ILU, FSAI and AINV factorizations. For ILU, we used the multilevel implementation available in the ILUPACK package [16] (this combination is referred to as *AMES_ILU* in the table).

For FSAI, we use the structure of the nonzero pattern of the lower (resp. upper) triangular part of the symmetrized block for the approximate inverse factors, and also the square of this pattern (this combination is referred to as *AMES_FSAI*). For AINV we use the implementation kindly provided by the authors (this combination is referred to as *AMES_AINV*). *AMES_FSAI* and *AMES_AINV* are multilevel generalizations of AINV and FSAI respectively. This motivates the comparison on several problems in our numerical experiments. On the other hand, *AMES_ILU* is tested mostly for comparison purpose. The dropping threshold value selected for the *AINV*, *AMES_AINV* and *AMES_ILU* methods (referred to as *Drop* in the Table) is an absolute value, and is computed so that the resulting preconditioners have roughly equal memory cost. We use the default value for the parameter $condest = 10$ (norm bound for the inverse factors L^{-1} and U^{-1}) in ILUPACK.

In our runs, the multilevel variants *AMES_FSAI* and *AMES_AINV* perform consistently better than the *FSAI* and *AINV* solvers in terms of convergence rate and storage cost. This shows that the proposed multilevel mechanism enable us to exploit sparsity in the inverse factors more effectively. The best solutions with AMES are obtained using ILU as local solver, while the threshold-based dropping rules of the AINV method often compute a better pattern for the approximate inverse factors than the static approach used in the FSAI method. We can see evidence of this behaviour in Figures 3.5 - 3.8, where for one of the last-level blocks of the permuted coefficient matrix (3.3) we compare the structure of its exact inverse factor L^{-1}, and of the approximations M_L and W^T of L^{-1} as computed by, respectively, the *AMES_FSAI* code using the square of the pattern of the symmetrized block, and by the *AMES_AINV* code. Large to small entries are depicted in different colors, from red to orange and yellow. The approximation is good for the `1138_bus` problem (Figure 3.5) but poor for the `orsirr_1` matrix (Figure 3.6), and this is confirmed by the different convergence results for the two problems, reported in Table 3.2. On some larger problems, like the `cz40948` and the `ABACUS_shell_ud` problems, shown in Figures 3.7 - 3.8, we find that L^{-1} has no evident structure; in this case we have to increase the number of nonzeros in M_L and W^T significantly to converge. For example on the `ABACUS_shell_ud` problem, *AMES_AINV* converges in 468 iterations with $nnz(Z+W)/nnz(A) = 11.6$ while *AMES_FSAI* does not converge at this value of density. In these situations, uniformly better convergence is obtained using ILU as local solver. We will focus mostly on this choice of local solver for the experiments of this chapter. Notice that in this case the entries of the inverse factors are not computed explicitly, and the application of the preconditioner is carried out through a backward and forward substitution procedure. Other options may be considered for the last level solver, such as the ARMS method [89] and enhanced FSAI methods [59], but these are not included in the presented analysis.

(a) orsirr_1

Method	Pattern	Drop/condest	*Its*	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s
AMES_FSAI	$A+A^T$	-	273	1.42	0.011	0.023	0.22
FSAI			304	1.45	-	0.070	0.23
AMES_FSAI	$(A+A^T)^2$	-	217	3.43	0.013	0.035	0.17
FSAI			236	3.76	-	0.088	0.16
AMES_AINV	-	0.03	67	2.27	0.016	0.014	0.034
AINV		0.07	80	2.22	-	0.016	0.024
AMES_ILU	-	8e-3/10	31	1.24	0.012	0.013	7.4e-3

(b) `1138_bus`

Method	Pattern	Drop/condest	*Its*	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s
AMES_FSAI	$A+A^T$	-	7	2.24	5.2e-3	0.032	1.2e-3
FSAI			9	2.32	-	0.074	8.9e-4
AMES_FSAI	$(A+A^T)^2$	-	5	2.70	5.0e-3	0.035	1.0e-3
FSAI			6	2.88	-	0.077	6.4e-4
AMES_AINV	-	0.6	13	2.85	7.0e-3	2.0e-3	1.9e-3
AINV		0.7	16	2.88	-	6.0e-3	3.2e-3
AMES_ILU	-	0/10	1	1.00	5.1e-3	3.9e-3	7.0e-4

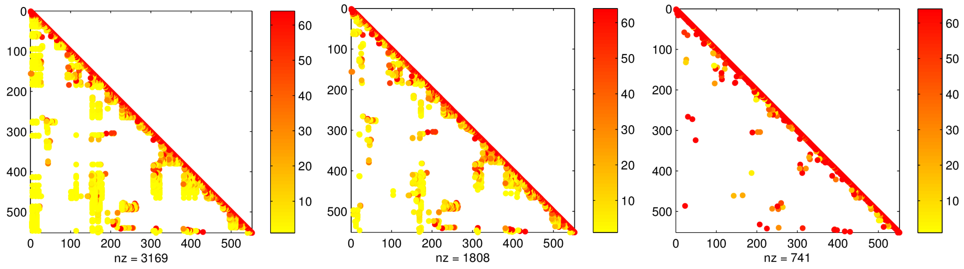
(c) bcsstk27

Method	Pattern	Drop/condest	*Its*	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s
AMES_FSAI	$A+A^T$	-	8	0.90	0.062	0.041	0.021
FSAI			19	1.27	-	0.20	4.1e-3
AMES_FSAI	$(A+A^T)^2$	-	5	1.16	0.063	0.071	0.018
FSAI			13	2.72	-	0.47	3.7e-3
AMES_AINV	-	1e-3	6	1.18	0.055	0.040	7.3e-3
AINV		0.06	16	0.98	-	0.063	5.7e-3
AMES_ILU	-	0.01/10	6	0.978	0.059	0.016	0.010

(d) epb0

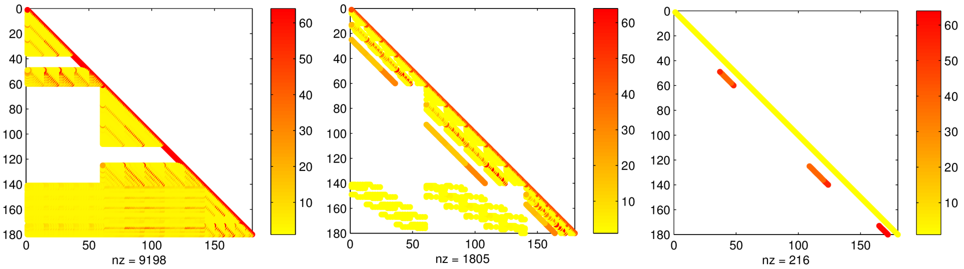
Method	Pattern	Drop/condest	*Its*	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s
AMES_FSAI	$A+A^T$	-	277	1.67	0.020	0.011	0.66
FSAI			400	1.69	-	0.19	0.59
AMES_FSAI	$(A+A^T)^2$	-	161	4.32	0.021	0.023	0.40
FSAI			290	4.81	-	0.23	0.27
AMES_AINV	-	0.5	132	3.32	0.024	4.5e-3	0.21
AINV		0.9	347	4.26	-	0.015	0.42
AMES_ILU	-	0.1/10	7	1.848	0.020	4.1e-3	0.019

Table 3.2: Numerical experiments on selected matrix problems illustrating the performance of the multilevel sparse approximate inverse preconditioner AMES against the factorized approximate inverse methods FSAI and AINV.



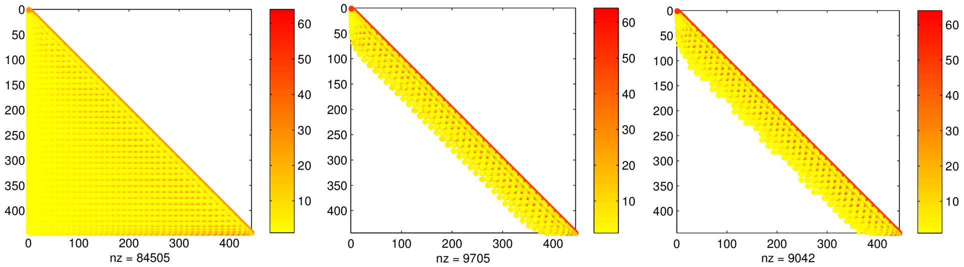
(a) L^{-1} (b) M_L (c) W^T

Figure 3.5: The exact (left) and approximate (middle and right) inverse lower triangular factors of the `1138_bus` matrix.



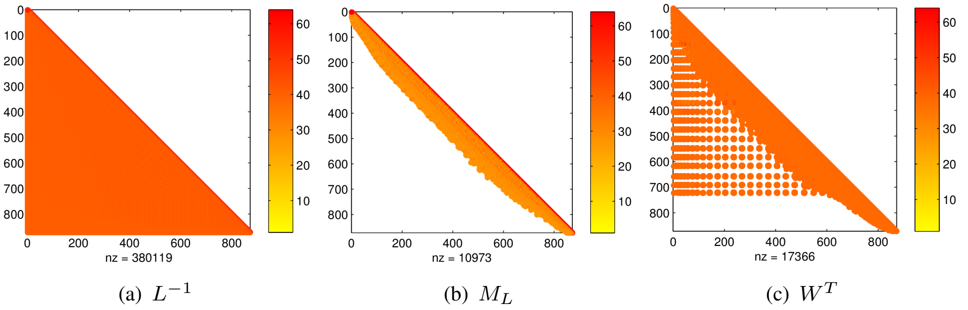
(a) L^{-1} (b) M_L (c) W^T

Figure 3.6: The exact (left) and approximate (middle and right) inverse lower triangular factors of the `orsirr_1` matrix.



(a) L^{-1} (b) M_L (c) W^T

Figure 3.7: The exact (left) and approximate (middle and right) inverse lower triangular factors of the `cz40948` matrix.



(a) L^{-1} (b) M_L (c) W^T

Figure 3.8: The exact (left) and approximate (middle and right) inverse lower triangular factors of the ABACUS_shell_ud matrix.

3.2.2 Varying the number of independent clusters at the first level

In the numerical tests, we observe that the pre-processing procedure is much cheaper than the factorization phase. To achieve the optimal parameter setting and take full advantage of the block bordered structure, we have conducted various numerical experiments with respect to different choices of parameters and present them in the following sections.

We considered three matrix problems in our runs `cz20468`, `ABACUS_shell_ud` and `cz40948`. In Table 3.3 we show the results varying the number of independent clusters p at the first level of reordering of A in (3.3). For each problem, we used the same number of levels n_{lev} in the AMES structure, and tuned the drop tolerance in the local ILU factorization to keep the memory ratio $\frac{nnz(M_L+M_U)}{nnz(A)}$ roughly constant while increasing p in different runs. Clearly, larger p results in more independent clusters of smaller size, and in larger Schur complement matrices. In the table we report the ratio $\frac{sizeB}{sizeA_S}$ between the average size of the independent clusters B_i and the size of the Schur complement at the first level. Increasing p reduces in turn $\frac{sizeB}{sizeA_S}$ to values smaller than 1. Using ILU as local solver, the best convergence results were obtained when $\frac{sizeB}{sizeA_S} \approx 1$. Our experiments indicate that for good performance the size of each independent cluster should be approximately equal to that of the Schur complement matrix.

Matrix	p	*Its*	t_p	t_f	t_s	t_{per_it}	$\frac{sizeB}{sizeA_S}$
cz20468	15	151	0.4	0.3	3.0	0.020	4.3
	20	139	0.4	0.3	2.8	0.020	2.5
	30	131	0.5	0.4	2.5	0.019	1.1
	40	137	0.5	0.6	2.6	0.019	0.6
ABACUS_shell_ud	4	258	0.5	1.2	9.1	0.035	7.8
	6	242	0.5	1.2	8.3	0.034	3.8
	12	213	0.5	2.0	6.8	0.032	1.0
	15	253	0.5	2.1	8.7	0.034	0.7
cz40948	15	219	1.1	0.5	10.8	0.049	8.7
	30	212	1.1	0.7	10.0	0.047	2.2
	45	198	1.1	1.2	9.2	0.046	0.9
	50	207	1.1	1.9	9.8	0.047	0.5

Table 3.3: The best performance of the multilevel sparse approximate inverse preconditioner are observed when $\frac{sizeB}{sizeA_S} \approx 1$.

3.2.3 Varying the number of reduction levels for the diagonal blocks

We consider again matrices cz40948, ABACUS_shell_ud and cz20468 for our tests. We varied the number of levels n_{lev} from 1 to 3 in the multilevel reordering of the diagonal blocks. In each run we tuned the dropping threshold parameter to have roughly the same memory cost in the experiments for each matrix. We chose the value of p for each problem so that $\frac{sizeB}{sizeA_S} \approx 1$ as reported in Section 3.2.2. We have fixed a maximum number of levels and a minimum block size, and incorporate this information in the criteria used to stop the multilevel reordering. We set them as parameters of the code, and changed them manually at every run if necessary. Then we selected values of these parameters for good performance. The last-level blocks were factorized using ILUPACK [16]. The results reported in Table 3.4 show that using more levels can reduce the number of iterations for similar memory ratio as we can gain additional sparsity during the factorization. However, probably due to our non optimized implementation, the solution cost tends to increase.

Since using many levels would intensively increase the solution cost, a small number of reduction levels is recommended to use. In this case, to remedy the

Matrix	n_{lev}	*Its*	t_p	t_f	t_s	t_{per_it}
	1	113	0.5	0.4	1.9	0.017
cz20468	2	80	0.5	0.5	2.3	0.028
	3	71	0.6	0.5	4.1	0.058
	1	388	0.5	1.7	17.6	0.045
ABACUS_shell_ud	2	381	0.5	1.9	21.9	0.057
	3	294	0.6	1.9	22.7	0.077
	1	198	1.1	1.2	9.2	0.046
cz40948	2	154	1.2	1.3	10.4	0.068
	3	133	1.3	1.3	17.3	0.13

Table 3.4: The number of iterations of the multilevel approximate inverse preconditioner can be reduced by increasing the number of reduction levels n_{lev} for the diagonal blocks, at roughly equal memory costs.

increasing solution cost, we could modify the strategy and use more independent clusters, that is increase p, at each level to make the blocks B smaller. Then within 2 or 3 levels, we could obtain small last-level blocks B. For this strategy, we need to conduct further research on the optimal choice of the parameters with the strategy illustrated in Section 3.2.2.

3.2.4 Varying the number of reduction levels for the Schur complement

The Schur complement matrix relative to the block C in (3.3) typically preserves a good deal of sparsity, and this can be further exploited during the factorization by applying, e.g., the multilevel nested dissection reordering to A_S, similarly to what is done to the upper leftmost block B. We implemented this idea at the first permutation level, using ILU factorization as local solver and selecting the same values of p and n_{lev} for each matrix problem. We tuned the drop tolerance in the ILU factorization to have roughly the same memory costs in different runs. We varied n_{levAS} from 0 to 3 ($n_{levAS} = 0$ means that only the diagonal blocks of the upper-left block B are permuted). The results reported in Table 3.5 show that the simultaneous permutation of both the diagonal blocks of B and of the Schur complement can make the preconditioner more robust. We adopted this strategy in the experiments illustrated in the coming sections, selecting in each run the value of n_{levAS} that minimized the total solution cost.

Matrix	n_{levAS}	*Its*	t_p	t_f	t_s
cz20468	0	331	0.5	0.2	8.2
	1	228	0.4	1.3	5.6
	2	209	0.4	1.3	4.8
	3	181	0.4	1.3	4.0
ABACUS_shell_ud	0	576	0.5	1.8	35.0
	1	485	0.5	1.8	29.5
	2	414	0.5	1.4	24.4
	3	393	0.5	1.6	22.2
cz40948	0	183	1.9	0.5	23.7
	1	166	1.9	6.4	16.6
	2	157	1.9	6.1	14.8
	3	152	1.8	6.1	14.3

Table 3.5: At roughly equal memory costs, larger reduction levels for the Schur complement can improve the convergence rate.

