OVERLAPPING AND NONOVERLAPPING ORDERINGS FOR PRECONDITIONING

A Dissertation
Submitted to
the Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
David Fritzsche
May, 2010

Examining Committee Members:

Daniel B. Szyld, Advisory Chair, Mathematics
Yury Grabovsky, Mathematics
Bejamin Seibold, Mathematics
Andreas Frommer, External Member, Bergische Universität Wuppertal

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

Overlapping and Nonoverlapping Orderings
for Preconditioning

David Fritzsche

DOCTOR OF PHILOSOPHY

Temple University, 2010

Professor Daniel B. Szyld, Chair

Several ordering techniques for block based preconditioning are presented.

The XPABLO algorithm as an extension of the original PABLO and TPABLO algorithms incorporates as a preprocessing step a nonsymmetric permutation combined with row and column scalings to obtain a large diagonal. A more general parametrization can be implemented in XPABLO while keeping the original time complexity of PABLO. It is shown that a block Gauss–Seidel preconditioner can be implemented to have the same execution time as the corresponding block Jacobi preconditioner. Experiments are presented showing that for certain classes of matrices, the block Gauss–Seidel preconditioner used with the system permuted with the XPABLO algorithm can outperform the best ILUTP preconditioners in a large set of experiments.

The new OBGP algorithm extends a given nonoverlapping block ordering to an overlapping ordering to be used for algebraic Schwarz preconditioners. It is shown by experiments that using a multiplicative Schwarz preconditioner based on the extended ordering instead of using a block Gauss–Seidel preconditioner based on the original ordering can result in faster convergence.

ACKNOWLEDGEMENTS

This thesis was written under the terms of the *Agreement for a Jointly Supervised Ph.D. Study Program in Mathematics* between *Bergische Universität Wuppertal* and *Temple University - Of the Commonwealth System of Higher Education.*

I want to thank my advisors Andreas Frommer and Daniel Szyld for all their support and help during my time as a graduate student. Their guidance, their comments, and their encouragements made this thesis possible. They made it possible for me to work and study at the Temple University for more than two years and this time has become an invaluable and unforgettable part of my life.

I also would like to thank the other members of my doctoral committees: Yury Grabovsky and Benjamin Seibold at Temple University and Bruno Lang and Walter Krämer at Bergische Universität Wuppertal. I am very grateful for their patience and for their willingness to participate in the committee.

I want to thank Benjamin Seibold for providing me with a very useful set of test problems. The numerical results obtained by solving these problems are an important part of this thesis and the graph visualizations based on the known geometry for these problems gave me some very helpful additional "feedback" on XPABLO and OBGP.

Last, but not least, I want to thank Sébastien Loisel. He gave me the idea of working with level sets during the development of OBGP.

Their help and suggestions are very much appreciated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

We consider a nonsingular linear system

$$Ax = b, \qquad A \in \mathbb{R}^{n\times n}, \quad x,\, b \in \mathbb{R}^n, \tag{1.1}$$

where n is large and A is sparse. We do not assume A to be symmetric. Note that we call a matrix sparse if most of its entries are zero. We use the $\mathrm{nnz}(A)$ to denote the number of nonzeros in A, i.e.,

$$\mathrm{nnz}(A) := \Big|\big\{(i,j) \,:\, a_{ij} \neq 0\big\}\Big|.$$

Direct methods, based on Gaussian elimination, have reached a mature state for the general system (1.1); see, e.g., [18], [20], [21], [24]. They may, however, suffer from severe fill-in so that memory resources may become insufficient and/or execution times may become too large.

An alternative is iterative methods, such as Krylov subspace methods, which are discussed in section 2.1. In most cases they need less memory, but may suffer from robustness problems or a lack of convergence. The key to robustness and convergence for a broad range of different problems is to find good preconditioners. We will give a short introduction to preconditioning in section 2.2.

In this thesis we explore several preconditioning techniques applicable for iterative methods for the general problem (1.1), see chapters 3, 4 and 5. In all of these preconditioners we need to solve linear systems of (much) smaller size than (1.1). Since we use direct methods to solve these smaller systems our approach can also be viewed as a hybrid method between direct and iterative solvers.

At the end of this chapter we present our test problems. In Chapter 2, we present some background material used in this thesis. In Chapter 3, which builds upon [30], we introduce the XPABLO algorithm, a variant of the PABLO and TPABLO algorithms. XPABLO finds a symmetric permutation of A such that the permuted system has a $q \times q$ block structure, which can be used for block Jacobi and block Gauss–Seidel preconditioning. We also discuss how XPABLO can be combined with a preprocessing step, which permutes and scales the system such that the diagonal is large compared to the off-diagonal part of the matrix. We also give some implementation details of the XPABLO algorithm and analyze its time complexity. In section 3.5.1, we present some results on the robustness of XPABLO-based preconditioning for H-matrices and spd matrices and discuss a way to handle singular blocks of general matrices. Moreover, we show in Chapter 3 how a block Gauss–Seidel preconditioner can be implemented to have the same execution time as the corresponding block Jacobi preconditioner. Finally, extensive numerical experiments show the robustness and low execution time of XPABLO-based preconditioning compared to ILUTP-based preconditioning. In Chapter 4, we present the new OBGP algorithm to grow an existing nonoverlapping block structure into an overlapping one. We also reintroduce the block Gauss–Seidel method and introduce the multiplicative Schwarz method. The block Gauss–Seidel method is presented using the same notation we later use for the presentation of the multiplicative Schwarz method. In this

way we can show that the block Gauss–Seidel method can be seen as the multiplicative Schwarz method without overlap. Furthermore, we discuss how to implement OBGP in an efficient way and analyze the time complexity of OBGP. We also present numerical experiments, which show the benefit of adding overlap compared to not adding overlap. In Chapter 5, we present our work toward an unsymmetric version of XPABLO.

1.1 Test Problems

The test problems used in our numerical experiments come from two different sources and cover several different application areas. Some of the test problems are from the University of Florida Sparse Matrix Collection [17], and the others were kindly provided by Benjamin Seibold; see 1.1.2.

Since the convergence of the iterative solver can depend on the given right-hand side, we need to specify exactly what right-hand side is used in our experiments: If a right-hand side b is provided with the matrix in [17], we use it as the right-hand side for (1.1). If no right-hand side is provided, we use $b := Ae$, where $e = (1, \ldots, 1)^T$ is the vector of all ones. For the `CAVITY16` and `CAVITY26` matrices we use $b := Ae$ as the right-hand side since the provided solution vector was found to be inconsistent with the provided right-hand side.

Some basic properties of our test matrices are summarized in Table 1.1. For each matrix we show the dimension (n), the number of nonzeros (nnz), the 1-norm condition estimate (Condest), the application area (Area) and the UF matrix group (Group). The given condition estimate is the result of MATLAB's `condest` function. If no condition estimate is given, the computation failed for some reason, usually the available memory was not sufficient. The application area and the matrix group are given only for the UF test matrices.