3.2.5 Comparison against other solvers

We compared the performance of the AMES preconditioner against three other popular algebraic preconditioners for solving linear systems, that are the ILUPACK solver developed by Bollhöfer and Saad [16], the Algebraic Recursive Multilevel Method (ARMS) proposed by Saad and Suchomel [89], and the SParse Approximate Inverse preconditioner (SPAI) introduced by Grote and Huckle [53]. As in the previous experiments, for each run we recorded the CPU time from the start of the solution until the initial residual was reduced by 12 orders of magnitude or until the process failed. We declared a solver failure when no convergence was achieved after 5000 iterations of the restarted GMRES method. We selected the parameters carefully to have a fair comparison between different methods. In AMES, following our conclusions from Section 3.2.3, we selected the number of blocks B_i at the first level so that their average size is almost equal to the size of the Schur complement. For every problem we tested different combinations of number of levels n_{lev} of recursive factorization and different values for the dropping threshold parameter $droptol$ for the factorization of the last-level blocks B_i and of the Schur complements. We chose the best combination in terms of memory and time to solution costs for the given problem. Then we tuned the value of the dropping threshold in the ILUPACK, ARMS, SPAI and AINV solvers to have roughly equal memory costs as in AMES, setting the other parameters equal to their default values defined in those packages. The performance of these methods is rather sensitive to the dropping threshold parameter. For example, on the $rma10$ problem, ILUPACK converged in only 9 iterations using the default value $droptol = 0.01$, but the computation costed $\frac{nnz(M)}{nnz(A)} = 8.9$ and $t_f = 45s$; ARMS converged in 26 iterations with the default $droptol = 0.001$, costing $\frac{nnz(M)}{nnz(A)} = 33.9$ and $t_f = 1111s$; and SPAI could not converge in 5000 iterations with $\frac{nnz(M)}{nnz(A)} = 0.19$, using the default value $droptol = 0.6$. The number of levels of recursive factorization in the multilevel methods ILUPACK and ARMS are calculated automatically by the original codes developed by their authors. We point out that the performance comparison between AMES and the other solvers at fixed memory occupation may be a little penalizing for the AINV, FSAI and SPAI preconditioners as one-level approximate inverses inherently need more memory; the ARMS method is a multilevel solver, but it factorizes the diagonal blocks without any permutation.

In Table 3.6, we show the complete results of our experiments. These include number of iterations (*Its*), density ratio ($nnz(M_L + M_U)/nnz(A)$), time costs for the preordering (t_p), factorization (t_f) and solve phase (t_s). We also tested the unpreconditioned GMRES for these matrix problems, and no convergence is achieved. Basically, the AMES preconditioner can exploit more sparsity. This benefits from the use of multilevel combinatorial algorithms to enforce sparsity in the approximate inverse factors. We clearly see the good potential of the multilevel mechanism incorporated in the AMES preconditioner to reduce the number of iterations of Krylov methods, also in comparison to other multilevel solvers at low to moderate memory costs. In our examples, AMES was more robust than these solvers especially at low memory ratios. We emphasize that the comparison was done against some of the most effective matrix solvers in use today.

(a) cz20468

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.26	187	0.3	0.2	4.2
ILUPACK	1.24	2500	-	0.4	40.3
ARMS	1.16	+5000	-	0.1	+6.5
SPAI	1.64	+5000	-	4.0	+8.0

(b) raefsky3

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.54	235	2.4	3.7	10.0
ILUPACK	0.55	1224	-	2.8	25.2
ARMS	2.38	+5000	-	2.4	+23.5
SPAI	1.83	+5000	-	5040	+243.0

(c) ABACUS_shell_ud

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.79	453	0.3	0.8	22.1
ILUPACK	1.82	1411	-	0.5	26.6
ARMS	1.88	+5000	-	0.2	+7.6
SPAI	2.41	+5000	-	11.0	+12.0

(d) sme3Db

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.85	407	3.5	8.4	39.3
ILUPACK	0.74	1210	-	4.1	41.4
ARMS	5.61	+5000	-	39.0	+54.9
SPAI	1.23	+5000	-	3360	+123.0

(e) viscoplastic2

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	3.07	78	0.9	14.3	3.9
ILUPACK	4.00	2500	-	1.6	70.0
ARMS	3.02	+5000	-	0.9	+10.9
SPAI	3.37	+5000	-	244.0	+24.0

(f) cz40948

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.41	170	0.7	0.4	7.4
ILUPACK	1.48	1627	-	1.0	51.1
ARMS	1.70	+5000	-	0.9	+21.8
SPAI	1.64	+5000	-	8.5	+17.2

(g) rma10

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	2.33	164	3.9	13.1	34.5
ILUPACK	2.27	1242	-	8.6	82.9
ARMS	14.30	+5000	-	203.9	+111.3
SPAI	4.84	+5000	-	11280	+180

(h) finan512

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.59	9	0.8	0.5	0.8
ILUPACK	0.62	11	-	0.7	0.1
ARMS	0.58	36	-	0.4	0.5
SPAI	0.61	7	-	4.2	0.2

(i) helm2d03

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.88	6	6.1	4.3	4.6
ILUPACK	0.91	7	-	3.7	0.4
ARMS	0.93	12	-	1.4	1.5
SPAI	0.87	15	-	100.7	2.7

(j) parabolic_fem

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.75	4	4.7	5.7	1.3
ILUPACK	0.68	10	-	5.3	0.5
ARMS	0.76	12	-	2.0	2.0
SPAI	0.77	4	-	175.3	0.8

Table 3.6: Performance comparison of the multilevel approximate inverse preconditioner against other iterative solvers, both one-level and multilevel.

4 Combining the AMES solver with overlapping

An incomplete factorization preconditioner based on a domain decomposition has been proposed in [38]. In this method the subdomains have a small overlap with certain parametrized algebraic boundary conditions. In a domain decomposition framework, incomplete factorizations with overlapping subdomains leads to so-called generalized Schwarz splitting methods [96]. Schwarz methods were developed to solve linear systems of algebraic equations derived from discretizations of PDEs [97]. They can be seen as generalizations of classic iterative methods such as block Jacobi or Gauss-Seidel methods, as they introduce overlap between the blocks. Overlap plays an important role in the convergence of Schwarz methods. It has been shown that using additive Schwarz methods with overlap variables can improve the convergence rate as well as reduce execution times [21]. Inspired by the domain decomposition approaches, Grigori, Nataf and Qu [52] have introduced an *overlapping technique* to enhance the robustness of multilevel incomplete LU factorization preconditioning. This overlapping strategy is based on an algebraic multi-level nested-dissection which reorders the matrix into an arrow form, e.g. using the nested dissection method proposed by George [80].

The multilevel mechanism incorporated in the AMES preconditioner described in the previous chapter is based on a nested dissection-like ordering, and thus it can easily accommodate for overlapping. We have combined the AMES solver with the overlapping strategy with the purpose of improving its robustness and reducing storage costs. In this chapter, we first review the overlapping strategy that adapts the idea of overlapping subdomains from domain decomposition methods to nested dissection-based methods in Section 4.1. We introduce the combination of the overlapping strategy with the AMES solver, elaborate the algorithm and show performance of the combinatorial method in Section 4.2. We have also tested the proposed combinatorial strategy, and the results of the experiments are reported in Section 4.3.

4.1 Background

A well-known technique to accelerate the convergence of domain decomposition methods, is overlapping. Overlapping extends the subdomains to their neighbours [49, 75]. In the popular restricted additive Schwarz (RAS) algorithm [27], the unknowns that belong to the overlapping regions are duplicated, and the preconditioner is built from an enlarged linear system. The solution of the original system is obtained from the solution of the enlarged system by applying a restriction procedure. In RAS, the computation of a subdomain i with overlap can be expressed as $R_i A R_i^T$, where R_i is a restriction operator that restricts the i-th subdomain to the overlapped subdomain. In AMES, the construction is slightly more complex in order to preserve the nested arrow structure computed in the preorder phase. Below we review the overlapping procedure in detail.

Let $\Omega = (V(\Omega), E(\Omega))$ be the graph of A, $V(\Omega)$ denoting the set of vertices and $E(\Omega)$ the set of edges in Ω. If the graph is directed, we denote an edge of E issuing from vertex u to vertex v as (u, v); u is called a predecessor of v, and v a successor of u. If the graph is undirected, we denote the edges of E by non-ordered pairs $\{u, v\}$; u is called a neighbour of v. As in the previous section, we assume that $\mathcal{G}$ is partitioned into p independent non-overlapping subgraphs Ω_1, ..., Ω_p, and we call S the set of separator nodes, straddling between two different partitions. The goal of overlapping is to extend each independent set of Ω by including its direct neighbours, similarly to the overlapping idea used in other domain decomposition methods, for example in the restricted additive Schwarz method [82, 86].

Following [52], we denote by $V(\Omega_{i,ext})$ the separator nodes that are successors of Ω_i,

$$V(\Omega_{i,ext}) = \{v \in V(S) | \exists u \in V(\Omega_i), (u, v) \in E(\Omega)\} \subset V(S), \tag{4.1}$$

and by $V(\Omega_{ext})$ the complete set of successor nodes of all the subdomains

$$V(\Omega_{ext}) = \bigcup_{i=1:p} V(\Omega_{i,ext}). \tag{4.2}$$

Then Ω_i is extended to the set $\hat{\Omega}_i$ as

$$V(\hat{\Omega}_i) = V(\Omega_i) \cup V(\Omega_{i,ext}),\ i = 1, \ldots, p, \tag{4.3}$$

and the separator S is extended to $\hat{S}$ by adding the successors of nodes in $V(\Omega_{ext})$, that is

$$V(\hat{S}) = V(S) \cup \{v \in V(\Omega_i), i = 1, \ldots, p \mid \exists u \in V(\Omega_{ext}), (u, v) \in E(\Omega)\}. \tag{4.4}$$

Using this notation, the overlapped graph of A, $\tilde{\Omega} = (V(\tilde{\Omega}), E(\tilde{\Omega}))$, is introduced as follows. First define the overlapped subgraph $\tilde{\Omega}_i$ and the overlapped separator $\tilde{S}$ as, respectively,

$$V(\tilde{\Omega}_i) = \{(x, i) : x \in \hat{\Omega}_i\},$$

$$V(\tilde{S}) = \{(x, s) : x \in \hat{S}\}.$$

For simplicity we refer to (x, i) as x_i. Then the vertex set $V(\tilde{\Omega})$ of the overlapped graph $\tilde{\Omega}$ is formed by the disjoint union of the $V(\hat{\Omega}_i)$'s and of $V(\hat{S})$ as

$$V(\tilde{\Omega}) = \left(\bigcup_{i \in 1:p} V(\hat{\Omega}_i) \right) \cup V(\hat{S}). \tag{4.5}$$

Recall that, given the union B of a family of sets indexed by the index set I,

$$B = \bigcup_{i \in I} A_i = \bigcup_{i \in I} \{x : x \in A_i\},$$

their disjoint union C is defined as the set

$$C = \bigcup_{i \in I} \{(x, i) : x \in A_i\}.$$

At this stage, it is useful to introduce the two projection operators Π_1 and Π_2 such that

$$\Pi_1 : (x, i) \mapsto x$$

and

$$\Pi_2 : (x, i) \mapsto i.$$

With this notation, the set of edges of the overlapped graph $\tilde{\Omega}$ is defined according to their projection onto the original graph as follows

$$E(\tilde{\Omega}_i) = \{(u_i, v_i) | u_i \in V(\tilde{\Omega}_i), v_i \in V(\tilde{\Omega}_i), (\Pi_1(u_i), \Pi_1(v_i)) \in E(\Omega)\}, \tag{4.6}$$

$$E(\tilde{S}) = \{(u_s, v_s) | u_s \in V(\tilde{S}), v_s \in V(\tilde{S}), (\Pi_1(u_s), \Pi_1(v_s)) \in E(\Omega)\}, \tag{4.7}$$

$$E(\tilde{\Omega}_i, \tilde{S}) = \{(u_i, v_s) | u_i \in V(\tilde{\Omega}_i), v_s \in V(\tilde{S}), (\Pi_1(u_i), \Pi_1(v_s)) \in E(\Omega),$$

$$\nexists v_i \in V(\tilde{\Omega}_i), (u_i, v_i) \in E(\tilde{\Omega}_i)\}, \tag{4.8}$$

$$E(\tilde{S}, \tilde{\Omega}_i) = \{(u_s, v_i) | u_s \in V(\tilde{S}), v_i \in V(\tilde{\Omega}_i), (\Pi_1(u_s), \Pi_1(v_i)) \in E(\Omega),$$

$$\nexists v_s \in V(\tilde{S}), (u_s, v_s) \in E(\tilde{S})\}. \tag{4.9}$$

The following property, established in [52], ensures an equivalence between the equations of the overlapped system and those of the original system.

Proposition 1. *Let Ω be the associated directed graph of a given system of linear equations and u be a vertex of $V(\Omega)$. Let $\tilde{\Omega}$ be the overlapped graph, and let u_i be a vertex of $V(\tilde{\Omega})$ such that $\Pi_1(u_i) = u \in V(\Omega)$. For each edge $(u, v) \in E(\Omega)$, there is a unique $v_j \in V(\tilde{\Omega})$ such that we have both $\Pi_1(v_j) = v$ and $(u_i, v_j) \in E(\tilde{\Omega})$.*

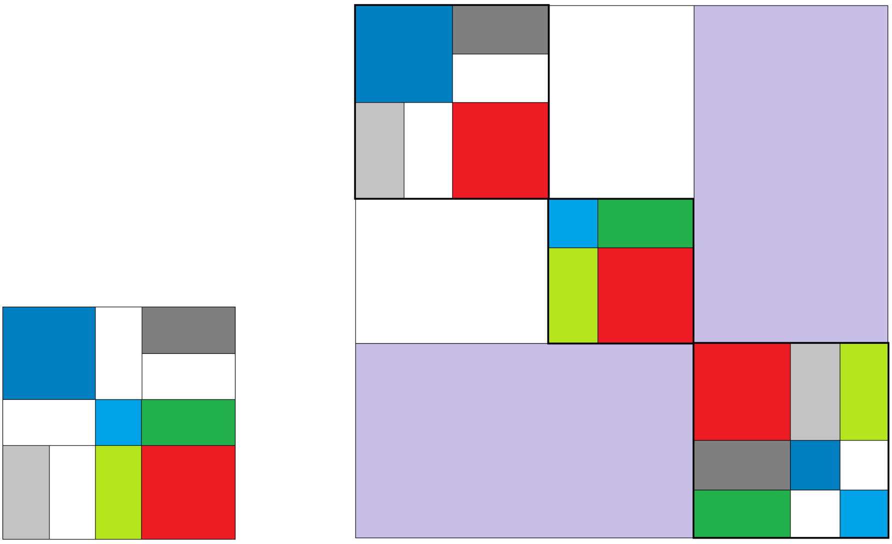
This property shows that there exists a bijection between the nonzeros of the equation corresponding to vertex u in the original system and the nonzeros of the equation corresponding to its dual u_i, where $\Pi_1(u_i) = u$. From a matrix viewpoint, to each nonzero entry $\tilde{a}_{u_i,v_i}$ in the overlapped matrix there is a unique nonzero entry $a_{u,v}$ in the original matrix, where $\Pi_1(u_i) = u$ and $\Pi_1(v_i) = v$. Therefore there is a one-to-one correspondence between equations in the original system and those in the overlapped system. By solving the overlapped system, we can automatically obtain the solution of the original system.

4.1.1 An Example

Below we explain the overlapping procedure on a small example specifically. We display a simple example from [52] to describe briefly how the overlapping procedure works in practice. We consider a 5×5 matrix

$$\left[\begin{array}{cc|c|cc} a_{11} & a_{12} & & a_{14} & a_{15} \\ a_{21} & a_{22} & & & \\ \hline & & a_{33} & a_{34} & a_{35} \\ \hline a_{41} & & a_{43} & a_{44} & \\ a_{51} & & a_{53} & & a_{55} \end{array}\right].$$

This matrix has the structure shown in Figure 4.1(a). There are two subdomains shown by blue blocks, and the separator is shown by red block. The off-diagonal blocks corresponding to the edges between the subdomains and the separator are denoted by grey and green blocks respectively.



(a) The original matrix

(b) The matrix after one-level overlapping

Figure 4.1: Matrix structure before and after applying the overlapping procedure.

After applying the overlapping procedure, both subdomains and the separators are extended following the procedure described in Section 4.1. The structure of the enlarged matrix is shown in Figure 4.1(b). Below we describe the overlapping steps on the example matrix in detail.

The graph of the original matrix consists of two independent subgraphs $\Omega_1 = \{1, 2\}$, $\Omega_2 = \{3\}$ and one separator $S = \{4, 5\}$. We just pick the first subgraph and the separator set to explain. Separator nodes that are successors of Ω_1 are the set

$$V(\Omega_{1,ext}) = \{4, 5\}$$

and we have

$$V(\hat{\Omega}_1) = V(\Omega)_1 \cup V(\Omega_{1,ext}) = \{1, 2, 4, 5\},$$

so that

$$V(\tilde{\Omega}_1) = \{1_1, 2_1, 4_1, 5_1\}.$$

Analogously,

$$V(\Omega_{2,ext}) = \{4, 5\}$$

and

$$V(\Omega_{ext}) = \Omega_{1,ext} \cup \Omega_{2,ext} = \{4, 5\}.$$

Next, we compute the overlapped separator set $\tilde{S}$. The vertices of $V(\Omega_1)$ and $V(\Omega_2)$ directed by $V(\Omega_{ext})$ are $\{1, 3\}$, so

$$V(\hat{S}) = V(S) \cup \{1, 3\} = \{4, 5, 1, 3\}$$

and

$$V(\tilde{S}) = \{4_s, 5_s, 1_s, 3_s\}.$$

According to Equations (4.6)-(4.9), the edges of the overlapped subdomain $E(\tilde{\Omega}_1)$ are defined based on their projection onto the original graph. The first diagonal block of the overlapped matrix is formed by picking the $V(\hat{\Omega}_1) = \{1, 2, 4, 5\}$ rows and columns of the original matrix

$$\begin{array}{c|cccc} & 1 & 2 & 4 & 5 \\ \hline 1 & \diamond & \diamond & \diamond & \diamond \\ 2 & \diamond & \diamond & & \\ 4 & \diamond & & \diamond & \\ 5 & \diamond & & & \diamond \end{array}.$$

The other two diagonal blocks are also formed in the same way, and this is shown in Figure 4.1(b).

From Equation (4.8), we can construct the edges from $\tilde{\Omega}_1$ to $\tilde{S}$. These are the nonzero entries of the overlapped interface block $\tilde{F}_1$, adopting the same notation as in (3.3). We pick the $V(\hat{\Omega}_1) = \{1, 2, 4, 5\}$ rows and $V(\hat{S}) = \{4, 5, 1, 3\}$ columns of the original matrix, and we set the columns corresponding to the common vertices of $\hat{\Omega}_1$ and $\hat{S}$ to zeros. In our example this results in zeroing out the columns of $\hat{F}_1$ indexed by $\hat{\Omega}_1 \cup \hat{S} = \{4, 5, 1\}$, giving

$$\begin{array}{c|cccc} & 4 & 5 & 1 & 3 \\ \hline 1 & \times & \times & \diamond & \\ 2 & & & \diamond & \\ 4 & * & & \times & \times \\ 5 & & * & \times & \times \end{array} \longrightarrow \begin{array}{c|cccc} & 4 & 5 & 1 & 3 \\ \hline 1 & & & & \\ 2 & & & & \\ 4 & & & & \times \\ 5 & & & & \times \end{array}.$$

A similar procedure is followed for the other blocks $\tilde{F}_i$, $\tilde{E}_i$. Finally, the overlapped matrix has the form

$$\left[\begin{array}{cccc|ccc|cccc}
a_{11} & a_{12} & a_{14} & a_{15} & & & & & & & \\
a_{21} & a_{22} & & & & & & & & & \\
a_{41} & & a_{44} & & & & & & & & a_{43} \\
a_{51} & & & a_{55} & & & & & & & a_{53} \\
\hline
 & & & & a_{33} & a_{34} & a_{35} & & & & \\
 & & & & a_{43} & a_{44} & & & & a_{41} & \\
 & & & & a_{53} & & a_{55} & & & a_{51} & \\
\hline
 & & & & & & & a_{44} & & a_{41} & a_{43} \\
 & & & & & & & & a_{55} & a_{51} & a_{53} \\
 & a_{12} & & & & & & a_{14} & a_{15} & a_{11} & \\
 & & & & & & & a_{34} & a_{35} & & a_{33}
\end{array}\right].$$

The block arrow structure of the original matrix is preserved. However, symmetry is lost and the sparsity pattern also changes significantly.

4.2 Analysis and algorithmics

It is interesting to analyse the effect that overlapping may produce on the AMES method. According to (4.3) and (4.6), the size and the number of nonzeros in each subgraph are increased after overlapping. According to (4.8), the interconnections between subdomains and separator are reduced in the overlapped system, as the original interconnectivities are all removed. The more nodes are added to the subgraphs, the richer they are in terms of information about the system matrix, and a larger performance improvement may be expected. In the overlapped system, each block $\tilde{B}_i$ has the following structure

$$\begin{array}{cc}
 & \begin{array}{cc} \quad V(\Omega_i) & \qquad V(\Omega_{i,ext}) \end{array} \\
\begin{array}{c} V(\Omega_i) \\ V(\Omega_{i,ext}) \end{array} & \begin{pmatrix} B & E(\Omega_i, \Omega_{i,ext}) \\ E(\Omega_{i,ext}, \Omega_i) & E(\Omega_{i,ext}) \end{pmatrix}
\end{array}.$$

From Equation (4.1) we see that the set of neighboring nodes $V(\Omega_{i,ext})$ corresponds to the nonzero columns of the block F_i, and the nonzero elements of F_i are determined by the set of interconnections $E(\Omega_i, \Omega_{iext})$. Therefore, the more dense and larger the blocks F_i (that is, the size of separator) in the original matrix, the more nodes and interconnections are added to subdomains, and a larger reduction of the number of iterations can be achieved.

The AMES preconditioning algorithm described in Section 3.1 with one extra overlapping phase can be written as follows:

1. a *scale phase*, where the matrix A is scaled by rows and columns so that the largest entry of the scaled matrix has magnitude smaller than one;
2. a *preorder phase*, where the structure of A is used to compute a suitable ordering that can maximize sparsity in the approximate inverse factors;
3. an *overlap phase*, which extends each block B_i and the Schur complement, and generates the overlapped matrix $\tilde{A}$ and the right-hand side vector $\tilde{b}$;
4. an *analysis phase*, where the sparsity preserving and overlapping orderings are analyzed and an efficient data structure is generated for the factorization;
5. a *factorization phase*, where the entries of $\tilde{A}$ are processed to explicitly compute the approximate inverse factors;
6. a *solve phase*, that accesses all the data structures for solving the overlapped linear system.
7. a *restriction phase*, that restricts the solution obtained from the overlapped system to the original system, and obtains the solution.

4.3 Effects of overlapping

We solved several problems from Table 3.1 combining the AMES method with overlapping after the first level of reordering in (3.3). In these runs, we set $n_{lev} = 2$, and we tuned the *droptol* parameter to have roughly the same memory costs in the experiments with and without overlapping. The experimental results are shown in Table 4.1. The number of iterations (*Its*) are almost the same after overlapping for problems cz20468 and cz40948, while for problems sme3Db, ABACUS_shell_ud and raefsky3 we observed a consistent reduction of the number of iterations *Its* by a factor between 9.5% and 23.8% and of the solving time t_s by a factor between 21.4% and 29.9%. This is in agreement with our analysis of Section 4.2. In Table 4.3 we give the effects of overlapping on the change in size and in number of nonzeros for the overlapped system. In Table 4.3, for each problem we studied the sparsity pattern of block F and the size of blocks B and C before and after overlapping is applied at the first reordering level. The quantity Sp_F denotes the ratio between the number of nonzero elements and the size of F, that is the sparsity degree $\frac{nnz(F)}{size(F)}$. As we can see, the cz20468 and cz40948 problems have the smallest relative size of the separator C and also the smallest value of Sp_F; this means that less information is added to the subdomains. Following the analysis reported in Section 4.2, the overlapping technique is less likely to help on these two matrices, and this is also confirmed by the numerical results. Differently, problems sme3Db and raefsky3 show larger values of $size_C$ and Sp_F and in fact overlapping has a better effect on convergence for these two problems. In our experiments we found that a small number of independent clusters p is recommended to use when overlapping.

Matrix	Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_s
cz20468	overlapping	1.33	147	4.3
	without overlapping	1.32	149	4.5
raefsky3	overlapping	0.56	218	14.3
	without overlapping	0.56	286	20.3
ABACUS_shell_ud	overlapping	3.06	238	13.6
	without overlapping	3.03	263	17.4
sme3Db	overlapping	0.91	389	49.1
	without overlapping	0.91	495	62.5
cz40948	overlapping	1.42	175	12.0
	without overlapping	1.40	172	17.5

Table 4.1: Experiments on the effect of block overlapping on the performance of the multilevel sparse approximate inverse.

Matrix	$\frac{n(A_{overlapped})}{n(A)}$	$\frac{nnz(A_{overlapped})}{nnz(A)}$
cz20468	1.005	1.004
raefsky3	1.134	1.135
ABACUS_shell_ud	1.020	1.019
sme3Db	1.588	1.639
cz40948	1.006	1.004

Table 4.2: The effect of block overlapping on the change in size and in number of nonzeros.

Matrix problem	Method	$size_B$	$size_C$	Sp_F
cz20468	original	20405	63	$1.1e-4$
	after overlapping	20450	116	$4.3e-5$
raefsky3	original	19776	1424	$1.1e-4$
	after overlapping	21184	2864	$5.1e-4$
ABACUS_shell_ud	original	23184	228	$1.3e-4$
	after overlapping	23412	458	$6.1e-5$
sme3Db	original	19956	9111	$9.2e-4$
	after overlapping	25932	20214	$3.4e-4$
cz40948	original	40825	123	$5.2e-5$
	after overlapping	40925	250	$2.4e-5$

Table 4.3: Effects of overlapping on matrix blocks size and sparsity.

We incorporate the overlapping strategy to enhance the robustness and convergence rate of our AMES method, and numerical experiments show good effects of the combination. The overlapping procedure does not destroy the sparsity structure of the matrix, and it can reduce further the interconnections between subdomains and separator set. Based on the analysis and numerical results of the impact of overlapping, both the matrix size and the number of nonzeros would be increased to some extent. The overlapping operation brings in information of the system matrix and accelerates the computation. For the problems that have small relative size of the separator C, less information can be added to the subdomains and hence the overlapping strategy does not improve the convergence rate much. As shown in Table 4.1, if applied on suitable problems, overlapping can accelerate the convergence rate and result in a smaller solve phase time t_s.

5 An implicit variant of the AMES solver

In Chapter 3.2 we have presented a performance analysis of the AMES solver against some state-of-the-art solvers. The numerical experiments have shown that faster convergence rates can be achieved at moderate memory and CPU time costs. However, too much recursive computation can make the explicit computation of the preconditioner time-consuming, especially when many reduction levels are used in the nested dissection ordering applied to the coefficient matrix of the linear system to solve.

To overcome this problem, we propose an improved variant of the AMES algorithm, referred to as Algebraic Multilevel Implicit Solver (AMIS). AMIS is also based on the multilevel recursive nested dissection reordering. The major difference with respect to AMES lies in the solve phase, where the preconditioner is applied implicitly without using the explicit computation of the approximate inverse. We will show that this modification can enhance the overall robustness of the AMES solver.

5.1 AMIS: an Algebraic Multilevel Implicit Solver for general linear systems

For the description of the AMIS algorithm, we utilize the framework of the AMES method proposed in Section 3.1. The main solution process of AMIS also consists of five phases, that are the *scale phase*, the *preorder phase*, the *analysis phase*, the *factorization phase* and the *solve phase*. Below we introduce the new AMIS implementation.

In the scale phase, the linear system $Ax = b$ is scaled by the diagonal scaling matrices D_1 and D_2 such that

$$D_1^{1/2} A y = D_1^{1/2} b, \;\; y = D_2^{1/2} x.$$

In the preorder phase, we use the nested dissection ordering [80] to permute the coefficient matrix of the above system into the block downward arrow structure

$$\tilde{A} = P^T A P = \begin{pmatrix} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_p & F_p \\ E_1 & E_2 & \cdots & E_p & C \end{pmatrix}.$$

For simplicity, we still refer to the linear system to be solved as $Ax = b$. The dissection process is repeated recursively on each submatrix B_i, until a maximum number of levels or a desirable block-size is reached. The permuted matrix $\tilde{A}$ is stored in a suitable data structure that is defined in the analysis phase. Then we factorize $\tilde{A}$ and compute the approximate inverse of the last-level blocks B_i, and the Schur complements S_i at each level.

5.1.1 Improvement of AMIS

Compared to AMES, the main improvement of AMIS lies in **the solve phase**. Below we elaborate the improvement of the AMIS solver with respect to AMES. We write the problem $Ax = b$ in the form

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

where x_1, x_2, b_1 and b_2 are column vectors. We denote the approximate inverse of B by $\tilde{B}^{-1}$ and the approximate inverse of S by $\tilde{S}^{-1}$. Then the equation $Ax = b$ can be rewritten as

$$\begin{aligned} & \begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \\ \Leftrightarrow & \begin{cases} Bx_1 + Fx_2 = b_1 \\ Ex_1 + Cx_2 = b_2 \end{cases} \\ \Leftrightarrow & \begin{cases} Sx_2 = b_2 - E(B^{-1}b_1) \\ x_1 = (B^{-1}b_1) - (B^{-1}F)x_2 \end{cases} \end{aligned} \tag{5.1}$$

where S is the Schur complement matrix $S = C - EB^{-1}F$.

At this stage, in the solve phase we invert the linear system implicitly from Equation (5.1) instead of assembling the approximate inverse of the coefficient matrix explicitly as in Equation (3.10). The new preconditioning operation in AMIS writes as in Algorithm 5.1.

Algorithm 5.1 *The AMIS preconditioning operation.*

```
$p_1 = \tilde{B}^{-1} \cdot b_1$
$x_2 = \tilde{S}^{-1} \cdot (b_2 - E \cdot p_1)$
$p_2 = F \cdot x_2$
$p_3 = \tilde{B}^{-1} \cdot p_2$
$x_1 = p_1 - p_3$
```

At lines 1-3 of the AMES Algorithm 3.1, five vectors need to be computed recursively. Three of these vectors are associated to the multilevel structure of the independent sets

$$\begin{aligned} p_1 &= \tilde{B}^{-1} b_1, \\ p_4 &= \tilde{B}^{-1} F p_2, \\ p_5 &= \tilde{B}^{-1} F p_3, \end{aligned}$$

and two vectors are associated to the multilevel structure of the first level Schur complement matrix

$$\begin{aligned} p_2 &= \tilde{S}^{-1} E p_1, \\ p_3 &= \tilde{S}^{-1} b_2. \end{aligned}$$

In contrast, in the AMIS Algorithm 5.1 only three vectors need to be computed recursively.

$$\begin{aligned} p_1 &= \tilde{B}^{-1} b_1, \\ x_2 &= \tilde{S}^{-1} (b_2 - E p_1), \\ p_3 &= \tilde{B}^{-1} p_2. \end{aligned}$$

In Figure 5.1, we use the matrix `raefsky3` as an example to illustrate how the recursive computation process traverses the tree data structure of the block reordered coefficient matrix. In this example the matrix is permuted into 2 levels. At the first level there are 5 blocks; at the second (last) level, each of those 5 blocks

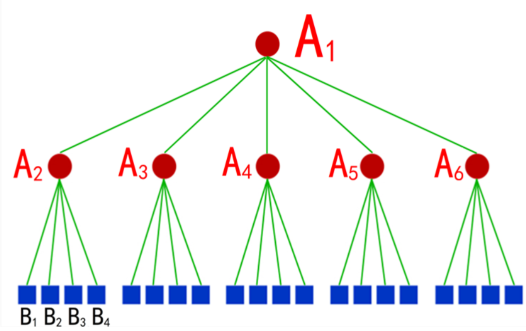


Figure 5.1: The multilevel tree structure of the permuted matrix `raefsky3`.

is divided into 4 blocks. In AMES, to compute $p_1 = \tilde{B}^{-1}b_1$ in A_1, we need to obtain p_1 respectively from $A_2, A_3, \dots, A_6$ at the first level, and then reach the second level to compute the inverse factorization of $B_1, B_2, \dots, B_{20}$. Then we move up level by level to retrieve p_1 in A_1. Clearly, this recursive process is also applied to solve p_4 and p_5. The recursive computation is quite time consuming. On the other hand, in AMIS only the computation of p_1 and p_3 involve recursion. The gain is two-fold. Much computation within the lower levels can be saved during the recursion as well.

In Table 5.1 we give account of the solve phase time t_s for AMES and AMIS expressed in seconds. Here we use the same parameter setting in the numerical tests with both solvers. In the AMES algorithm, the computation of p_1, p_4 and p_5 involving recursion takes a large portion of t_s. In AMIS, only p_1 and p_3 need recursion and thus cost less time. The multilevel strategy is also applied to the Schur complement S_1 in A_1. The size of S_1 and the number of levels in S_1 are set small in this example. Hence the total time consumption involving the recursion in S_1, such as t(p_2) and t(p_3) in AMES and t(x_2) in AMIS, is insignificant compared to t_s. However, we can see that AMES also involves more recursive calls for S_1, and the sum of t(p_2) and t(p_3) in AMES is larger than t(x_2) in AMIS. It is reasonable to conclude that AMIS would save significant time costs for inverting the Schur complement as well.

Figure 5.2 shows the solve phase time t_s required by AMES and AMIS to solve the `raefsky3` problem. Both curves are increasing as the number of iterations grows. As expected, the new implicit variant is cheaper to use than the original one throughout the iterative process. The gain can be even larger if more levels are

(a) AMES

t(p_1)	t(Ep_1)	t(p_2)	t(Fp_2)	t(p_3)	t(Fp_3)	t(p_4)	t(p_5)
5.2	0.2	0.6	0.2	0.5	0.2	9.6	9.6

(b) AMIS

t(p_1)	t(Ep_1)	t(x_2)	t(p_2)	t(p_3)
3.5	0.1	0.4	0.2	6.5

Table 5.1: In the solve phase, the quantities to be computed and the corresponding times to compute them are different in AMES and AMIS.