In Table 1.2 we give results obtained by using a direct solver to solve the linear systems of Table 1.1. We employed the UMFPACK [18] sparse (direct) solve package via MATLAB's backslash operator in the expression `x = A \ b`. In the results, oom (out of memory) denotes that the available memory was not sufficient, i.e., MATLAB gave an out of memory error for `x = A \ b`. For a "successful" run we give additional information: The solving time, measured in seconds (Time), the relative residual norm $\|b - Ax\|/\|b\|$ (Rel. Res.), and, if the exact solution x_* is known, the relative error norm $\|x_* - x\|/\|x_*\|$ (Rel. Err.). For both the relative residual norm and the relative error norm x denotes the solution computed by UMFPACK.

1.1.1 University of Florida Sparse Matrix Collection

The University of Florida Sparse Matrix collection (UF) [17] is a large collection of publicly available sparse matrices coming from real applications. As of October 2009, it contains 2272 matrices. We selected our test problems from two application areas: Computational fluid dynamics (CFD) and semiconductor device simulation (SDS). The UF matrix group describes the origin of a matrix.

1.1.2 Poisson's Equation

Consider Poisson's equation

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = g & \text{on } \Gamma_D \\ \dfrac{\partial u}{\partial n} = h & \text{on } \Gamma_N \end{cases} \tag{1.2}$$

where $\Gamma_D \cup \Gamma_N = \partial\Omega$.

Numerical methods can be used to convert the Poisson equation (1.2) into a linear system of the form (1.1). Traditional approaches are finite difference methods and finite element methods. Finite difference methods work well if the discretization

points are placed on a regular grid. However, a regular distribution of the discretization points may not be possible if the geometry is complex or if the points are given from the application. Finite element methods do not suffer from these drawbacks, but they need a mesh and the construction of a mesh can be very expensive. The meshfree finite difference approach which we shortly describe now allows to construct meshfree finite difference stencils for discretization points not on a regular grid without the costly meshing of finite element methods.

Let $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} \subset \bar{\Omega}$ be a *point cloud*, which consists of interior points $X_i \subset \Omega$ and boundary points $X_b \subset \partial\Omega$, i.e., $X = X_i \cup X_b$. The point cloud is meshfree, i.e., no information about connections between points is provided. The meshfree finite difference approach converts problem (1.2) into a linear system

$$A\hat{u} = \hat{f},$$

where the vector $\hat{u}$ contains approximations u_i to the values $u(\boldsymbol{x}_i)$. The ith row of the matrix A consists of the *stencil* corresponding to the point $\boldsymbol{x}_i$.

For our experiments we use discretizations kindly provided by Benjamin Seibold. A detailed description of the different meshfree discretization approaches and the test problem is given in [45]. In the following we give a short overview about these provided discretizations. The problem is to solve Poisson's equation (1.2) in the unit square (for $d = 2$) or unit box ($d = 3$) with a ball cut out, i.e., $\Omega = [0,1]^d \setminus B(m; 0.44)$ where d is the dimension and $B(m; r)$ denotes the ball with midpoint m and radius r. The midpoint is $m = (0.5, 1.1)$ for $d = 2$ and $m = (0.5, 0.5, 1.1)$ for $d = 3$. Moreover, to compare the accuracy of the different discretizations, the problems are constructed such that the solution is known. Given g, choose $f = \Delta g$ and $h = \frac{\partial g}{\partial n}$ so that (1.2)

has the known solution $u = g$. According to [45], the function g is set to

$$g(x_1, x_2) = \frac{1}{c_2}\big(x_1 \sin(x_2 + 2) + x_2 \sin(2x_2 + 1)\big) \qquad \text{in 2d,}$$

$$g(x_1, x_2, x_3) = \frac{1}{c_3}\big(x_1 \sin(x_2 + 2) + x_2 \sin(2x_3 + 3) + x_3 \sin(3x_1 + 1)\big) \qquad \text{in 3d}$$

with c_2 and c_3 such that $\max g - \min g = 1$.

The problem is discretized by a sequence of point clouds. The point clouds have a uniform average density and a minimum separation of $\delta = 0.05$. For a fixed point cloud two different types of approximations for the derivative are considered: The *least squares* (LSQ) approach and the *minimal positive stencil* (MPS) approach.

Figure 1.1 shows the geometry of the point cloud for $n = 1000$ points. Figure 1.2 shows a plot of the graph of the LSQ discretization for the point cloud shown in Figure 1.1.

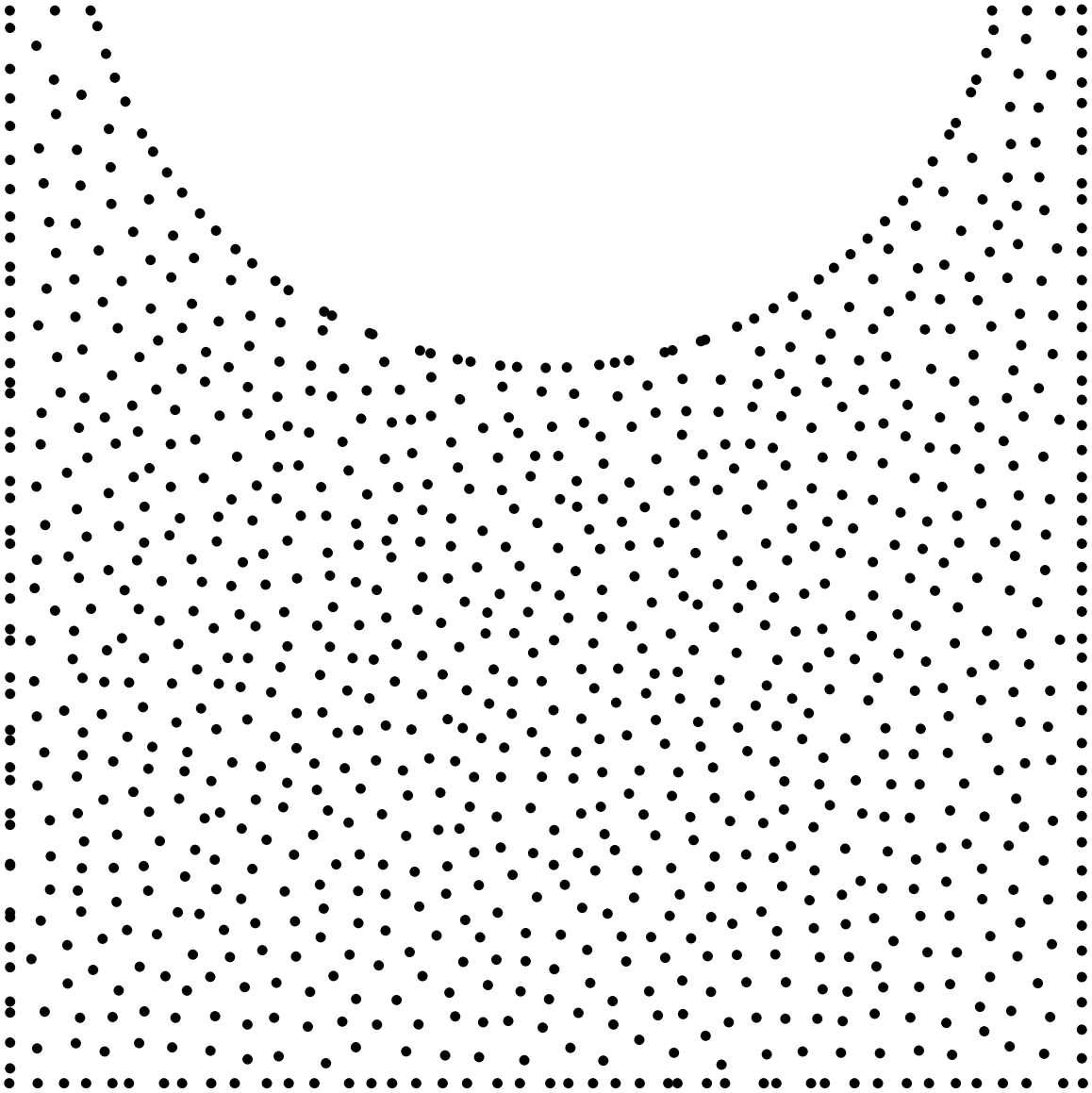


Figure 1.1. Geometry of the 2d Point Cloud for Poisson's Equation ($n = 1000$)

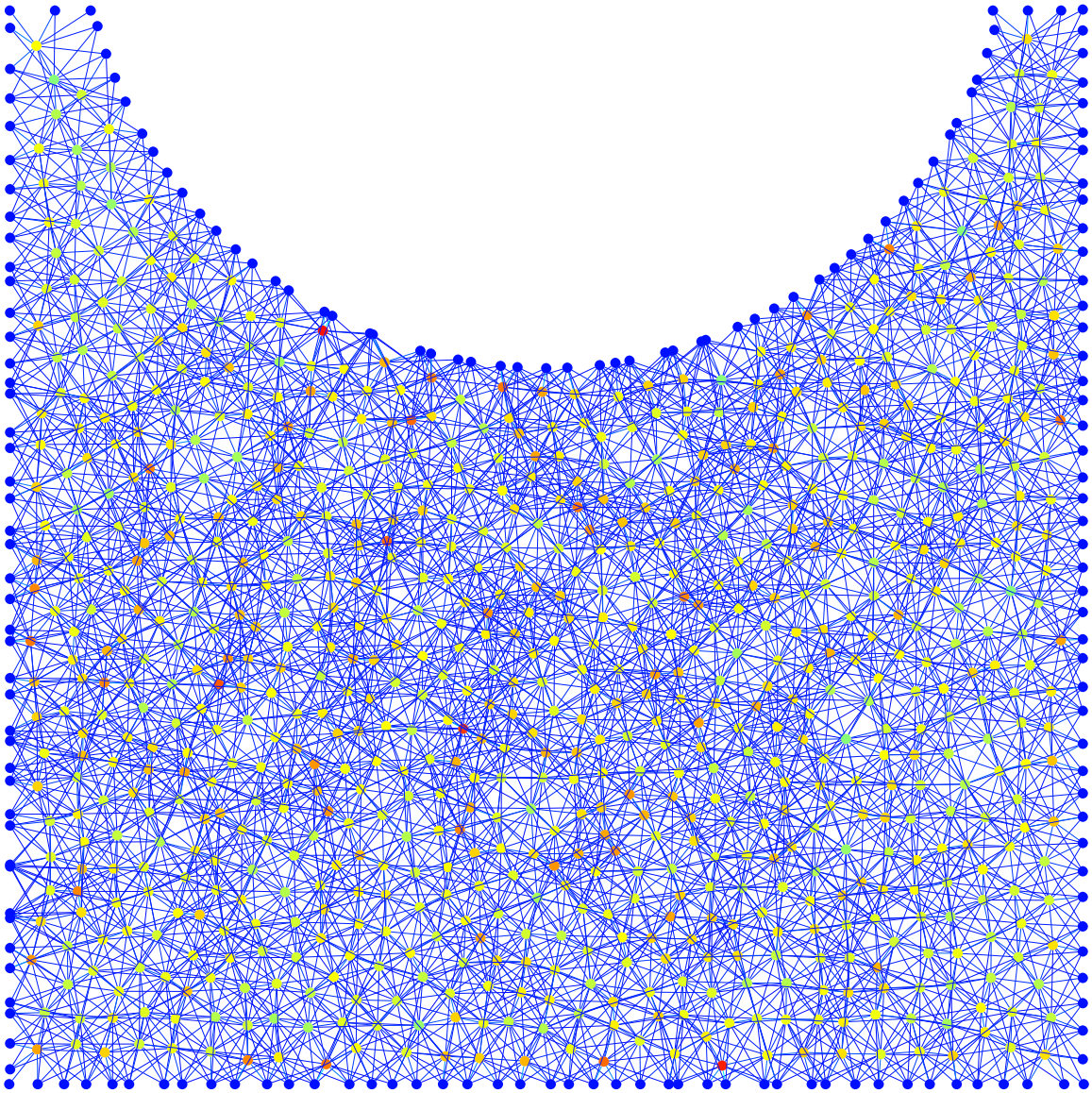


Figure 1.2. Graph of `LSQ_2D_1000` ($n = 1000$, nnz $= 15\,803$)