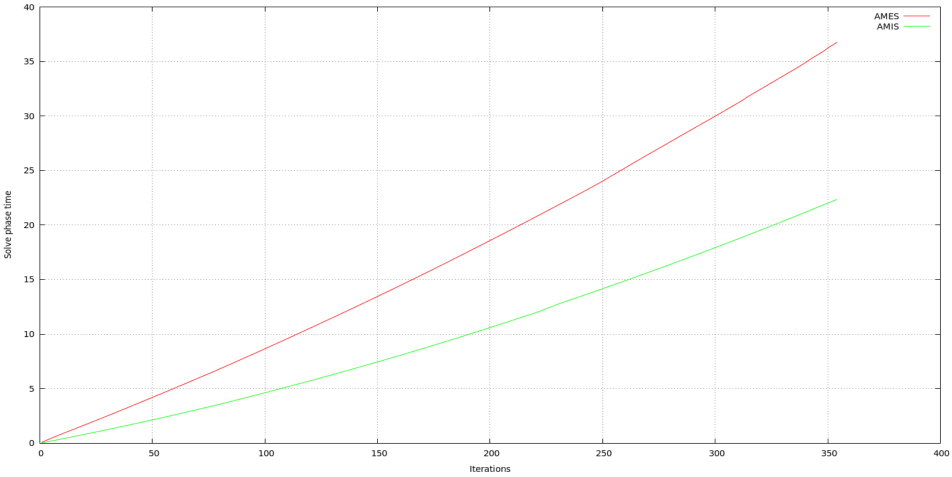


Figure 5.2: The solve phase time of AMES and AMIS with respect to iterations

used in the nested dissection partitioning.

5.2 Numerical experiments with AMIS

We present the results of numerical experiments that illustrate the performance of the AMIS preconditioner. We compare the AMIS method against the AMES method and other popular algebraic preconditioners, ILUPACK [16] and ARMS [89], for solving general linear systems. The selected matrix problems are extracted from the public-domain matrix repository available at the SuiteSparse [37], and arise from various application fields. We present a summary of the characteristics of each linear system in Table 5.2. The codes were developed in Fortran 95. We execute the numerical experiments in double precision floating point arithmetic on a PC equipped with an CPU AMD A8-5600K, 3.6 GHz of frequency, 4 GB of RAM and 4096 KB of cache memory.

Matrix problem	n	Field	$nnz(A)$
cz20468	20,468	Closest Point Method	206,076
raefsky3	21,200	Fluid Structure Interaction	1,488,768
ABACUS_shell_ud	23,412	ABAQUS benchmark	218,484
sme3Db	29,067	3D structural mechanics problem	2,081,063
viscoplastic2	32,769	FEM discretization	381,326
cz40948	40,948	Closest Point Method	412,148
rma10	46,835	3D CFD Model	2,374,001
finan512	74,752	Portfolio optim	596,992
helm2d03	392,257	Helmholtz eq. on a unit square	2,741,935

Table 5.2: Set and characteristics of the test matrix problems.

For both AMES and AMIS solvers, we inverted the last-level blocks using the multilevel inverse-based ILU factorizations, available in the ILUPACK package [16]. We used the default value for the parameters in ILUPACK. For each run we recorded the CPU time from the start of the solution until the initial residual was reduced by 12 orders of magnitude or until the process failed (no convergence was achieved after 5000 iterations of the restarted GMRES method). To give a fair comparison, we tested both solvers with the same values of n_{lev} and p, that are the number of levels of diagonal blocks and number of first-level blocks. We also tuned the dropping threshold to make both solvers have roughly equal memory costs. We declared a solver failure when no convergence was achieved after 5000 iterations of the restarted GMRES method. For the ILUPACK and ARMS solvers, we tuned the value of the dropping threshold so that they had roughly equal memory costs as in AMES and AMIS. The number of levels of recursive factorization in the multilevel methods ILUPACK and ARMS are calculated automatically by the original codes developed by their authors.

The results are reported in Table 5.3. We clearly see the good potential of the multilevel mechanism incorporated in AMES and AMIS to reduce the number of iterations of Krylov methods in comparison to ILUPACK and ARMS at low to moderate memory costs. This performance is in accordance with the numerical comparison in Section 3.2.5. The AMES and AMIS algorithms have the same preorder phase and factorization phase, and we store the same blocks for both methods. Hence they have the same memory requirements for the same parameter setting. We observe that the convergence rates of the two solvers are very similar. This is because the modification to the recursive calls in the solve phase only reduces the amount of calculation to apply the preconditioner, and this does not affect much the number of iterations. In our runs, the implicit variant AMIS performed consistently better than AMES in terms of time consumption. For the solving time t_s, we observe a consistent reduction by a factor between 11% and 65%. The results show that the AMIS solver could simplify the computation in the solve phase and reduce the solving time t_s sometimes dramatically, depending on the parameter setting. By increasing the number of blocks at each level and the number of levels, we obtain a larger and more complicated multilevel tree data structure to handle. Although this helps us exploit sparsity effectively, it also brings in higher complexity due to the recursive computation in the solve phase. By construction, the AMIS method requires less recursion, hence its overall more robust performance.

(a) cz20468

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.26	186	0.5	0.3	4.6
AMIS	1.26	186	0.5	0.2	4.1
ILUPACK	1.24	+5000	-	0.6	+29.6
ARMS	1.60	+5000	-	0.1	+44.1

(b) raefsky3

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.54	235	3.3	6.4	12.0
AMIS	0.54	235	3.3	6.3	9.2
ILUPACK	0.55	+5000	-	4.0	+103.0
ARMS	2.38	1485	-	2.0	28.8

(c) ABACUS_shell_ud

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.79	453	0.5	1.2	23.3
AMIS	1.79	453	0.5	1.3	20.6
ILUPACK	1.82	+5000	-	0.7	+38.1
ARMS	1.88	+5000	-	0.2	+40.0

(d) sme3Db

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.85	407	4.8	12.7	61.0
AMIS	0.85	407	4.9	12.6	38.2
ILUPACK	0.76	+5000	-	5.9	+182.4
ARMS	5.61	+5000	-	31.5	+222.9

(e) viscoplastic2

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	3.61	85	0.9	9.9	8.2
AMIS	3.61	85	0.9	10.0	4.2
ILUPACK	4.07	+5000	-	2.6	+66.5
ARMS	3.37	+5000	-	1.2	+46.3

(f) cz40948

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	1.54	38	1.1	1.2	3.3
AMIS	1.54	38	1.1	1.2	1.6
ILUPACK	1.48	+5000	-	1.2	+64.4
ARMS	1.70	+5000	-	0.4	+101.6

(g) rma10

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	2.39	162	5.1	22.5	69.7
AMIS	2.39	162	5.2	22.4	34.5
ILUPACK	2.25	146	-	12.2	10.8
ARMS	14.30	+5000	-	178.4	+516.4

(h) finan512

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.59	9	1.0	0.7	1.0
AMIS	0.58	9	1.0	0.7	0.4
ILUPACK	0.62	11	-	1.0	0.1
ARMS	0.59	40	-	0.4	0.4

(i) helm2d03

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	*Its*	t_p	t_f	t_s
AMES	0.88	6	5.7	4.7	5.1
AMIS	0.88	6	5.4	4.7	1.8
ILUPACK	0.91	7	-	5.0	0.5
ARMS	0.92	9	-	1.5	0.9

Table 5.3: Performance comparison of the AMES and AMIS solvers.

6 VBAMIS: a variable block variant of the Algebraic Multilevel Implicit Solver

One major source of sparse linear systems in computational science and engineering applications arises from the discretization of partial differential equations (PDEs). When several physical unknown quantities in the underlying PDE are associated with the same grid point, the discretized matrix often exhibits fully dense and typically small nonzero blocks in the sparsity pattern. Examples arise, e.g., in plane elasticity problems with both x- and y-displacement components associated with each grid point, in Navier-Stokes systems for the analysis of turbulent compressible flows where five distinct variables (density, scaled energy, two components of the scaled velocity, and turbulence transport variable) are assigned to each node of the physical mesh, or in cardiac bidomain systems coupling the intra-and extra-cellular electric potentials at each ventricular cell of the heart. These small blocks of fully coupled unknowns in the linear system form clusters of nodes in the adjacency graph of the coefficient matrix. We refer to such clusters as *supernodes*. By reformulating the factorization on the quotient graph built upon the supernodal structure, higher level BLAS kernels can be used, which improve computational efficiency and accelerate the solve phase. The supernodal graph can be much smaller and the factorization time can be shortened. Below the word ‘block’ is used as the matrix counterpart of the word ‘supernode’. We use variable block compressed sparse row storage formats, saving column indices and pointers for the block entries, and high-level BLAS (Basic Linear Algebra Subprograms) [40] routines in the factorization, achieving better cache performance on computer architectures with a hierarchical memory. Better cache reuse is possible for block algorithms. One problem, however, is how to detect the presence of fine-grained dense structures in the linear system automatically, without any user’s knowledge of the underlying problem, and how to exploit them efficiently during the factorization. In this chapter we address these problems for our AMIS solver.

In Section 6.1 we overview some conventional graph compression techniques to compute small dense matrix blocks, such as the checksum-based method, the angle-based method and the graph-based method. In Section 6.2, we introduce a variable block variant of the AMIS solver presented in Chapter 5, designed to exploit the presence of dense components in the solution of the linear system. We refer to this new variant as VBAMIS(Variable Block Algebraic Multilevel Implicit Solver). Some numerical experiments illustrated in Section 6.3 reveal the potential of higher efficiency of VBAMIS against AMIS for solving several general linear systems. We also look at the problem of finding a good parameter setting in the VBAMIS method.

6.1 Graph compression techniques

Below we review some compression techniques that can be used to discover dense matrix blocks in the sparsity pattern of a matrix. These techniques were proposed in connection with the use of block iterative methods, often showing better convergence rates and less solution time compared to their pointwise analogues, see e.g. the studies in [8, 85].

6.1.1 Checksum-based compression method

The first method is due to Ashcraft [5]. The method is based on the idea of traversing the nonzero structure of a matrix A and looking for rows or columns that have the same pattern.

It is simpler to describe the algorithm in terms of graphs. Let $\mathcal{G}(A) = (V, E)$ be the adjacency graph of A, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices and E is the set of edges (v_i, v_j). We call adjacency list $adj(i)$ the set of neighbors v_j of vertex v_i connected by an edge in the adjacency graph. Ashcraft's method looks for the so-called *indistinguishable* vertices $v_i, v_j \in V$ that are by definition vertices with the same adjacency list, or $adj(i) = adj(j)$ using our terminology. To do this, a quantity called *checksum* is assigned to each vertex v_i (or equivalently to the i-th row of A); the checksum quantity is defined as follows

$$chk(i) = \sum_{(i,k)\in E} k. \qquad (6.1)$$

$$B_1 = \left[\begin{array}{c|c|cc|c|c} * & * & 0 & 0 & 0 & * \\ \hline * & * & 0 & 0 & * & * \\ \hline 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 \\ \hline 0 & * & 0 & 0 & * & * \\ \hline * & * & 0 & 0 & * & * \end{array}\right], \quad B_2 = \left[\begin{array}{cc|cc|cc} * & * & 0 & 0 & * & * \\ * & * & 0 & 0 & * & * \\ \hline 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 \\ \hline * & * & 0 & 0 & * & * \\ * & * & 0 & 0 & * & * \end{array}\right]$$

Figure 6.1: Two examples matrices with imperfect (on the left) and perfect (on the right) block structure. Nonzero entries in the matrices are denoted as $*$.

The algorithm marks the ith row of A and loops over the subsequent rows. If two rows i and j have the same checksum value, their patterns are compared; if the patterns are the same, they are added to the same group.

When the matrix is general unstructured, it is still possible to have imperfect dense block structures in the pattern, meaning that some zero entries appear in the blocks. Figure 6.1 shows the difference between a perfect (the left matrix B_1) and imperfect (the right matrix B_2) block matrix structures. B_2 can be viewed as B_1 where the entries $B_1(1,5)$ and $B_1(5,1)$ are treated as nonzeros. Since the checksum-based method only detects rows with the same pattern, it would find only one nontrivial diagonal block in B_1. On the other hand, in matrix B_2 there are three nontrivial diagonal blocks, so we would prefer to treat some zero entries as nonzeros in this situation, and use high level BLAS routines on larger blocks for better efficiency. Some modifications are necessary in the checksum-based compression method to compute imperfect dense blocks, and this is discussed below.

6.1.2 Angle-based compression method

Saad has proposed an angle-based compression method in [85] to compute dense blocks in a sparse matrix. The angle-based compression method compares angles of rows of A. Denoting by C the pattern matrix of A (C has the same pattern of A and entries equal to one), the angle-based compression method computes the upper triangular part of each row i of CC^T. When $j > i$, entry (i, j) in the i-th row of

CC^T is the cosine value between row i and row j.

$$\cos(< i, j >) = (i, j)/(|i| \cdot |j|) > \tau.$$

A large cosine value means a small angle between row i and row j. If this value is big enough, then the angle between row i and row j is small, which means they can be assigned to the same group.

The angle-based compression method depends on a user-defined parameter $\tau \in [0, 1]$. When the cosine value is larger than τ, the two rows are grouped together. When $\tau = 1$, the method computes a perfect block structure. Smaller values of τ could produce larger blocks containing some zero entries. This option can be used to discover imperfect block structures and is computationally attractive as it would enable us to use BLAS routines on larger blocks. Hence choosing the value of τ is a crucial issue. Tuning the τ parameter to optimize the total solution time and storage costs may require several trial tests.

6.1.3 Graph-based compression method

Carpentieri et al. propose a graph-based compression method that requires one simple to use parameter [32]. In the graph-based compression method, the average block density ($b_{density}$) value, defined as the amount of nonzeros in the matrix divided by the amount of elements in the nonzero blocks, is constantly monitored. Parameter $b_{density}$ is assigned to shows the ratio of the number of nonzero entries in A before and after the compression. The value $b_{density} = 1$ means that a fully dense block structure is found in A, while $b_{density} < 1$ means that some zero entries are allowed to be treated as nonzero entries, and the blocks to be computed will have an imperfect dense structure.

Let $\mathcal{G}(A) = (V, E)$ be the adjacency graph of A, and assume that the set of vertices V can be partitioned into disjoint subsets. The quotient graph $\mathcal{Q}(A)$ can be obtained by viewing each subset of vertices as a supervertex. The checksum-based method proceeds by merging the vertices of $\mathcal{G}$ into supervertices of $\mathcal{Q}$. Supervertices with similar adjacent sets are successively merged together. Before actually merging two supervertices I and J, the $b_{density}$ of $I \bigcup J$ needs to be calculated. The algorithm continues until the average block density $b_{density}$ is at least a specified value μ; otherwise it stops. Numerical experiments in [32] show similar performance between the angle-based and the graph-based methods. However, in the graph-based method there is only one simple to use parameter μ to tune, that is the acceptable lower bound for $b_{density}$. In contrast, the angle-based method may need to run the algorithm from scratch several times to find a good value for the compression parameter τ.

6.2 VBAMIS: Variable Block AMIS

In the last section we have introduced several graph compression techniques to compute block orderings for general sparse matrices. These techniques can be used in combination with block methods that often provide better performance than their point-wise counterparts [6, 31, 88]. Block incomplete LU (ILU) factorization methods, when they are applicable, are amongst the most efficient solvers delivering fast convergence rates for some challenging problems [8, 85]. This motivates our interest to embed a blocking strategy into the AMIS algorithm. The new variable block variant of AMIS will be referred to as VBAMIS. We divide the VBAMIS solution of a linear system into five phases:

1. a *blocking phase*, where a suitable block ordering is computed for the coefficient matrix A, to permute it into a new block structured matrix $\bar{A}$;

2. a *preorder phase*, where the block structure of $\bar{A}$ is used to compute a suitable ordering that maximizes sparsity in the approximate inverse factors;

3. an *analysis phase*, where the sparsity preserving ordering is analyzed and an efficient data structure is generated for the factorization;

4. a *factorization phase*, where the nonzero entries of the preconditioner are computed;

5. a *solve phase*, where all the data structures are accessed for solving the linear system.

Below we describe each phase in details.

Blocking phase

In this phase, we compute the block structure of the coefficient matrix A using the matrix compression function `init_blocks` available in the ITSOL package [69]. This routine implements an angle-based compression algorithm described in Section 6.1.2, and requires as a threshold parameter $0 \leq \tau \leq 1$ to decide when merging two rows into the same group. In the ITSOL package [69], the function init_blocks is written in the C language. In our implementation, the function init_blocks is converted into Fortran language.