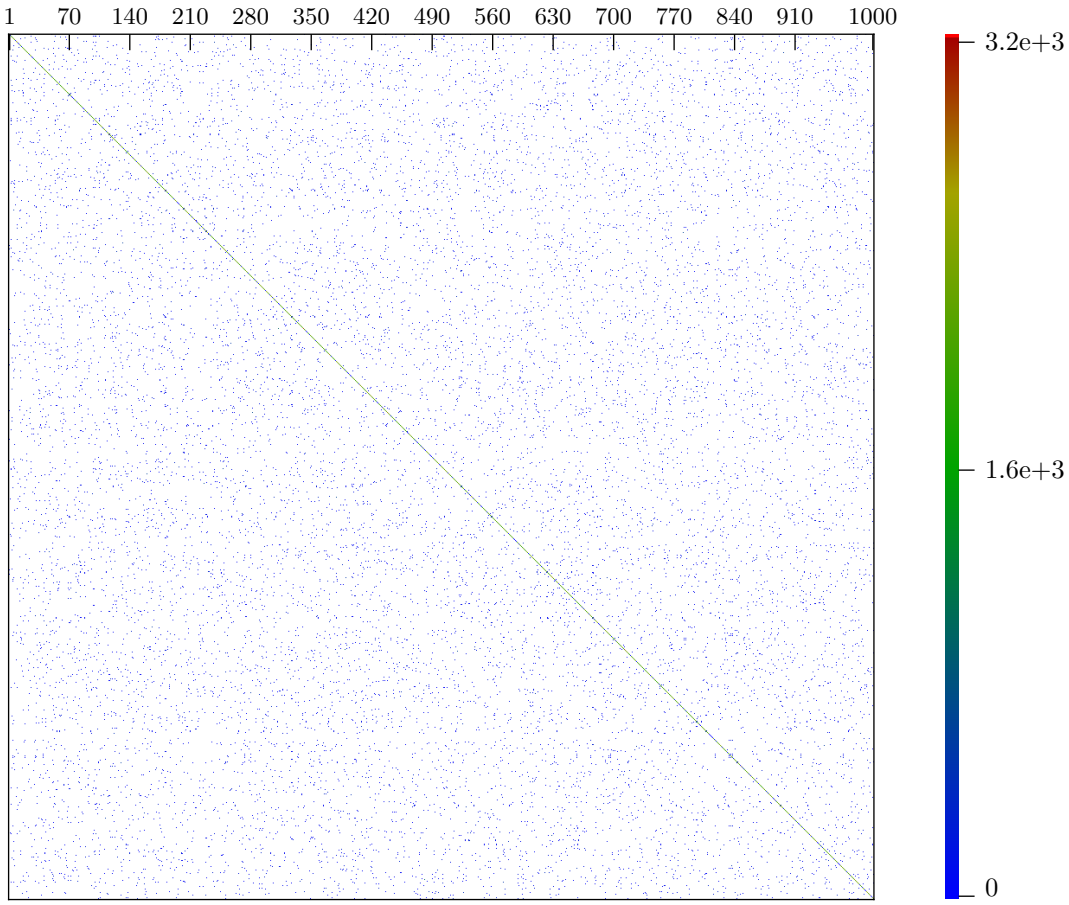


Figure 1.3. Spy Plot of `LSQ_2D_1000`

Table 1.1. Summary Information on the Test Matrices

Matrix	n	nz	Condest	Area	Group
CAVITY16	4 562	137 887	$1.39 \cdot 10^{+7}$	CFD	DRIVCAV
CAVITY26	4 562	138 040	$1.19 \cdot 10^{+8}$	CFD	DRIVCAV
EX19	12 005	259 577	$1.34 \cdot 10^{+13}$	CFD	FIDAP
EX35	19 716	227 872	$8.76 \cdot 10^{+12}$	CFD	FIDAP
GARON1	3 175	84 723	$1.46 \cdot 10^{+7}$	CFD	Garon
GARON2	13 535	373 235	$9.54 \cdot 10^{+7}$	CFD	Garon
RAEFSKY2	3 242	293 551	$1.08 \cdot 10^{+4}$	CFD	Simon
RAEFSKY3	21 200	1 488 768	$4.53 \cdot 10^{+11}$	CFD	Simon
SHYY41	4 720	20 042	$3.51 \cdot 10^{+48}$	CFD	Shyy
SHYY161	76 480	329 762	$8.23 \cdot 10^{+277}$	CFD	Shyy
IGBT3	10 938	130 500	$4.74 \cdot 10^{+19}$	SDS	Schenk ISEI
NMOS3	18 588	237 130	$1.09 \cdot 10^{+21}$	SDS	Schenk ISEI
BARRIER2-1	113 076	2 129 496	–	SDS	Schenk ISEI
PARA-4	153 226	2 930 882	–	SDS	Schenk ISEI
PARA-8	155 924	2 094 873	–	SDS	Schenk ISEI
OHNE2	181 343	6 869 939	–	SDS	Schenk ISEI
2D_54019_HIGHK	54 019	486 129	$7.55 \cdot 10^{32}$	SDS	Schenk IBMSDS
3D_51448_3D	51 448	537 038	–	SDS	Schenk IBMSDS
IBM_MATRIX_2	51 448	537 038	–	SDS	Schenk IBMSDS
MATRIX_9	103 430	1 205 518	–	SDS	Schenk IBMSDS
MATRIX-NEW_3	125 329	893 984	–	SDS	Schenk IBMSDS
LSQ_2d_1000	1000	15 803	$5.60 \cdot 10^{+4}$		
LSQ_2d_2000	2000	33 716	$1.50 \cdot 10^{+5}$		
LSQ_2d_5000	5000	91 339	$8.03 \cdot 10^{+5}$		
LSQ_2d_10000	10 000	187 949	$2.00 \cdot 10^{+6}$		
LSQ_2d_50000	50 000	978 621	$2.71 \cdot 10^{+7}$		
LSQ_2d_100000	100 000	1 987 201	$8.71 \cdot 10^{+7}$		
LSQ_2d_200000	200 000	4 010 198	$2.08 \cdot 10^{+8}$		
LSQ_2d_400000	400 000	8 144 136	–		
MPS_2d_10000	10 000	56 915	$9.44 \cdot 10^{+6}$		
MPS_2d_50000	50 000	293 015	$1.39 \cdot 10^{+8}$		
MPS_2d_100000	100 000	590 210	$3.90 \cdot 10^{+8}$		
MPS_2d_200000	200 000	1 186 000	$1.13 \cdot 10^{+9}$		
MPS_2d_400000	400 000	2 380 500	$3.01 \cdot 10^{+9}$		
LSQ_3d_10000	10 000	309 153	$1.66 \cdot 10^{+4}$		
LSQ_3d_50000	50 000	1 908 691	–		
LSQ_3d_100000	100 000	4 138 471	–		
LSQ_3d_200000	200 000	8 691 582	–		
MPS_3d_10000	10 000	66 304	$9.81 \cdot 10^{+4}$		
MPS_3d_50000	50 000	393 431	$8.11 \cdot 10^{+5}$		
MPS_3d_100000	100 000	830 863	–		
MPS_3d_200000	200 000	1 729 316	–		

Table 1.2. Direct Solve Results for the Test Matrices

Matrix	Time	Rel. Res.	Rel. Err.
CAVITY16	0.108	$1.21 \cdot 10^{-15}$	$3.59 \cdot 10^{-14}$
CAVITY26	0.094	$7.04 \cdot 10^{-16}$	$2.58 \cdot 10^{-13}$
EX19	0.141	$3.51 \cdot 10^{-16}$	$1.06 \cdot 10^{-8}$
EX35	0.21	$6.05 \cdot 10^{-16}$	$2.7 \cdot 10^{-8}$
GARON1	0.057	$7.4 \cdot 10^{-16}$	$4.99 \cdot 10^{-14}$
GARON2	0.393	$8.87 \cdot 10^{-16}$	$1.06 \cdot 10^{-13}$
RAEFSKY2	0.359	$7.34 \cdot 10^{-16}$	na
RAEFSKY3	1.17	$2.42 \cdot 10^{-16}$	na
SHYY41	0.021	$1.44 \cdot 10^{+8}$	$3.54 \cdot 10^{+26}$
SHYY161	0.716	$1.23 \cdot 10^{+236}$	$1.71 \cdot 10^{+255}$
IGBT3	0.183	$2.03 \cdot 10^{-12}$	na
NMOS3	0.45	$6.24 \cdot 10^{-15}$	na
BARRIER2-1	133.0	$1.19 \cdot 10^{-10}$	na
PARA-4	**oom**		
PARA-8	**oom**		
OHNE2	**oom**		
2D_54019_HIGHK	1.18	$3.14 \cdot 10^{-15}$	na
3D_51448_3D	6.16	$1.85 \cdot 10^{-15}$	na
IBM_MATRIX_2	6.18	$1.56 \cdot 10^{-16}$	na
MATRIX_9	56.8	$5.79 \cdot 10^{-14}$	na
MATRIX-NEW_3	11.4	$1.97 \cdot 10^{-18}$	na
LSQ_2D_1000	0.012	$7.03 \cdot 10^{-13}$	$1.13 \cdot 10^{-15}$
LSQ_2D_2000	0.024	$1.55 \cdot 10^{-12}$	$6.22 \cdot 10^{-16}$
LSQ_2D_5000	0.113	$5.18 \cdot 10^{-12}$	$1.61 \cdot 10^{-15}$
LSQ_2D_10000	0.211	$1.28 \cdot 10^{-11}$	$1.32 \cdot 10^{-15}$
LSQ_2D_50000	1.98	$9.86 \cdot 10^{-11}$	$5.49 \cdot 10^{-15}$
LSQ_2D_100000	5.45	$2.36 \cdot 10^{-10}$	$4.82 \cdot 10^{-15}$
LSQ_2D_200000	15.2	$5.43 \cdot 10^{-10}$	$1.42 \cdot 10^{-14}$
LSQ_2D_400000	43.3	$1.04 \cdot 10^{-9}$	$1.39 \cdot 10^{-14}$
MPS_2D_10000	0.077	$2.31 \cdot 10^{-11}$	$2.22 \cdot 10^{-15}$
MPS_2D_50000	0.651	$1.91 \cdot 10^{-10}$	$5.75 \cdot 10^{-15}$
MPS_2D_100000	1.68	$4.35 \cdot 10^{-10}$	$1.12 \cdot 10^{-14}$
MPS_2D_200000	4.32	$5.34 \cdot 10^{-10}$	$7.23 \cdot 10^{-15}$
MPS_2D_400000	10.7	$8.21 \cdot 10^{-10}$	$1.78 \cdot 10^{-14}$
LSQ_3D_10000	1.02	$1.79 \cdot 10^{-13}$	$3.75 \cdot 10^{-16}$
LSQ_3D_50000	55.6	$8.26 \cdot 10^{-13}$	$4.52 \cdot 10^{-16}$
LSQ_3D_100000	**oom**		
LSQ_3D_200000	**oom**		
MPS_3D_10000	0.314	$3.77 \cdot 10^{-13}$	$2.89 \cdot 10^{-16}$
MPS_3D_50000	9.54	$1.59 \cdot 10^{-12}$	$3.94 \cdot 10^{-16}$
MPS_3D_100000	45.9	$3.05 \cdot 10^{-12}$	$1.05 \cdot 10^{-15}$
MPS_3D_200000	**oom**		

CHAPTER 2

PRELIMINARIES

In this chapter we give a short overview over several topics that are touched in this thesis but are not a main focus point of this thesis: Krylov subspace methods are presented in section 2.1, preconditioning is introduced in section 2.2, several classes of matrices are covered in section 2.3, and a short introduction to graph theory is given in section 2.4.

2.1 Krylov Subspace Methods

In this section we present a very short overview of Krylov subspace methods. For a detailed introduction we refer to the literature, see, e.g., [41].

Definition 2.1: The Krylov subspace $\mathcal{K}_m(A, v)$ is defined as

$$\mathcal{K}_m(A, v) := \text{span}\{v, Av, \ldots, A^{m-1}v\}. \qquad \diamond$$

Let x_0 be an initial approximation to the solution (often $x_0 = 0$) and let $r_0 = b - Ax_0$ be the initial residual. From now on we write just $\mathcal{K}_m$ to denote $\mathcal{K}_m(A, r_0)$. The "template" Krylov subspace methods compute iterates $x_m \in x_0 + \mathcal{K}_m$ such that they satisfy the Petrov-Galerkin condition

$$b - Ax_m \perp \mathcal{L}_m,$$

where $\mathcal{L}_m$ is another subspace of dimension m. The GMRES method [42] used in the numerical experiments uses $\mathcal{L}_m = A\mathcal{K}_m$. It therefore belongs to the family of minimal residual methods, since $b - Ax_m \perp A\mathcal{K}_m$ is equivalent to minimizing the residual norm $\|b - Ax_m\|_2$ over all vectors in $x_0 + \mathcal{K}_m$, see [41, p. 133].

2.2 Preconditioning

Preconditioning describes the transformation of the original system (1.1) into an equivalent system, i.e., a system with the same solution, such that the chosen iterative method converges faster for the transformed system. As its basic operations a preconditioned Krylov subspace method usually performs in each iteration one matrix-vector multiplication with A and one application of the preconditioner, i.e., one has to solve a linear system $Mz = v$, where M is the preconditioner. A good preconditioner contains as much information about the matrix A as possible while allowing to solve $Mz = v$ with low cost. Since these two goals contradict each other in all practical cases there is no optimal preconditioner. The perfect preconditioner in the sense of containing as much information about A would be $M = A$, i.e., the transformed system (using left preconditioning, which will be explained later) would be $A^{-1}Ax = A^{-1}b$. A Krylov subspace method would need only one iteration to reach the solution, but the application of the preconditioner involves the solution of a system like (1.1). The perfect preconditioner in the sense of being inexpensive to apply would be $M = I$, the identity matrix, but the preconditioned system would be the same as the un-preconditioned one.

There are three ways of applying a preconditioner, depending on how the system (1.1) is transformed into the preconditioned system:

- We can apply the preconditioner from the left. The preconditioned system is then

$$M^{-1}Ax = M^{-1}b.$$

- We can also apply the preconditioner from the right and get

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

- Finally, we can have a *split preconditioner* $M = M_L M_R$ with the preconditioned system

$$M_L^{-1}AM_R^{-1}y = M_L^{-1}b, \quad x = M_R^{-1}y.$$

The preconditioned matrices $M^{-1}A$, AM^{-1}, and $M_L^{-1}AM_R^{-1}$ are all similar and hence have the same eigenvalues. Since for many Krylov subspace methods the convergence is determined by the eigenvalues and their distribution, in many cases the convergence will be pretty similar. This is particularly so for the conjugate gradients (CG) method, where A and M are assumed to be symmetric positive definite. However, the convergence of GMRES can be different depending on the type (left, right or split) of preconditioning used, especially if M is ill conditioned; see [41, pp. 267–272] for a comparison of left and right preconditioned GMRES.

We finally note that the residual obtained using right preconditioning in exact arithmetic, represents a residual of the original system; see, e.g,. [4, p. 420], [41, p. 270].

2.2.1 Incomplete LU Factorization Preconditioners

A widely used approach for preconditioning is based on computing an incomplete LU (ILU) factorization $LU \approx A$ of A where L and U are triangular matrices. The ILU

preconditioner based on this factorization is then $M = LU$. Computing a complete factorization $LU = A$ is usually not feasible, otherwise we could use this factorization to directly solve the linear system (1.1). A common problem is that the factors L and U of a complete LU decomposition contain too many nonzero entries. Therefore, in incomplete LU factorizations some of the nonzero entries of L and U are dropped. There are many different approaches how to do this dropping, see, e.g., [41] for a detailed description.