The init_blocks function returns

- a permutation array *perm* such that $\bar{A} = A(perm, perm)$ has a block structure of the form below

$$\bar{A} = A(perm, perm) = \begin{pmatrix} \bar{A}_{1,1} & \bar{A}_{1,2} & \cdots & \bar{A}_{1,np} \\ \bar{A}_{2,1} & \bar{A}_{2,2} & \cdots & \bar{A}_{2,np} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{np,1} & \bar{A}_{np,2} & \cdots & \bar{A}_{np,np} \end{pmatrix}; \qquad (6.2)$$

- an integer array nB containing the size of each block.
- an integer np, storing the block dimension of $\bar{A}$, which is equal to the number of blocks per row/column of $\bar{A}$.

Upon permutation, all the blocks on the same (block) row will have the same row dimension, and analogously all the blocks on the same (block) column will have the same column dimension. The permutation $\bar{A} = A(perm, perm)$ is equivalent to clustering nodes in the graph of A into different supervertices, and renumbering the nodes in the graph supervertex by supervertex. A nonzero block $A_{i,j}$ in (6.2) expresses the relation that supervertex V_i is connected to cluster V_j in the quotient graph of $\bar{A}$.

We construct $\mathcal{Q}(\tilde{A})$, the quotient graph of $\tilde{A}$

$$\tilde{A} = \begin{cases} \bar{A}, & \text{if } \bar{A} \text{ is symmetric,} \\ \bar{A} + \bar{A}^T, & \text{if } \bar{A} \text{ is nonsymmetric.} \end{cases}$$

In $\mathcal{Q}(\tilde{A})$, each vertex corresponds to one block of $\tilde{A}$, and each edge connects one pair of nonzero blocks of $\tilde{A}$.

To represent the blocked coefficient matrix $\bar{A}$ conveniently in memory, we use the data structure defined in [31], which is called *variable block compressed sparse row* format (referred to as VBCSR). Basically, the VBCSR format is like the CSR format, but the individual entries are blocks; the entries of a nonzero block are stored one after the other in contiguous memory locations. The VBCSR storage format consists of three vectors: one double precision vector a, and two integer vectors ia and ja:

- *a* is an array of the nonzero blocks in the matrix. Each entry of *a* stores all the entries of the block using a dense matrix format.

- *ja* is an array of the column indices of the elements in the *a* vector.
- *ia* stores the locations of the beginning of each block row in the *a* vector.

As an example, consider the nonsymmetric matrix $\bar{A}$ defined by

$$\begin{pmatrix} 2 & 5 & 0 & 0 & 0 & 5 & 7 \\ 3 & 0 & 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 8 & 9 & 3 & 0 & 0 \\ 0 & 0 & 4 & 8 & 6 & 0 & 0 \\ 0 & 0 & 1 & 9 & 8 & 0 & 0 \\ 0 & 0 & 6 & 0 & 2 & 2 & 7 \\ 0 & 0 & 2 & 5 & 3 & 3 & 4 \end{pmatrix},$$

with $nB = (2, 3, 2)$. Then the block dimension of $\bar{A}$ is $np = 3$, and the VBCSR vectors of matrix $\bar{A}$ are:

$$a = ((2, 5, 3, 0), (5, 7, 6, 8), (8, 9, 3, 4, 8, 6, 1, 9, 8), (6, 0, 2, 2, 5, 3), (2, 7, 3, 4))$$
$$ja = (1, 3, 2, 2, 3)$$
$$ia = (1, 3, 4, 6)$$

The VBCSR storage enables us to use easily the level 3 BLAS routines on the individual blocks, as these can be immediately accessed in the data structure and passed as input arguments.

Preorder phase

After the blocking operation the nodes of the initial graph of A are grouped into separate clusters. Each of these clusters represents one node of the quotient graph $\mathcal{Q}(\tilde{A})$. We apply the multilevel graph partitioning algorithms available in the Metis package [61] on $\mathcal{Q}(\tilde{A})$ instead of the quotient graph of the original matrix A. This is an important implementation difference between the VBAMIS and AMES / AMIS solvers. After this operation, $\mathcal{Q}(\tilde{A})$ is partitioned into p non-overlapping subgraphs $\mathcal{Q}_i$ of roughly equal size. For each partition $\mathcal{Q}_i$ we distinguish two disjoint sets of nodes (or vertices): the *interior nodes* that are connected only to nodes in the same partition, and the *interface nodes* that straddle between two different partitions; the set of interior nodes of $\mathcal{Q}_i$ form a so called *separable* or *independent cluster*. Upon renumbering the vertices of $\mathcal{Q}$ one cluster after another, followed by the interface nodes as last, and permuting $\bar{A}$ according to this new ordering, a block bordered linear system is obtained, with coefficient matrix of the form

$$\tilde{A} = P^T \bar{A} P = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_p & F_p \\ E_1 & E_2 & \cdots & E_p & C \end{pmatrix}. \tag{6.3}$$

In Equation (6.3), the diagonal blocks B_i correspond to the interior nodes of $\mathcal{Q}_i$, and the blocks E_i and F_i correspond to the interface nodes of $\mathcal{Q}_i$; the block C is associated to the mutual interactions between the interface nodes. In the multilevel scheme we apply the same block downward arrow structure to the diagonal blocks B_i of $\tilde{A}$; the procedure is repeated recursively until a maximum number of levels is reached, or until the blocks at the last level are sufficiently small to be easily factorized.

Analysis phase

In this phase, a suitable data structure for storing the linear system is defined, allocated and initialized. We use a tree structure to store the block bordered form (6.3) of $\tilde{A}$. The root is the whole graph $\mathcal{Q}$, and the leaves at each level are the independent clusters of each subgraph. Each node of the tree corresponds to one partition $\mathcal{Q}_i$ of $\mathcal{Q}(\tilde{A})$, or equivalently to one block B_i of matrix $\tilde{A}$. A new block data structure is defined to represent the block bordered coefficient matrix $\tilde{A}$. In this data structure, the blocks E and F at each level are stored in the VBCSR format. Blocks E_i and F_i can be very sparse, as many of their rows and columns can be zero. The last-level blocks B_i and the Schur complements S_i are stored in the CSR format as in the current implementation we are still using pointwise factorization algorithms to invert the last level solvers.

Factorization phase

The approximate inverse factors $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ of $\tilde{A}$ can be written in the following form

$$\tilde{L}^{-1} \approx \begin{pmatrix} U_1^{-1} & & & & W_1 \\ & U_2^{-1} & & & W_2 \\ & & \ddots & & \vdots \\ & & & U_p^{-1} & W_p \\ & & & & U_S^{-1} \end{pmatrix}, \; \tilde{U}^{-1} \approx \begin{pmatrix} L_1^{-1} & & & & \\ & L_2^{-1} & & & \\ & & \ddots & & \\ & & & L_p^{-1} & \\ G_1 & G_2 & \cdots & G_p & L_S^{-1} \end{pmatrix}$$

where

$$B_i = L_i U_i, W_i = -U_i^{-1} L_i^{-1} F_i U_S^{-1}, G_i = -L_S^{-1} E_i U_i^{-1} L_i^{-1} \tag{6.4}$$

and L_S, U_S are the triangular factors of the Schur complement matrix

$$S = C - \sum_{i=1}^{p} E_i B_i^{-1} F_i. \tag{6.5}$$

During the factorization, fill-ins in $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ only occur within the nonzero blocks. We apply the arrow structure (6.3) recursively to the diagonal blocks and to the first-level Schur complement as well, so that most of the nonzero entries of the original matrix are clustered into a few nonzero blocks. This effectively maximizes sparsity in the inverse factors and reduces the factorization costs. The multilevel factorization algorithm requires to invert only the last-level blocks and the small Schur complements at each reordering level. The blocks W_i, G_i do not need to be assembled explicitly, as they may be applied using Equation (6.4). The last-level blocks are factorized by ILUPACK. The Schur complements are assembled as in Equation (6.5). First the products $B_i^{-1} \times F_i$ are computed by the ILUPACK routine in CSR format. To exploit the block structure within the matrix, we use the level 3 BLAS routines to compute the multiplications of the dense block entries. The matrix-matrix operations $E_i \times B_i^{-1} F_i$ and $C - \sum_{i=1}^{p} E_i B_i^{-1} F_i$ are performed block-wise efficiently.

Solve phase

For the last-level blocks B and Schur complements at each level, we compute their inverse factorization as what we do in AMES and AMIS. Owing to the merit of the implicit factorization strategy, we observed better performance on the AMIS solver comparing to the AMES solver. Hence in the VBAMIS solver we apply a similar strategy to that of the AMIS solver. Instead of computing the inverse factors of matrix A explicitly, we only compute the approximate inverse factorization of the blocks B that are not at the last level recursively in the way as we do in AMIS.

The equation $Ax = b$ is rewritten as

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$
$$\Leftrightarrow \begin{cases} Bx_1 + Fx_2 = b_1 \\ Ex_1 + Cx_2 = b_2 \end{cases}$$
$$\Leftrightarrow \begin{cases} Sx_2 = b_2 - E(B^{-1}b_1) \\ x_1 = (B^{-1}b_1) - (B^{-1}F)x_2 \end{cases} \tag{6.6}$$

where S is the Schur complement matrix $S = C - EB^{-1}F$. The preconditioning operation is the same as Algorithm 5.1, except the operations are in VBCSR format.

6.3 Numerical experiments with VBAMIS

In this section we present the results of our numerical experiments to illustrate the performance of the VBAMIS preconditioner. First we compare VBAMIS against AMIS for solving general linear systems. Then we explore further the sensitivity of the parameter selection in the VBAMIS solver. The selected matrix problems are extracted from the public-domain matrix repository available at the SuiteSparse [37], and arise from various application fields. We present a summary of the characteristics of each linear system in Table 6.1. The codes are developed in Fortran 95. We execute the numerical experiments in double precision floating point arithmetic on a PC equipped with an CPU AMD A8-5600K, 3.6 GHz of frequency, 4 GB of RAM and 4096 KB of cache memory.

Matrix problem	n	Field	$nnz(A)$
`stacom`	8,415	Compressible flow	271,936
`ABACUS_shell_ud`	23,412	ABAQUS benchmark	218,484
`cz20468`	20,468	Closest point method	206,076
`cz40948`	40,948	Closest point method	412,148
`raefsky3`	21,200	Fluid structure interaction	1,488,768
`olafu`	16,146	structural problem	1,015,156
`venkat01`	62,424	Unstructured 2D Euler solver	1,717,792

Table 6.1: Set and characteristics of the test matrix problems.

6.3.1 Numerical comparison between AMIS and VBAMIS

In Table 6.2, we present a numerical comparison between the AMIS and VBAMIS solvers to show the benefit of the block variant strategy.

For both solvers, we inverted the last-level blocks using the multilevel inverse-based ILU factorization available in the ILUPACK package [16], with default values for the parameters. For each run we recorded the CPU time from the start of the solution until the initial residual was reduced by 12 orders of magnitude or until the process failed (no convergence was achieved after 5000 iterations of the restarted GMRES method). To give a fair performance comparison, we used the same values of n_{lev} and p, that are the number of levels of diagonal blocks and number of first-level blocks, in both methods. We also tuned the dropping threshold parameter to have roughly equal memory costs. For the VBAMIS solver, we initially set $\tau = 1.0$ and decreased it by 0.1 in each run; then we selected the value of τ which gave us the best performance for the given problem.

We recorded the following performance measures:

1. the density ratio $\frac{nnz(M)}{nnz(A)}$, that is the ratio between the number of nonzeros in the preconditioner matrix M versus the number of nonzeros in the coefficient matrix A;

2. the number of iterations Its required to reduce the initial residual by 12 orders of magnitude starting from the zero vector;

3. the CPU time cost in seconds for completing the preorder phase (denoted by t_p), for constructing the approximate inverse factorization (t_f), and for

solving the linear system (t_s). For the VBAMIS method, the time of the blocking phase is counted in t_p. The column t_{all} gives the total CPU time $t_{all} = t_p + t_f + t_s$.

4. the average block size of A after the compression (denoted by b_size), and the ratio of the number of nonzero entries in A before and after the compression (denoted by $b\_density$). It is $b\_density = 1$ if the blocks obtained after compression are all fully dense nonzero blocks, and $b\_density < 1$ means that some zero entries in the blocks are treated as nonzeros. These two quantities do not apply to the AMIS method and hence we use the symbol "-".

From the numerical results reported in Table 6.2 we can conclude that the block variant solver VBAMIS performs better than AMIS with respect to convergence rate and time consumption. The VBAMIS method requires a blocking operation, and hence the time of t_p is generally higher than the AMIS method. For the matrix problems `stacom`, `raefsky3` and `venkat01`, the average size of blocks b_size is relatively large, which means a good compression is achieved. Smaller compressed systems in VBAMIS result in a faster factorization phase than the AMIS method. In the case of matrix problems `cz20468` and `cz20468`, we observe that the results of VBAMIS and AMIS are close. In these two cases, the average block size $b\_size = 1.04$ implies that the matrices are hardly compressed and not many blocks are formed. The VBAMIS solver is based on the block structure. If the matrix has no block structure or the blocking algorithm could not generate a matrix with good block structure, the block algorithm may require more computation cost and perform worse than the point-wise solvers.

In Table 6.2, the optimal performance of matrix `raefsky3` appears when $\tau = 1.0$. We present the block arrow structure of matrix `raefsky3` with $\tau = 1.0$ in Figure 6.2(a). In Figure 6.2(b) we present one last-level block with block structure, whose entries are small blocks of dimension 8. The original size of the matrix is 21200, and the size of matrix in Figure 6.2 is dramatically reduced to 2650 by treating small blocks as its entries. The blocking phase results in $b\_size = 8.00$ and $b\_density = 1.00$, which means the blocks involved in computation are relatively large and fully dense. The much smaller dimension of problem and dense block structure enhance the efficiency of level 2 and 3 BLAS block-wise operation and produce better convergence rate and time performance.

(a) stacom

Method	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS	3.20	9	0.65	2.61	0.22	3.48	-	-
VBAMIS	3.03	7	0.62	2.01	0.23	2.86	2.42	0.97

(b) cz20468

Method	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS	1.29	168	0.47	0.28	3.43	4.18	-	-
VBAMIS	1.29	167	0.77	0.30	3.33	4.40	1.04	0.90

(c) cz40948

Method	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS	1.23	397	0.93	0.55	26.99	28.47	-	-
VBAMIS	1.27	367	1.73	1.41	23.13	26.27	1.04	0.90

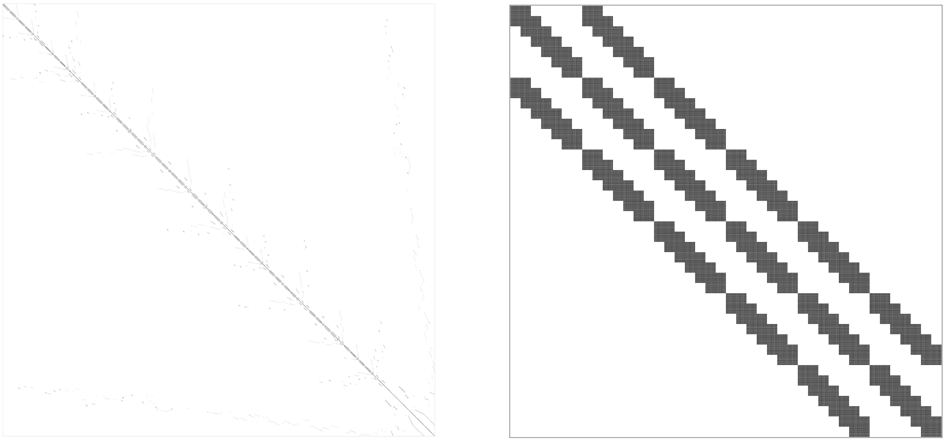
(d) raefsky3

Method	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS	0.87	351	3.50	15.19	21.91	40.60	-	-
VBAMIS	0.72	297	3.39	3.65	13.35	20.39	8.0	1.00

(e) venkat01

Method	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS	3.87	12	3.65	22.47	1.37	27.49	-	-
VBAMIS	3.65	7	4.02	9.34	2.76	16.12	4.00	1.00

Table 6.2: Performance comparison of the AMIS and VBAMIS solvers.



(a) `raefsky3`

(b) One last-level block

Figure 6.2: The block arrow structure of matrices `raefsky3` and one last-level block with block structure with each block of size 8.