For the experiments in this thesis, we use MATLAB's `ilu` function to compute an incomplete LU factorization with threshold and pivoting using an ILUTP approach, see [41, p. 312–314]. For a given matrix A, it computes a unit lower triangular matrix L, an upper triangular matrix U and a permutation matrix P such that

$$LU \approx PA.$$

The dropping of small entries can be controlled by the drop tolerance (threshold) parameter *droptol*. According to the MATLAB documentation (`help ilu`), the computed factors L and U have the following properties: The off-diagonal nonzero entries of $U = (u_{ij})$, $i, j = 1, \ldots, n$ satisfy

$$|u_{ij}| \geq \mathit{droptol} \cdot \|A^{(j)}\|, \tag{2.1}$$

where $A^{(j)}$, $j = 1, \ldots, n$, denotes the jth column of A. The diagonal entries u_{jj}, $j = 1, \ldots, n$ of U are retained even if they do not satisfy (2.1). The entries of $L = (l_{ij})$, $i, j = 1, \ldots, n$, are scaled by the pivot only after they are tested against the local drop tolerance $\mathit{droptol}\|A^{(j)}\|$. Therefore, for nonzeros in L we have

$$|l_{ij}| \geq \mathit{droptol} \cdot \frac{\|A^{(j)}\|}{|u_{jj}|}.$$

2.3 Classes of Matrices

In this section we give the definitions of some classes of matrices. The definitions are fairly standard and can also be found in many books, see, e.g., [41]. For the definition of M-matrices and related classes see also [11]. In later chapters we will give some results on robustness or convergence involving spd matrices, H-matrices or M-matrices.

In the symmetric case a very import class are the positive definite matrices:

Definition 2.2: A square matrix $A \in \mathbb{R}^{n\times n}$ is called *symmetric positive definite* (*spd*) if $A = A^T$ and $x^T Ax > 0$ for all $x \neq 0$, $x \in \mathbb{R}^n$. $\diamond$

Definition 2.3: A matrix $A \in \mathbb{R}^{n\times n}$ is termed an *H-matrix* if there exist weights $u_j > 0$, $j = 1, \ldots, n$, such that for all $j = 1, \ldots, n$

$$|a_{ii}| \cdot u_i \; > \sum_{\substack{j=1,\ldots,n \\ j\neq i}} |a_{ij}| \cdot u_j. \qquad \diamond$$

If the H-matrix condition holds with weights $u_j = 1$, $j = 1, \ldots, n$, the matrix is called (strictly) *diagonally dominant*. Let A be an H-matrix and let $U = \mathrm{diag}(u_1, \ldots, u_n)$. Then AU is strictly diagonally dominant.

Definition 2.4: The *comparison matrix* $\langle A \rangle$ of A is defined by

$$\big(\langle A \rangle\big)_{ij} := \begin{cases} |a_{ii}| & \text{if } i = j \\ -|a_{ij}| & \text{otherwise.} \end{cases} \qquad \diamond$$

Definition 2.5: A square matrix $A \in \mathbb{R}^{n\times n}$ is called *Z-matrix* if $a_{ij} \leq 0$ for all $i \neq j$, $i, j = 1, \ldots, n$. A Z-matrix is called L-matrix if $a_{ii} > 0$ for all $i = 1, \ldots, n$. $\diamond$

We say that a matrix A is *nonnegative* (*positive*) if its entries are nonnegative (positive), i.e., if $a_{ij} \geq 0$ (> 0) for all $i, j = 1, \ldots, n$. We denote this by $A \geq 0$ ($A > 0$). This notation is used in the same way for vectors.

Definition 2.6: A nonsingular Z-matrix $A \in \mathbb{R}^{n\times n}$ is called an *M-matrix* if its inverse is nonnegative, i.e., if $A^{-1} \geq 0$. $\diamond$

Lemma 2.7: Every M-matrix A is an L-matrix, i.e., every M-matrix has strictly positive diagonal entries.

Proof: Let $C = A^{-1}$. Since $AC = I$ we get $\sum_{j=1,\ldots,n} a_{ij}c_{ji} = 1$. Then

$$a_{ii}c_{ii} = 1 - \sum_{\substack{j=1,\ldots,n \\ j\neq i}} a_{ij}c_{ji}.$$

Since $a_{ij}c_{ji} \leq 0$ the right hand side is at least 1 and hence $a_{ii} > 0$. $\square$

Theorem 2.8: The class of H-matrices contains the class of M-matrices, i.e., every M-matrix is an H-matrix.

Proof: Let $A \in \mathbb{R}^{n\times n}$ be an M-matrix. Let $e = (1, \ldots, 1)^T \in \mathbb{R}^n$ be the vector of ones and let $u = A^{-1}e$. Since A^{-1} is nonnegative so is u and by A^{-1} being nonsingular we can even conclude that $u > 0$, i.e., $u_j > 0$, $j = 1, \ldots, n$. We will show that these u_js are suitable weights for the H-matrix condition. We begin by observing that

$$Au = AA^{-1}e = Ie = e > 0.$$

It follows that $\sum_{j=1,\ldots,n} a_{ij}u_j > 0$ for all $i = 1, \ldots, n$. Since A is an M-matrix we know that $|a_{ii}| = a_{ii}$ and $|a_{ij}| = -a_{ij}$ for $i \neq j$. Therefore,

$$\sum_{j=1,\ldots,n} a_{ij}u_j > 0 \;\Leftrightarrow\; a_{ii}u_i > -\sum_{\substack{j=1,\ldots,n \\ j\neq i}} a_{ij}u_j \;\Leftrightarrow\; |a_{ii}|u_i > \sum_{\substack{j=1,\ldots,n \\ j\neq i}} |a_{ij}|u_j,$$

which shows that A is an H-matrix. $\square$

2.4 Graph Theory Concepts

In this chapter we introduce some basic graph theory concepts and notation used throughout this thesis. For the reader familiar with graph theory we note that the

definitions of adjacency and incidence are somewhat nonstandard for the case of directed graphs. Furthermore, the definition of a bipartite graph was chosen to simplify the definition of the bipartite graph of a matrix.

2.4.1 Directed and Undirected Graphs

Definition 2.9: An *undirected graph* or simply *graph* is an ordered pair (V, E) of sets, where E contains two-element subsets of V. The elements of V are called *vertices* or *nodes*. The elements of E are called *edges* of the graph. The elements of an edge are the *endpoints* of this edge. Edges are said to *connect* their endpoints. $\diamond$

Example 2.10: Let $G = (V, E)$ be the graph with

$$V = \{1, 2, 3, 4, 5\}$$

$$\text{and} \qquad E = \big\{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\big\}.$$

Figure 2.1 shows two different ways to draw G.