6.3.2 The impact of τ on VBAMIS

To inspect the impact of the parameter τ on the performance of VBAMIS in terms of convergence rate and CPU time, we ran some more experiments. For each run with a different value of τ, we recorded the CPU time from the start of the solution until the initial residual was reduced by 12 orders of magnitude or until the process failed. We declared a solver failure when no convergence was achieved after 5000 iterations of the restarted GMRES method. We also recorded the memory cost $\frac{nnz(M_L+M_U)}{nnz(A)}$ and the number of iterations *Its*. The numerical results are displayed in Table 6.3. The column *b_size* shows the average block size of A after the compression, and the column *b_density* shows the ratio of the number of nonzeros in A and $\bar{A}$. The value $b\_density = 1$ means that the nonzero blocks in $\bar{A}$ are fully dense, whereas $b\_density < 1$ means that some blocks in $\bar{A}$ contain some zero entries.

For each run, we applied VBAMIS on the selected matrices and reduced the value of τ by 0.1 from 1.0 to 0.6. We tuned the dropping threshold to make the memory cost for each setting of τ roughly the same, and compared their convergence rate and time cost. On the `olafu` and the `cz40948` problems, when τ was reduced from 1.0 to 0.9, VBAMIS converged relatively fast and used less solution time. For the `cz20468` problem, the best performance was obtained when $\tau = 0.7$. The quantities *b_size* and *b_density* did not vary much for the `cz20468` and `cz40948` problems, and the performance for different values of τ were quite close. In most cases, reducing the value of τ enabled us to enlarge the size of blocks, treat some zero entries in each block as nonzeros, increase b_size and decrease $b\_density$. Due to larger blocks and smaller factorization costs, it was possible to achieve faster convergence, provided *b_density* was around 0.90 or larger. To sum up, a large value of τ, e.g. 1.0 or 0.9, is recommended to use in VBAMIS.

Small values of τ introduced too many zero entries to be treated as nonzeros in the data structure, increasing significantly the memory expenses and factorization costs, and this penalised the performance of VBAMIS. The results on the `raefsky3` problem are slightly different. The best performance was obtained when $\tau = 1.0$ and the results did not improve much when we tuned τ. In this case $b\_density$ remained around 1 as we tuned τ. The value of b_size did not show significant change as we tuned τ, especially in `raefsky3` the initial b_size was already very large.

We choose two typical matrices `cz20468` and `raefsky3` from Table 6.3,

(a) cz40948

τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	1.36	227	1.72	0.62	10.56	12.90	1.04	0.90
0.9	1.36	213	1.60	0.61	9.63	11.84	1.04	0.90
0.8	1.36	230	1.70	0.64	10.75	13.09	1.05	0.89
0.7	1.36	216	1.68	0.63	9.66	11.97	1.07	0.85
0.6	1.36	215	1.62	0.59	9.68	11.89	1.22	0.66

(b) cz20468

τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	1.14	344	0.74	0.28	10.38	11.40	1.04	0.90
0.9	1.15	336	0.74	0.28	9.93	10.95	1.04	0.90
0.8	1.15	331	0.72	0.30	9.70	10.72	1.05	0.89
0.7	1.14	331	0.73	0.31	9.69	10.73	1.07	0.85
0.6	1.14	332	0.72	0.29	9.76	10.77	1.22	0.66

(c) raefsky3

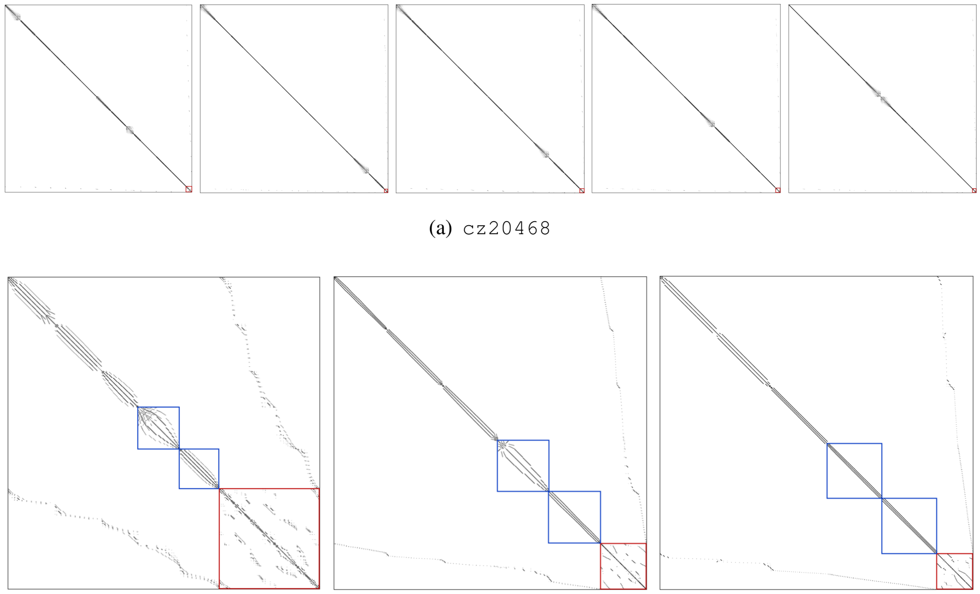
τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	1.54	10	3.21	4.71	0.70	8.62	8.00	1.00
0.9	1.54	10	3.34	4.74	0.69	8.77	8.00	1.00
0.8	1.55	26	3.22	15.71	1.43	20.36	8.63	0.89
0.7	1.55	26	3.29	15.74	1.44	20.47	8.63	0.89
0.6	1.56	41	3.08	39.32	2.80	45.20	16.02	0.39

(d) olafu

τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	2.01	237	1.77	9.76	16.81	28.34	1.54	1.00
0.9	2.05	75	1.12	6.12	5.20	12.44	5.10	0.89
0.8	2.17	209	1.08	9.14	14.62	24.84	6.47	0.64
0.7	2.05	168	1.08	8.13	11.38	20.59	7.23	0.59

Table 6.3: Large value of τ are recommended to obtain better performance of VBAMIS.

and plot the compressed block structure with differen values of τ. The graphs



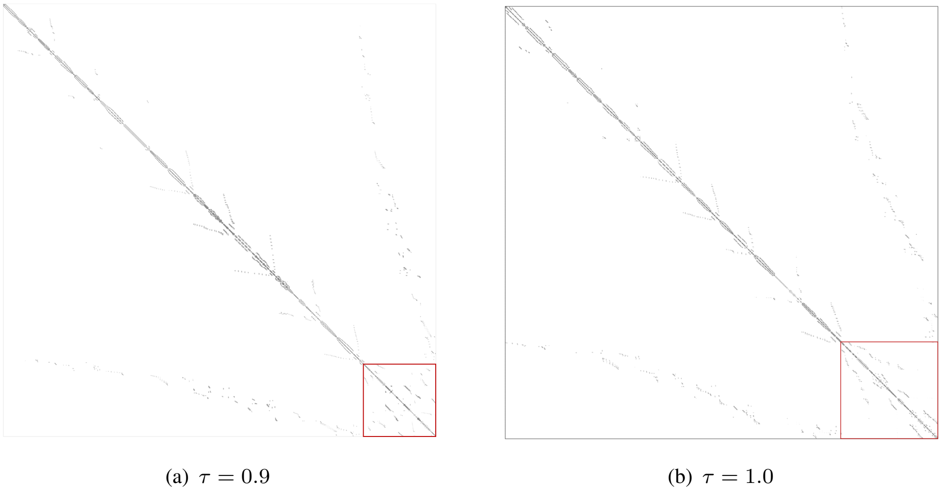
(a) cz20468

(b) raefsky3

Figure 6.3: The block arrow structure of matrices cz20468 and raefsky3 with different values of τ.

of the compressed block structures are illustrated in Figure 6.3. The first-level complement is marked with a red box. In Figure 6.3(a) the block arrow structures of matrix cz20468 are listed with $\tau = 0.6, 0.7, 0.8, 0.9, 1.0$ from left to right. As shown in Figure 6.3(a), the block structures are nearly unchanged as τ varies. This corresponds to the results in Table 6.3(b) that the choice of τ has little impact on the performance of VBAMIS for this problem matrix. In Figure 6.3(b) the block arrow structures of matrix raefsky3 are listed with $\tau = 0.6, 0.8, 1.0$ from left to right. The cases of $\tau = 0.7$ and 0.9 are omitted since they give exactly the same block structures and computational results as those of $\tau = 0.8$ and 1.0. Figure 6.3(b) shows that as τ increases, block structure with dense blocks can be obtained. Once the number of independent sets p at the first level is selected, a larger τ could result in a smaller Schur complement and larger independent sets, which is also beneficial for the multilevel strategy. However, we observe a different behavior on matrix olafu. In Figure 6.4, larger τ produces smaller blocks, hence smaller supervertices and more interface nodes, which yields a larger Schur complement.

This deteriorates the convergence and time performance, and numerical results in Table 6.3(d) also confirms this. The influence of parameter τ on the block structure of matrices is complicated and calls for more research.



(a) $\tau = 0.9$

(b) $\tau = 1.0$

Figure 6.4: The block arrow structure of matrices `olafu` with different values of τ.

We also ran some tests to analyse the sensitivity to other parameters. In Table 6.4 we show results using a larger value of dropping threshold on the `raefsky3` problem. When we dropped more entries, using $\tau = 0.6$ gave better performance than using $\tau = 0.7$, which is different from that in Table 6.3. In Table 6.5 we show the results of using a different number of independent blocks p on matrix `olafu`. We still applied the same parameter setting on `olafu` as in the tests in Table 6.3 except p. When we used a different value of p, VBAMIS gave the best performance when $\tau = 1.0$ instead of $\tau = 0.9$. The effect of other parameters on VBAMIS is more complex to investigate and requires further investigation.

τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	1.03	34	3.20	4.03	1.49	8.72	8.00	1.00
0.9	1.03	34	3.27	4.04	1.48	8.79	8.00	1.00
0.8	1.12	260	3.17	15.75	14.13	33.05	8.63	0.89
0.7	1.12	260	3.23	15.76	13.84	32.83	8.63	0.89
0.6	1.03	165	3.10	37.70	9.11	49.91	16.02	0.39

Table 6.4: When the dropping threshold is changed, the performance on matrix `raefsky3` with respect to diverse values of *eps* is different.

τ	$\frac{nnz(M)}{nnz(A)}$	*Its*	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
1.0	2.47	92	1.76	10.47	7.60	19.83	1.54	1.00
0.9	2.49	219	1.14	6.68	15.76	23.58	5.10	0.89

Table 6.5: When we tuned p, the best performance on matrix *olafu* appeared when $eps = 1.0$.

6.3.3 Combining VBAMIS with a direct solver and other iterative solvers

Utilizing VBAMIS as direct solver

The results shown in Chapter 3 indicate that the multilevel mechanism can be effective to reduce the memory burden but, at least in our implementation, tends to increase the cost per iteration. As an attempt of a possible remedy, we ran some experiments with an exact VBAMIS solver, where no entries are dropped during the factorization, the dropping threshold parameter *droptol* is set equal to zero, and the *condest* parameter is set equal to a very large value ($condest > 1000$), and the last-level blocks and the Schur complements are inverted exactly. In this situation, the VBAMIS method will perform as a direct solver, and thus will be denoted as VBAMIS_direct. For `MA48` we used the Fortran 95 variant which offers additional features to the Fortran 77 version of `MA48`. We executed the numerical experiments with respect to direct solvers in double precision floating point arithmetic. As no approximation is introduced during the factorizations, in each problem we can obtain convergence in one or two iterations, and the solving phase is much cheaper. This can be observed in Table 6.6.

(a) cz40948

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	3.30	-	0.96	0.01	0.97	3.7e-6	2.7e-4	-	-
MA48	2.30	-	1.12	0.004	1.124	1.4e-6	4.4e-4	-	-
VBAMIS_direct	1.54	1.97	0.53	0.07	2.57	1.6e-12	2.4e-7	1.04	0.90

(b) cz20468

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	3.15	-	0.48	0.004	0.484	84.5	4020.1	-	-
MA48	2.27	-	0.55	0.004	0.554	85.0	6404.5	-	-
VBAMIS_direct	1.53	0.74	0.28	0.04	1.06	9.6e-15	3.9e-8	1.04	0.90

(c) raefsky3

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	11.31	-	10.48	0.04	10.52	6.6e-3	4.7e-9	-	-
MA48	7.81	-	25.37	0.03	25.40	6.4e-8	5.3e-13	-	-
VBAMIS_direct	3.94	3.27	8.69	0.36	12.32	2.9e-14	7.3e-16	8.00	1.00

(d) ABACUS_shell_ud

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	13.65	-	0.82	0.01	0.83	3.5e-2	1.8e-2	-	-
MA48	11.38	-	3.25	0.01	3.26	1.9e-2	1.7e-2	-	-
VBAMIS_direct	7.04	0.95	1.56	0.11	2.62	2.9e-6	2.9e-11	1.04	1.00

(e) venkat01

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	10.74	-	8.17	0.05	8.22	2.8e-3	1.3e-5	-	-
MA48	7.61	-	20.00	0.04	20.04	4.2e-13	6.9e-14	-	-
VBAMIS_direct	4.37	3.89	9.02	0.73	13.64	1.1e-15	1.2e-15	4.00	1.00

(f) stacom

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	$\|x-x'\|$	$\|Ax-b\|$	b_size	$b\_density$
MA38	18.20	-	2.20	0.01	2.21	8.8e-4	3.4e-8	-	-
MA48	22.98	-	20.34	0.01	20.35	6.2e-5	8.1e-10	-	-
VBAMIS_direct	4.03	0.62	1.90	0.08	2.60	3.1e-8	7.1e-9	2.42	0.97

Table 6.6: Performance comparison of MA38, MA48 and VBAMIS_direct.

In Table 6.6 we show the numerical results obtained using VBAMIS_direct, MA38 and MA48. Symbol "-" means that the corresponding phase does not apply to the given run. For the VBAMIS_direct solver, we inverted the last-level blocks and the Schur complements using ILU factorizations, using the multilevel implementation available in the ILUPACK package [16]. No approximation is introduced during factorizing the last-level blocks and Schur complements, and we could obtain a good convergence rate for each test matrix. On the other hand, the multilevel mechanism also effectively reduces the memory costs. Comparing against the results of MA38 and MA48, we see that using VBAMIS as a direct solver can save much storage cost only at moderate extra time cost. In Table 6.6 we also report on the forward error $\|x - x'\|$ (x' is the exact solution vector with all the entries equal to 1) and $\|Ax - b\|$. For most given matrices, VBAMIS produced a smaller error comparing to direct solvers. For the case of `cz20468`, MA38 and MA48 could not produce a correct solution. Numerical results show the good robustness and accuracy of VBAMIS applied as a direct solver.

(a) raefsky3

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
VBAMIS_direct	3.94	3.27	8.69	0.36	12.32	8.00	1.00
AMIS_direct	3.93	3.28	17.98	0.27	21.53	-	-

(b) venkat01

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
VBAMIS_direct	4.92	4.57	7.96	0.46	12.99	4.00	1.00
AMIS_direct	5.00	3.69	20.87	0.46	25.02	-	-

(c) cz20468

Method	τ	$\frac{nnz(M_L+M_U)}{nnz(A)}$	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
VBAMIS_direct	1.0	1.53	0.74	0.26	0.04	1.04	1.04	0.90
	0.8	1.53	0.82	0.26	0.04	1.12	1.05	0.89
	0.6	1.54	0.76	0.25	0.03	1.04	1.22	0.66
AMIS_direct	-	1.54	0.52	0.21	0.03	0.76	-	-

Table 6.7: Performance comparison of VBAMIS_direct and AMIS_direct.

We also applied the AMIS method as an exact factorization, denoted by AMIS_direct, without dropping any entries. We present the comparison of AMIS_direct and VBAMIS_direct in Table 6.7. We tuned the dropping threshold to make the memory cost for each method roughly the same, and compared their convergence rate and time cost. For the matrices for which VBAMIS can create large blocks (large b_size), VBAMIS outperforms AMIS. For the matrices like `cz20468` and `cz40948`, reducing τ still can not produce large blocks. Then VBAMIS does not gain much on these matrices, and the performance of AMIS and VBAMIS are nearly the same.

Combining (VB)AMIS with other iterative solvers

Although in our experiments we used the GMRES accelerator, it should be observed that the multilevel preconditioning strategies introduced in this thesis can also be combined with other Krylov solvers. We have applied the MI23 and MI26 routines from HSL and incorporated them into the solve phase of our methods. Routines MI23 and MI26 use the CGS (conjugate gradient squared) method and the BiCGStab (BiConjugate Gradient Stabilized) method to solve an unsymmetric linear system. We denote these two combinations as (VB)AMIS_CGS and (VB)AMIS_BiCGStab, respectively, and the AMIS method accelerated by GMRES as (VB)AMIS_GMRES. According to the results presented in Table 6.8 on the matrix problems extracted from the SuiteSparse [37], and listed in Table 6.1, with the same preconditioner used the three combinations performed roughly equally well. This shows that the AMIS and VBAMIS methods are robust and easy to combine with other state-of-the-art iterative solvers.

(a) raefsky3

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	Its	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS_GMRES	0.95	28	3.30	9.69	1.23	14.73	-	-
AMIS_CGS		26	3.22	9.74	1.89	14.96		
AMIS_BiCGStab		27	3.22	9.78	1.92	14.92		
VBAMIS_GMRES	1.03	34	3.37	3.96	1.47	8.80	8.00	1.00
VBAMIS_CGS		17	3.34	4.05	1.27	8.66		
VBAMIS_BiCGStab		20	3.22	3.98	1.49	8.69		

(b) stacom

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	Its	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS_GMRES	2.95	6	0.65	2.61	0.17	3.43	-	-
AMIS_CGS		3	0.65	2.61	0.11	3.37		
AMIS_BiCGStab		4	0.63	2.60	0.14	3.37		
VBAMIS_GMRES	3.03	7	0.64	2.04	0.22	2.90	2.42	0.97
VBAMIS_CGS		3	0.65	1.99	0.12	2.76		
VBAMIS_BiCGStab		3	0.68	2.01	0.10	2.79		

(c) venkat01

Method	$\frac{nnz(M_L+M_U)}{nnz(A)}$	Its	t_p	t_f	t_s	t_{all}	b_size	$b\_density$
AMIS_GMRES	2.95	7	4.02	28.61	2.72	35.35	-	-
AMIS_CGS		4	4.04	28.41	2.14	34.59		
AMIS_BiCGStab		4	4.06	27.96	1.87	33.89		
VBAMIS_GMRES	3.65	7	4.19	9.19	2.76	16.14	4.00	1.00
VBAMIS_CGS		4	4.11	9.18	2.22	15.51		
VBAMIS_BiCGStab		4	4.03	9.35	2.03	15.41		

Table 6.8: Combination of AMIS/VBAMIS with GMRES/CGS/BiCGStab.

7 Summary and perspectives

7.1 Summary

In this thesis, we have proposed a class of preconditioning methods for the iterative solution via Krylov subspace methods of sparse systems of linear equations arising from computational science and engineering applications. We have presented the main lines of developments of the new methods, and analysed their effectiveness to reduce the number of iterations of Krylov methods and their cost, identifying potential causes of failure and proposing ideas to enhance their robustness. We have assessed their performance to the solution of various matrix problems, also in comparison against some of the most popular algebraic preconditioners in use today. Numerical results demonstrate the good performance of our methods.

The initial concern of the thesis has been to design robust sparse approximate inverse preconditioners. This class of methods offers inherent parallelism as the application phase reduces to performing one or more sparse matrix-vector products, making them very good candidates to solve very large linear systems on massively parallel computers and modern hardware accelerators such as GPUs [26, 74, 100]. We focused in particular on approximate inverse preconditioners in factorized form, as their constructions reduce to solving independent linear systems, and this task can be performed concurrently. One problem that often prevents the use of sparse approximate inverses in practical applications is the lack of a good pattern selection strategy. Indeed, the inverse of a sparse matrix is typically dense and often totally unstructured. This is even more true for the inverse factors, with the result that the approximate inverse can be fairly dense and the preconditioner is costly to compute [104]. This was an important concern addressed in the thesis.

In Chapter 1 we presented the motivations and background behind this research. In Chapter 2 we proposed a short review of modern Krylov subspace methods and preconditioning techniques for solving large linear systems. Krylov subspace methods are iterative in nature and economical in computation. Therefore they are often considered the methods of choice for solving large linear systems such as those arising from the discretization of partial differential equations. We introduced the basic concepts underlying the development of Krylov subspace methods and briefly overviewed several standard algorithm, alongside their comparison in terms of computational and storage costs. Although iterative methods are very efficient memory-wise, they lack the typical robustness of direct methods. Therefore, it is recommended to use a preconditioner to make them converge faster. The aim is to compute a preconditioning matrix such that the equivalent preconditioned system will require a smaller number of iterations to converge to the approximate solution. It is nowadays recognized that preconditioning is a crucial ingredient in the development of efficient solvers for computational science and engineering applications. We have reviewed preconditioning techniques of both explicit and implicit form, with particular attention to Incomplete LU factorization (ILU) and several sparse-approximate-inverse-based preconditioners. We discussed in detail the ARMS and ILUPACK methods. We also revisited the SPAI, AINV and FSAI methods, including recent block variants.

In Chapter 3, we introduced the new solver AMES (Algebraic Multilevel Explicit Solver) that can be defined as a recursive multilevel factorization based on a distributed Schur complements formulation. We presented the development steps of the AMES algorithm. Multilevel graph partitioning algorithms are used to reorder the coefficient matrix, to enhance sparsity and to introduce parallelism in the construction. We designed a suitable data structure for representing the reordered matrix in the computer memory. We assessed the overall performance of AMES by showing numerical experiments on realistic matrix problems. We discussed different parameter settings of the new method, such as the number of independent clusters and the number of partitioning levels. We also put forward suggestions of a reasonable parameter setting. The results reported in Table 3.2 showed that the AMES method can enhance significantly the efficiency of factorized approximate inverse methods like AINV or FSAI. Our techniques can produce very sparse patterns and make the factorization cheap and effective. It is remarkable that they can compete very well with , and sometimes outperform, some of the state-of-the-art methods currently in use today.

In Chapter 4, we presented a combination of an overlapping strategy with the AMES solver, aiming at enhancing the robustness and the convergence rate. We recalled the theoretical background and presented numerical experiments showing the positive effect of overlapping. We observed a consistent reduction of the number of iterations and of the solving time for some of the problems tested.

One practical difficulty with AMES is that although the multilevel mechanisms can be effective to reduce the factorization costs, the application of the preconditioner can be more expensive due to the long oct-tree traverses of the multilevel mechanism. To overcome this problem, in Chapter 5 we have proposed an implicit formulation of AMES that is cheaper to apply. This variant is referred to as AMIS. AMIS is also based on the multilevel recursive nested dissection reordering. The major difference with respect to AMES lies in the solve phase which is applied implicitly. In our numerical tests AMIS consistently outperformed AMES in terms of time to solution. By construction, the AMIS method requires less recursion, hence its overall more robust performance.

In Chapter 6, we combined the AMIS method with graph compression strategies, and proposed a variable block variant referred to as VBAMIS. Block methods are often more effective than their pointwise analogues. We reformulated our AMIS algorithm using level 3 BLAS operations. We use the data structure that is called variable block compressed sparse row format (referred to as VBCSR) to represent block sparse matrices. Numerical results revealed good robustness and higher efficiency obtained by exploiting additionally the matrix block structure. The multilevel mechanism also effectively reduced the memory costs, making the AMIS and VBAMIS methods compete well with existing direct methods.

7.2 Perspectives

For the moment our methods focus on the solution of nonsymmetric systems only, but they can be used equally well for solving symmetric systems. The techniques proposed in this thesis can also be useful for solving realistic engineering applications [23]. To support this point, we show below a case study for solving sparse linear systems arising in computational electromagnetics[1]. The selected matrix problems listed in Table 7.1 are available at the SuiteSparse [37].

Matrix problem	Size	nnz(A)	Field
dw2048	2,048	10,114	Square dielectric waveguide
dw8192	8,192	41,746	Square dielectric waveguide
utm3060	3,060	42,211	Uedge Test Matrix
utm5940	5,940	83,842	Uedge Test Matrix

Table 7.1: Set and characteristics of the test matrix problems from electromagnetism simulations.

[1] This is based on the published article [23].

We solved the right preconditioned system $AMy = b,\ x = My$ instead of $Ax = b$, using the restarted GMRES method [86] as the accelerator for the algebraic multilevel solver developed in this thesis, and compared these results against two other popular algebraic preconditioners for linear systems in use today, that are the Algebraic Recursive Multilevel Method (ARMS) proposed by Saad and Suchomel in [89] and the SParse Approximate Inverse preconditioner (SPAI) introduced by Grote and Huckle in [53]. In Table 7.2 we report on the number of iterations Its, the density ratio $\frac{nnz(M)}{nnz(A)}$, the time t_p for the preorder phase, the time t_f for the factorization phase, the time t_s for the solve phase, and the total CPU time $t_{all} = t_p + t_f + t_s$. Symbol "-" means that the corresponding phase does not apply to the given run. Again, on the solution of a practical real-world application, the techniques developed in this thesis are effective to reduce the number of iterations of Krylov subspace methods, competing very well against some of the state-of-the-art solvers, at low memory costs.

In our experiments, we tuned the dropping thresholds and other parameters to make the solvers have roughly equal memory costs. The results confirm that the implicit variant (AMIS) performed consistently better than the explicit one (AMES) in terms of time to solution, and the block variant (VBAMIS) outperformed both in all respects.

Matrix	Method	$\frac{nnz(M)}{nnz(A)}$	Its	t_p	t_f	t_s	t_{all}
dw2048	AMES	2.31	19	0.04	0.05	0.09	0.18
	AMIS	2.31	19	0.04	0.04	0.03	0.11
	VBAMIS	2.33	14	0.02	0.05	0.01	0.08
	ARMS	2.24	418	-	0.01	0.12	0.13
	SPAI	3.12	+5000	-	0.21	+1.76	+1.97
dw8192	AMES	3.44	53	0.14	0.42	1.54	2.10
	AMIS	3.44	53	0.14	0.44	0.59	1.17
	VBAMIS	3.53	38	0.17	0.91	0.29	1.37
	ARMS	4.26	1063	-	0.05	4.32	4.37
	SPAI	6.98	+5000	-	5.78	+35.59	+41.37
utm3060	AMES	2.93	127	0.12	0.26	0.97	1.35
	AMIS	2.93	127	0.12	0.26	0.56	0.94
	VBAMIS	2.83	32	0.16	0.46	0.14	0.76
	ARMS	2.93	+5000	-	0.05	+9.54	+9.59
	SPAI	3.24	+5000	-	6.06	+5.91	+11.97
utm5940	AMES	3.37	98	0.22	0.88	1.79	2.89
	AMIS	3.37	98	0.24	0.86	0.96	2.06
	VBAMIS	3.27	18	0.30	1.12	0.18	1.60
	ARMS	6.15	+5000	-	0.35	+27.69	+28.04
	SPAI	3.86	+5000	-	21.88	+23.22	+45.10

Table 7.2: Performance comparison of AMES, AMIS, VBAMIS and some state-of-the-art solvers on the electromagnetic problems. The iterative solution wa started from the zero vector, and was stopped when either the initial residual was reduced by twelve orders of magnitude or when no convergence was achieved after 5000 M-V products.

An interesting and challenging source of very large linear systems arises in the solution of Markov chain problems. PageRank [76] in particular is one of the most popular Markov problems. In a separate study, we have investigated the numerical methods for solving the PageRank problem using adaptive reordered methods [25]. Benzi and Tůma have illustrated the good performance of preconditioned Krylov subspace methods on solving the Markov chain problems with large state spaces in [12]. It would be interesting in a future work to test the multilevel methods proposed in this thesis for solving Markov chain problems, as well as in other areas. Following Nick Trefethen, Professor at Oxford University, “nothing will be more central to computational science in the coming decades than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly.” For Krylov subspace methods, this is preconditioning [98].

Samenvatting

In dit proefschrift hebben we een klasse van preconditioneringsmethoden voor de iteratieve oplossing voorgesteld via Krylov-deelruimtemethoden van ijle systemen van lineaire vergelijkingen die voortkomen uit computationele wetenschap en technische toepassingen. We hebben de hoofdlijnen van de ontwikkelingen van de nieuwe methoden gepresenteerd en de doeltreffendheid ervan geanalyseerd om het aantal iteraties van Krylov-deelruimtemethoden en hun kosten te verminderen, potentiële oorzaken van falen te identificeren en ideeën voor te stellen om hun robuustheid te verbeteren. We hebben hun prestaties beoordeeld op de oplossing van verschillende matrixproblemen, ook in vergelijking met enkele van de meest populaire algebraïsche preconditioners die tegenwoordig worden gebruikt. Numerieke resultaten tonen de goede prestaties van onze methoden.

De eerste zorg van het proefschrift was om robuuste, spaarzame approximatieve inverse preconditioners te ontwerpen. Deze klasse van methoden biedt inherent parallellisme wanneer de toepassingsfase vermindert tot het uitvoeren van een of meer ijle matrix-vectorproducten, waardoor ze zeer goede kandidaten zijn om zeer grote lineaire systemen op massaal parallelle computers en moderne hardwareversnellers zoals GPU's op te lossen. We hebben ons met name gericht op benaderde inverse preconditioners in gefactionaliseerde vorm, aangezien hun constructies verminderen tot het oplossen van onafhankelijke lineaire systemen en deze taak tegelijkertijd kan worden uitgevoerd. Een probleem dat vaak het gebruik van schaarse benaderende inverse in praktische toepassingen voorkomt, is het ontbreken van een goede patroonkeuzestrategie. Inderdaad, de inverse van een ijle matrix is meestal dicht en vaak totaal ongestructureerd. Dit geldt zelfs meer voor de inverse factoren, met als gevolg dat de benaderde inverse vrij dicht kan zijn en de preconditioner kostbaar is om te berekenen. Dit was een belangrijke zorg die in het proefschrift wordt behandeld.

In Hoofdstuk 1 hebben we de motivaties en achtergronden van dit onderzoek gepresenteerd. In Hoofdstuk 2 hebben we een korte bespreking voorgesteld van moderne Krylov-deelruimtemethoden en preconditioneringstechnieken voor het oplossen van grote lineaire systemen. Krylov-deelruimtemethoden zijn iteratief van aard en economisch in berekening. Daarom worden ze vaak beschouwd

als de methoden bij uitstek voor het oplossen van grote lineaire systemen zoals die voortkomend uit de discretisatie van partiële differentiaalvergelijkingen. We introduceerden de basisconcepten die ten grondslag liggen aan de ontwikkeling van Krylov-deelruimtemethoden en bekeek kort een aantal standaardalgoritmen, naast hun vergelijking in termen van computer- en opslagkosten. Hoewel iteratieve methoden zeer efficiënt geheugeneffectief zijn, missen ze de typische robuustheid van directe methoden. Daarom wordt aanbevolen om een preconditioner te gebruiken om ze sneller te laten convergeren. Het doel is om een preconditioneringsmatrix zo te berekenen dat het equivalente vooraf geconditioneerde systeem een kleiner aantal iteraties vereist om te convergeren naar de benaderde oplossing. Tegenwoordig wordt erkend dat preconditioning een cruciaal ingrediënt is bij de ontwikkeling van efficiënte oplossers voor computationele wetenschap en technische toepassingen. We hebben preconditioneringstechnieken beoordeeld van zowel expliciete als impliciete vorm, met bijzondere aandacht voor onvolledige LU-ontbinding (ILU) en verschillende sparse-approximate-inverse gebaseerde preconditioners. We bespraken in detail de ARMS- en ILUPACK-methoden. We hebben ook de SPAI-, AINV- en FSAI-methoden opnieuw bekeken, inclusief recente blokvarianten.

In Hoofdstuk 3 hebben we de nieuwe oplosser AMES (Algebraïsche Multilevel Explicit Solver) geïntroduceerd, die kan worden gedefinieerd als een recursieve multiniveau-factorisering op basis van een gedistribueerde formulering van Schur-complementen. We hebben de ontwikkelingsstappen van het AMES-algoritme gepresenteerd. Multilevel grafische partitioneringsalgoritmen worden gebruikt om de coëfficiëntmatrix opnieuw te ordenen, om de sparsiteit te verbeteren en om parallellisme in de constructie te introduceren. We hebben een geschikte gegevensstructuur ontworpen voor het weergeven van de opnieuw ordende matrix in het computergeheugen. We hebben de algehele prestaties van AMES beoordeeld door numerieke experimenten te laten zien op realistische matrixproblemen. We hebben verschillende parameterinstellingen van de nieuwe methode besproken, zoals het aantal onafhankelijke clusters en het aantal partitioneringsniveaus. We doen ook suggesties voor een redelijke parameterinstelling. De resultaten tonen aan dat de AMES-methode de efficiëntie van gefactoriseerde benaderde inverse methoden zoals AINV of FSAI aanzienlijk kan verbeteren. Onze technieken kunnen zeer ijle patronen produceren en de factorisatie goedkoop en effectief maken. Het is opmerkelijk dat ze zeer goed kunnen concurreren met, en soms beter kunnen presteren, met enkele van de meest moderne methoden die tegenwoordig worden gebruikt.

In Hoofdstuk 4 presenteerden we een combinatie van een overlappende strategie met de AMES-oplosser, gericht op het verbeteren van de robuustheid en de convergentietarief. We herinnerden de theoretische achtergrond en presenteerden numerieke experimenten die het positieve effect van overlapping aantoonden. We constateerden een consistente vermindering van het aantal iteraties en van de tijd die nodig was om een aantal van de geteste problemen op te lossen.