Figure 2.1. Two Ways to Draw the Example Graph G

Definition 2.11: Let $A \in \mathbb{R}^{n\times n}$, $A = (a_{ij})$ be a symmetric matrix. The undirected graph $\mathcal{G}(A) = (V, E)$ of A is given by $V = \{1, \ldots, n\}$ and

$$E = \big\{\{i, j\} \;\big|\; a_{ij} = a_{ji} \neq 0 \text{ and } i \neq j\big\},$$

i.e., the edges of $\mathcal{G}(A)$ correspond to the off-diagonal nonzero entries of A. $\diamond$

Definition 2.12: A *directed graph* or *digraph* is an ordered pair (V, E), where E is a set of pairs of elements of V. The terms *vertex*, *node*, *edge* and *endpoints* are used as for undirected graphs. $\diamond$

The edges in a directed graph have a direction. An edge (v, w) is said to go from v to w. Moreover, (v, w) and (w, v) are two different edges. In an undirected graph $\{v, w\}$ and $\{w, v\}$ are the same edge.

In this thesis we mainly consider directed graphs. Therefore, for this thesis a general graph which is not explicitly specified as being directed or undirected is considered to be directed.

In a fashion similar to definition of the undirected graph of a symmetric matrix we can define the directed graph of a general matrix:

Definition 2.13: The directed graph or digraph $\mathcal{G}(A) = (V, E)$ associated with $A \in \mathbb{R}^{n\times n}$ is given by $V = \{1, \ldots, n\}$ and $E = \big\{(i, j) \;\big|\; a_{ij} \neq 0 \text{ and } i \neq j\big\}$. $\diamond$

For incidence between vertices and edges in a digraph we use a somewhat nonstandard terminology. The general definition for both directed and undirected graphs is the following:

Definition 2.14: Let $\mathcal{G} = (V, E)$ be a graph (directed or undirected). A vertex $v \in V$ and an edge $e \in E$ are *incident* if v is one of the endpoints of e. The set of edges incident to $v \in V$ is denoted by $\mathrm{inc}(v) \subset E$. $\diamond$

For directed graphs this definition of incidence is nonstandard, because we do not distinguish between the starting point and the end point of an edge.

Definition 2.15: Let $\mathcal{G} = (V, E)$ be a graph (directed or undirected). Two vertices $v, w \in V$ are called *adjacent* if there is a connecting edge between them. In a directed graph the vertices v and w are adjacent if $(v, w) \in E$ or $(w, v) \in E$.

Let $S \subseteq V$ be a set of vertices. The *adjacency set* $\mathrm{adj}(S)$ contains all vertices which are not in S but which are adjacent to a vertex in S; i.e.,

$$\mathrm{adj}(S) = \big\{ j \in V \;\big|\; j \notin S \text{ and } j \text{ is adjacent to some } i \in S \big\}. \qquad \diamond$$

A commonly used concept in this thesis is to obtain subgraphs by restricting a graph to a subset of the vertex set.

Definition 2.16: Let $\mathcal{G} = (V, E)$ be a digraph and let $V' \subseteq V$ be a vertex set. The *induced subdigraph* $\mathcal{G}|_{V'} = (V', E')$ is defined by $E' = \big\{ (i,j) \in E \;\big|\; i, j \in V' \big\}$.

The induced subgraph $\mathcal{G}|_{V'} = (V', E')$ of an undirected graph $\mathcal{G} = (V, E)$ is defined by $E' = \big\{ \{i,j\} \in E \;\big|\; i, j \in V' \big\}$. $\diamond$

Throughout this thesis we will use the restriction notation $|_{V'}$ to denote quantities of an induced subgraph, e.g., $\mathrm{adj}|_{V'}(S)$ refers to the adjacency set of S in the subgraph induced by V'. Moreover, we write $|E|_{V'}$ to denote the cardinality of the edge set E' of the induced subgraph $\mathcal{G}|_{V'} = (V', E')$.

We will also use a generalization of incidence.

Definition 2.17: Let $\mathcal{G} = (V, E)$ be a digraph and let $S \subset V$ be a node set. We define $\mathrm{inc}(S) \subset E$ to be the set of edges in E which are incident to some node in S, i.e.,

$$\begin{aligned} \mathrm{inc}(S) &= \bigcup_{v \in S} \{ e \in E \,|\, e \text{ incident to } v \} \\ &= \text{edges in } \mathcal{G}|_{S \cup \mathrm{adj}(S)}. \end{aligned}$$

The set $\mathrm{inc}(S)$ is called the set of edges incident to S.

For two sets $S, T \subset V$ we define

$$\mathrm{inc}(S,T) := \mathrm{inc}(S) \cap \mathrm{inc}(T)$$

to be the set of edges incident to both S and T. $\diamond$

Definition 2.18: Let $\mathcal{G} = (V, E)$ be an undirected graph. The *degree* of $v \in V$ is the number of edges incident to v, i.e.,

$$\deg(v) := \Big|\big\{\{v, w\} \in E\big\}\Big|.$$

For a digraph $\mathcal{G} = (V, E)$ we use three different degrees:

$$\deg_{\text{in}}(v) := \Big|\big\{(w, v) \in E\big\}\Big|,$$

$$\deg_{\text{out}}(v) := \Big|\big\{(v, w) \in E\big\}\Big|,$$

$$\text{and} \qquad \deg(v) := \deg_{\text{in}}(v) + \deg_{\text{out}}(v).$$

If $V' \subseteq V$, we write, in a slight abuse of the restriction notation, $\deg|_{V'}(i)$ to denote $\deg(i)$ in the graph induced by $V' \cup \{i\}$. $\diamond$

Remark 2.19: With our definition of *incident* the degree of a vertex is always the number of incident edges, both in the undirected and the directed case, i.e., an equivalent definition of $\deg(i)$ is

$$\deg(i) := \Big|\{e \in E \,|\, i \text{ is incident with } e\}\Big|.$$

Lemma 2.20: If $\mathcal{G}(A) = (V, E)$ is the digraph of a matrix A with $\operatorname{nnz}(A)$ non-zero elements, then

$$\sum_{q \in V} \deg(q) \leq 2 \operatorname{nnz}(A).$$

Proof: The number of edges in $\mathcal{G}(A)$ is less than or equal to $\operatorname{nnz}(A)$. Each edge in $\mathcal{G}$ is incident to exactly two vertices and is therefore counted exactly two times

when we add together all degrees. This proves Lemma 2.20. Note that we reach $\sum_{q\in V} \deg(q) = 2\,\mathrm{nnz}(A)$, if the diagonal of A contains only zeros. □

Definition 2.21: Let $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$ be digraphs. $\mathcal{G}$ and $\mathcal{G}'$ are *isomorphic*, written $\mathcal{G} \cong \mathcal{G}'$, if there is a bijection $f : V \to V'$ such that $(v_i, v_j) \in V$ if and only if $(f(v_i), f(v_j)) \in E'$. We write $f((v_i, v_j))$ to denote the edge $(f(v_i), f(v_j))$.