Een praktische moeilijkheid met AMES is dat, hoewel de multiniveau-mechanismen effectief kunnen zijn om de factorisatiekosten te verlagen, de toepassing van de preconditioner duurder kan zijn vanwege de lange oct-boom-traverses van het multiniveau-mechanisme. Om dit probleem op te lossen, hebben we in Hoofdstuk 5 een impliciete formulering van AMES voorgesteld die goedkoper is om toe te passen. Deze variant wordt AMIS genoemd. AMIS is ook gebaseerd op de multilevel recursieve geneste dissectieherordening. Het grote verschil met betrekking tot AMES ligt in de oplossingsfase die impliciet wordt toegepast. In onze numerieke tests presteerde AMIS consistent beter dan AMES in termen van tijd tot oplossing. Door constructie vereist de AMIS-methode minder recursie, vandaar de algehele robuustere prestaties.

In Hoofdstuk 6 hebben we de AMIS-methode gecombineerd met strategieën voor grafiekcompressie en een variante blokvariant voorgesteld die VBAMIS wordt genoemd. Blokmethoden zijn vaak effectiever dan hun pointwise-analogen. We hebben ons AMIS-algoritme opnieuw geformuleerd met BLAS-bewerkingen op niveau 3. We gebruiken de datastructuur die een gecomprimeerd dun schijfformaat met variabel blok wordt genoemd (VBCSR genoemd) om blokschaarse matrices weer te geven. Numerieke resultaten lieten goede robuustheid en hogere efficiëntie zien die werd verkregen door het extra gebruik van de matrixblokstructuur. Het multiniveau-mechanisme verminderde ook effectief de geheugenkosten, waardoor de AMIS- en VBAMIS-methoden goed konden concurreren met bestaande directe methoden.

Bibliography

[1] Guillaume Alléon, Michele Benzi, and Luc Giraud. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numerical Algorithms*, 16:1–15, 1997.

[2] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] Douglas N Arnold, Franco Brezzi, Bernardo Cockburn, and L Donatella Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, 2001.

[4] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9, 1951.

[5] Cleve Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing*, 16(6):1404–1411, 1995.

[6] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[7] Michele Benzi, Jane Cullum, and Miroslav Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 22(4):1318–1332, 2000.

[8] Michele Benzi, Reijo Kouhia, and Miroslav Tůma. Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics. *Computer Methods in Applied Mechanics and Engineering*, 190(49):6533–6554, 2001.

[9] Michele Benzi, José Marín, and Miroslav Tůma. A two-level parallel preconditioner based on sparse approximate inverses. In D.R. Kincaid and

A.C. Elster, editors, *Iterative Methods in Scientific Computation IV*, IMACS Series, 1999.

[10] Michele Benzi, Carl D Meyer, and Miroslav Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996.

[11] Michele Benzi and Miroslav Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.

[12] Michele Benzi and Miroslav Tůma. A parallel solver for large-scale Markov chains. *Applied Numerical Mathematics*, 41(1):135–153, 2002.

[13] Matthias Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338:201–218, 2001.

[14] Matthias Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing*, 25(1):86–103, 2003.

[15] Matthias Bollhöfer, Marcus J Grote, and Olaf Schenk. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM Journal on Scientific Computing*, 31(5):3781–3805, 2009.

[16] Matthias Bollhöfer and Youcef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput., Special Issue on the 8-th Copper Mountain Conference*, 27(5):1627–1650, 2006.

[17] Matthias Bollhöfer and Yousef Saad. On the relations between ILUs and factored approximate inverses. *SIAM Journal on Matrix Analysis and Applications*, 24(1):219–237, 2002.

[18] Matthias Bollhöfer and Yousef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing*, 27(5):1627–1650, 2005.

[19] P. Bonnet, Xavier Ferrières, João Paulo Souto Grando, J. C. Alliot, and J L Fontaine. Frequency-domain finite volume method for electromagnetic scattering. *IEEE Antennas and Propagation Society International*

Symposium. 1998 Digest. Antennas: Gateways to the Global Network. Held in conjunction with: USNC/URSI National Radio Science Meeting (Cat. No.98CH36, 1:252–255 vol.1, 1998.

[20] Matthys M Botha. Solving the volume integral equations of electromagnetic scattering. *Journal of Computational Physics*, 218(1):141–158, 2006.

[21] Rafael Bru, Francisco Pedroche, and Daniel B Szyld. Additive schwarz iterations for markov chains. *SIAM Journal on Matrix Analysis and Applications*, 27(2):445–458, 2005.

[22] Yiming Bu and Bruno Carpentieri. A recursive multilevel sparse approximate inverse-based preconditioner for solving general linear systems. In *Proceedings of the 6th International Conference on Numerical Analysis*. Numerical Analysis, 2015.

[23] Yiming Bu, Bruno Carpentieri, Zhao-Li Shen, and Ting-Zhu Huang. Multilevel inverse-based factorization preconditioner for solving sparse linear systems in electromagnetics. *Applied Computational Electromagnetics Society Journal*, 1(4):125–128, 2016.

[24] Yiming Bu, Bruno Carpentieri, Zhaoli Shen, and Tingzhu Huang. A hybrid recursive multilevel incomplete factorization preconditioner for solving general linear systems. *Applied Numerical Mathematics*, 104:141–157, 2016.

[25] Yiming Bu and Tingzhu Huang. An adaptive reordered method for computing PageRank. *Journal of Applied Mathematics*, 2013:1–6, 2013.

[26] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *international conference on computer graphics and interactive techniques*, 23(3):777–786, 2004.

[27] Xiaochuan Cai and Marcus Sarkis. A restricted additive schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, 1999.

[28] Bruno Carpentieri. Fast iterative solution methods in electromagnetic scattering. *Progress in Electromagnetics Research*, 79:151–178, 2008.

[29] Bruno Carpentieri and Matthias Bollhöfer. Symmetric inverse-based multilevel ILU preconditioning for solving dense complex non-hermitian systems in electromagnetics. *Progress in Electromagnetics Research*, 128(14):55–74, 2012.

[30] Bruno Carpentieri, Iain S Duff, Luc Giraud, and Guillaume Sylvand. Combining fast multipole techniques and an approximate inverse preconditioner for large electromagnetism calculations. *SIAM Journal on Scientific Computing*, 27(3):774–792, 2005.

[31] Bruno Carpentieri, Jia Liao, and Masha Sosonkina. VBARMS: A variable block algebraic recursive multilevel solver for sparse linear systems. *Journal of Computational and Applied Mathematics*, 259(1):164–173, 2014.

[32] Bruno Carpentieri, Jia Liao, Masha Sosonkina, Aldo Bonfiglioli, and Sven Baars. Using the VBARMS method in parallel computing. *Parallel Computing*, 57:197–211, 2016.

[33] Edmond Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 1999.

[34] Edmond Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High Perf. Comput. Appl*, 15:56–74, 2001.

[35] Edmond Chow and Yousef Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998.

[36] Bernardo Cockburn. Discontinuous galerkin methods. *Zamm-zeitschrift Fur Angewandte Mathematik Und Mechanik*, 83(11):731–754, 2003.

[37] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1–1:25, 2011.

[38] E De Sturler. Incomplete block LU preconditioners on slightly overlapping subdomains for a massively parallel computer. *Parallel Computing*, 19:129–146, 1995.

[39] H A Van Der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *Siam Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[40] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[41] Jack J Dongarra, Lain S Duff, Danny C Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High-Performance Computers*, volume volume 7 of Software, Environments, and Tools. Society for Industrial and Applied Mathematics, 1998.

[42] Iain S Duff, Albert M Erisman, and John K Reid. *Direct methods for sparse matrices, 2nd edition.* Oxford Science Press, 2017.

[43] Iain S Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.

[44] Iain S Duff and G A Meurant. The effect of ordering on preconditioned conjugate gradients. *Bit Numerical Mathematics*, 29(4):635–657, 1989.

[45] Mark Embree. The tortoise and the hare restart GMRES. *SIAM Review*, 45(2):259–266, 2003.

[46] Massimiliano Ferronato. Preconditioning for sparse linear systems at the dawn of the 21st century: History, current developments, and future perspectives. *Isrn Applied Mathematics*, 2012(3):60–108, 2012.

[47] Massimiliano Ferronato, Carlo Janna, and Giorgio Pini. A generalized Block FSAI preconditioner for nonsymmetric linear systems. *Journal of Computational and Applied Mathematics*, 256(1):230–241, 2014.

[48] Roland W Freund and Noel M Nachtigal. QMR: a quasi-minimal residual method for non-hermitian linear systems. *Numerische Mathematik*, 60(3):315–339, 1991.

[49] David Fritzsche, Andreas Frommer, Stephen D Shank, and Daniel B Szyld. Overlapping blocks by growing a partition with applications to preconditioning. *SIAM Journal on Scientific Computing*, 35(1), 2013.

[50] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 2006.

[51] Gene H. Golub and Henk A. van der Vorst. Closer to the solution: Iterative linear solvers. In *STATE OF THE ART IN NUMERICAL ANALYSIS*, pages 63–92. Cambridge University Press, 1996.

[52] Laura Grigori, Frederic Nataf, and Long Qu. Overlapping for preconditioners based on incomplete factorizations and nested arrow form. *Numerical Linear Algebra With Applications*, 22(1):48–75, 2015.

[53] Marcus J Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[54] Magnus R Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[55] Thomas Huckle, A Kallischko, A Roy, Matous Sedlacek, and Tobias Weinzierl. An efficient parallel implementation of the MSPAI preconditioner. *Parallel Computing*, 36(5):273–284, 2010.

[56] Thomas Huckle and Matous Sedlacek. Smoothing and regularization with modified sparse approximate inverses. *Journal of Electrical and Computer Engineering*, 2010:1–16, 2010.

[57] Ilse C F Ipsen and Carl D Meyer. The idea behind Krylov methods. *American Mathematical Monthly*, 105(10):889–899, 1998.

[58] Carlo Janna, Massimilano Ferronato, and Giuseppe Gambolati. A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems. *SIAM Journal on Scientific Computing*, 32(5):2468–2484, 2010.

[59] Carlo Janna, Massimilano Ferronato, and Giuseppe Gambolati. Enhanced Block FSAI preconditioning using domain decomposition techniques. *SIAM Journal on Scientific Computing*, 35(5), 2013.

[60] Carlo Janna and Massimiliano Ferronato. Adaptive pattern research for Block FSAI preconditioning. *SIAM Journal on Scientific Computing*, 33(6):3357–3380, 2011.

[61] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[62] S A Kharchenko, L Yu Kolotilina, A A Nikishin, and A Yu Yeremin. A robust AINV-type method for constructing sparse approximate inverse preconditioners in factored form. *Numerical Linear Algebra With Applications*, 8(3):165–179, 2001.

[63] L Yu Kolotilina, A A Nikishin, and A Yu Yeremin. Factorized sparse approximate inverse preconditionings. IV: Simple approaches to rising efficiency. *Numerical Linear Algebra With Applications*, 6(7):515–531, 1999.

[64] L Yu Kolotilina and A Yu Yeremin. Factorized sparse approximate inverse preconditionings I: theory. *SIAM Journal on Matrix Analysis and Applications*, 14(1):45–58, 1993.

[65] L Yu Kolotilina and A Yu Yeremin. Factorized sparse approximate inverse preconditioning II: Solution of 3d fe systems on massively parallel computers. *International Journal of High Speed Computing*, 07(02):191–215, 2012.

[66] Dimitri Komatitsch and Jeroen Tromp. Introduction to the spectral element method for three-dimensional seismic wave propagation. *Geophysical Journal International*, 139(3):806–822, 1999.

[67] Karl S Kunz and Raymond J Luebbers. The finite difference time domain method for electromagnetics. *Crc Press*, 1993.

[68] Cornelius Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[69] Na Li, Brian Suchomel, Daniel Osei-Kuffuor, and Youcef Saad. ITSOL v. 2.0: Iterative solvers package. `http://www-users.cs.umn.edu/~saad/software/ITSOL/index.html`.

[70] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[71] Mardochee Magolu Monga Made, Robert Beauwens, and Guy Warzee. Preconditioning of discrete Helmholtz operators perturbed by a diagonal complex matrix. *Communications in Numerical Methods in Engineering*, 16(11):801–817, 2000.

[72] Murat Manguoglu. A domain-decomposing parallel sparse linear system solver. *Journal of Computational and Applied Mathematics*, 236(3):319–325, 2011.

[73] J A Meijerink and H A Van Der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[74] Paulius Micikevicius. 3d finite difference computation on GPUs using cuda. pages 79–84, 2009.

[75] Frederic Nataf, Hua Xiang, Victorita Dolean, and Nicole Spillane. A coarse space construction based on local dirichlet-to-neumann maps. *SIAM Journal on Scientific Computing*, 33(4):1623–1642, 2011.

[76] Larry Page. The PageRank citation ranking: Bringing order to the web. *Stanford Digital Libraries Working Paper*, 9(1):1–14, 1998.

[77] Christopher C Paige and Michael A Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982.

[78] Xiaomin Pan and Xinqing Sheng. Improved algebraic preconditioning for MoM solutions of large-scale electromagnetic problems. *IEEE Antennas and Wireless Propagation Letters*, 13:106–109, 2014.

[79] Xiaomin Pan and Xinqing Sheng. Sparse approximate inverse preconditioner for multiscale dynamic electromagnetic problems. *Radio Science*, 49(11):1041–1051, 2014.

[80] Joseph E Pasciak, Alan George, and Joseph W H Liu. Computer solution of large sparse positive definite systems. *Mathematics of Computation*, 39(159):305, 1981.

[81] Anthony T Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984.

[82] Alfio Quarteroni and Alberto Valli. *Domain decomposition methods for partial differential equations*. Clarendon Press, 1999.

[83] Youcef Saad and Martin H Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[84] Yousef Saad. ILUM: a multi-elimination ilu preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 17(4):830–847, 1996.

[85] Yousef Saad. Finding exact and approximate block structures for ILU preconditioning. *SIAM Journal on Scientific Computing*, 24(4):1107–1123, 2002.

[86] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. SIAM, 2003.

[87] Yousef Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM Journal on Scientific Computing*, 27(3):1032–1057, 2005.

[88] Yousef Saad, Azzeddine Soulaimani, and Ridha Touihri. Variations on algebraic recursive multilevel solvers (ARMS) for the solution of CFD problems. *Applied Numerical Mathematics*, 51:305–327, 2004.

[89] Yousef Saad and B Suchomel. ARMS : an algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra With Applications*, 9(5):359–378, 2002.

[90] Yousef Saad and Jun Zhang. BILUM: Block versions of multielimination and multi-level ilu preconditioner for general sparse linear systems. *SIAM J. Sci. Comput*, 20:2103–2121, 1998.

[91] Peter P. Silvester and Ronald L. Ferrari. *Finite elements for electrical engineers*. Cambridge University Press, 1983.

[92] Valeria Simoncini and Daniel B Szyld. Interpreting IDR as a Petrov-Galerkin method. *SIAM Journal on Scientific Computing*, 32(4):1898–1912, 2010.

[93] Peter Sonneveld. CGS, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52, 1989.

[94] Peter Sonneveld and Martin B Van Gijzen. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2008.

[95] A. Taflove and S. C. Hagness. *Computational Electromagnetics: The Finite-Difference Time-Domain Method.* 2005.

[96] Wei Pai Tang. Generalized schwarz splittings. *SIAM Journal on Scientific and Statistical Computing*, 13(2):573–595, 1992.

[97] Andrea Toselli and Olof B Widlund. Domain decomposition: Algorithms and theory. *International Journal of Computer Mathematics*, 2005.

[98] L.N. Trefethen and III David Bau. *Numerical Linear Algebra.* SIAM Book, Philadelphia, 1997.

[99] John Leonidas Volakis, Arindam Chatterjee, and Leo C Kempel. Finite element method for electromagnetics : antennas, microwave circuits, and scattering applications. *B & T Weekly*, 1998.

[100] Vasily Volkov and James Demmel. Benchmarking GPUs to tune dense linear algebra. pages 1–11, 2008.

[101] Pieter Wesseling and Peter Sonneveld. Numerical experiments with a multiple grid and a preconditioned Lanczos type method. *Lecture Notes in Mathematics*, 771(4):543–562, 1980.

[102] A Yu Yeremin and A A Nikishin. Factorized-sparse-approximate-inverse preconditionings of linear systems with unsymmetric matrices. *Journal of Mathematical Sciences*, 121(4):2448–2457, 2004.

[103] Alex Yu Yeremin, Lily Yu Kolotilina, and A A Nikishin. Factorized sparse approximate inverse preconditionings. III. iterative construction of preconditioners. *Journal of Mathematical Sciences*, 101(4):3237–3254, 2000.

[104] Jun Zhang. Sparse approximate inverse and multilevel block ilu preconditioning techniques for general sparse matrices. *Applied Numerical Mathematics*, 35(1):67–86, 2000.