Two undirected graphs $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$ are *isomorphic* if there is a bijection $f : V \to V'$ such that $\{v_i, v_j\} \in E$ if and only if $\{f(v_i), f(v_j)\} \in E'$. We write $f(\{v_i, v_j\})$ to denote the edge $\{f(v_i), f(v_j)\} \in E'$. ◇

2.4.2 Partitions, Covers and Permutations

Definition 2.22: A family $\mathcal{V} = \{V_i\}_{i=1,\ldots,q}$ of ordered nonempty subsets $V_i \subset V$ is called a *cover* of V if $\bigcup_{i=1}^{q} = V$. A cover is called a *partition* if the sets V_i are pairwise disjoint, i.e., if $V_i \cap V_j = \emptyset$ for $i \neq j$. Let $n_i = |V_i|$. We denote the $n_i = |V_i|$ elements of V_i by $v_1^{(i)}, \ldots, v_{n_i}^{(i)}$. ◇

Note that the node sets $V_1 = \{1, 2, 3\}$ and $V_2 = \{3, 1, 2\}$ are the same sets, but they do not have the same order. We will explicitly speak of ordered sets if the order is important, see, e.g., Example 4.2 in section 4.1. If the order does not make a difference we will just speak of sets.

A partition can be used to define a permutation of the elements of V:

Definition 2.23: Let $\mathcal{V}_q = \{V_1, \ldots, V_q\}$ be a partition of $V = \{1, \ldots, n\}$ with $n_i = |V_i|$. The permutation function $\pi : V \to V$ with respect to the partition $\mathcal{V}_q$ is defined as $\pi(1) = v_1^{(1)}$, …, $\pi(n_1) = v_{n_1}^{(1)}$, $\pi(n_1 + 1) = v_1^{(2)}$, …, $\pi(n) = v_{n_q}^{(q)}$. ◇

The permutation with respect to a partition can also be understood as a map $\pi : V \to V$ that groups the vertices $1, \ldots, n$, from V into the sets V_1, …, V_q, i.e., $V_1 = \{\pi(1), \ldots, \pi(k_1)\}$, $V_2 = \{\pi(k_1{+}1), \ldots, \pi(k_2)\}$, …, $V_q = \{\pi(k_{q-1}{+}1), \ldots, \pi(k_q)\}$, with $k_i = \sum_{j=1}^{i} n_i$.

2.4.3 Bipartite Graphs

Definition 2.24: A *bipartite graph* $\mathcal{B} = (V_r, V_c, E)$ consists of two disjoint sets of vertices (nodes), namely V_r and V_c, and a set of edges

$$E \subset \big\{\{i,j\} \;\big|\; i \in V_r,\, j \in V_c\big\}$$

connecting the vertices. The vertices in V_r are called *row vertices* (or *row nodes*) and the vertices in V_c are called *column vertices* (or *column nodes*). $\diamond$

Definition 2.25: For a node set $S \subset V_r \cup V_c$ the *induced subgraph* is defined to be the bipartite graph $\mathcal{B}' = (V_r \cap S, V_c \cap S, E')$ where

$$E' = \big\{e = \{v,w\} \in E \;\big|\; v, w \in S\big\}. \qquad \diamond$$

Definition 2.26: Let $\mathcal{B} = (V_r, V_c, E)$ be a bipartite graph with $V = V_r \cup V_c$, $m = |V_r|$, and $n = |V_c|$. The undirected graph $\mathcal{G}(V, E)$ is called the undirected graph associated with $\mathcal{B}$. $\diamond$

Definition 2.27: Let $A \in \mathbb{R}^{m\times n}$, $A = (a_{ij})$, $i = 1, \ldots, m$, $j = 1, \ldots, n$ be a matrix. The *bipartite graph* $\mathcal{B}(A) = (V_r, V_c, E)$ of A consists of vertices $V_r = \{r_1, \ldots, r_m\}$, $V_c = \{c_1, \ldots, c_n\}$ and edges

$$E = \big\{\{r_i, c_j\} \;\big|\; a_{i,j} \neq 0\big\}. \qquad \diamond$$

Remark 2.28: Let the bijection $f : V_r \cup V_c \to \{1, \ldots, m+n\}$ be defined by $f(r_1) = 1, \ldots, f(r_m) = m$ and $f(c_1) = m+1, \ldots, f(c_n) = m+n$. The undirected graph $\mathcal{G} = (V, E)$ associated with $\mathcal{B}(A)$ is isomorphic to the undirected graph $\tilde{\mathcal{G}} = \mathcal{G}(\tilde{A})$ of the symmetric matrix

$$\tilde{A} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \in \mathbb{R}^{(m+n)\times(m+n)},$$

i.e., $\mathcal{G} \cong_f \tilde{\mathcal{G}}$.

CHAPTER 3

PABLO AND ITS VARIANTS

The original PABLO (parameterized block ordering) and TPABLO (threshold parameterized block ordering) algorithms are collections of algorithms which compute a symmetric permutation P of a linear system $Ax = b$ such that the permuted system

$$\widehat{A}\widehat{x} = \widehat{b}, \quad \text{where} \quad \widehat{A} = PAP^T,\ \widehat{x} = Px,\ \text{and}\ \widehat{b} = Pb, \tag{3.1}$$

has a relatively full block diagonal with relatively large nonzero entries. Block sizes are determined dynamically; see [5], [14], [15], [36].

The permutation found by PABLO or TPABLO can be used to determine a preconditioner. In the simplest case the preconditioner is taken as the block diagonal $\widehat{D}$ using the PABLO or TPABLO blocks. One then solves the permuted system (3.1) using GMRES (or some other Krylov subspace method) with preconditioner $\widehat{D}$; see, e.g., [33], [41], [46] for their description. To solve the preconditioning system $\widehat{D}s = r$, a (complete) LU decomposition $\widehat{D} = \widehat{L}\,\widehat{U}$ is computed. Since this reduces to an LU decomposition on each of the diagonal blocks, we can use readily available packages. In our experiments we use UMFPACK [18] to compute the sparse LU decomposition on each diagonal block. Moreover, the LU decomposition of $\widehat{D}$ can be done in parallel.

While PABLO determines the blocks for $\widehat{D}$ by just using structural information of A given through its associated directed graph, the TPABLO variants also take the

size of entries in A into account, i.e., it works on the weighted directed graph of A with edge weights corresponding to the magnitude of the matrix entries.

In this chapter, which builds upon [29] and [30], we propose and analyze three extensions of the PABLO and TPABLO preconditioners. First, we apply a nonsymmetric permutation to put large entries on the diagonal. We discuss this in section 3.2. In section 3.3 we introduce new parametrizations to be used with PABLO and TPABLO. We describe the resulting algorithmic framework, termed XPABLO, in detail and analyze its computational complexity. XPABLO is more general than PABLO and TPABLO, since it includes new criteria to produce the blocking permutation, and it reduces to the previous versions for specific choices of its parameters. In section 3.6 we discuss how one can use either the lower or the upper block triangular part of the matrix as a preconditioner at the same cost as the block diagonal preconditioner.

After reviewing several practical issues related to the implementation in section 3.5, we finally give results of numerical experiments in section 3.7, including a comparison with ILU preconditioning. In section 3.8, we present some concluding remarks.

A short survey of the literature about reorderings for preconditioners is given in section 3.1.

We conclude this introduction to the chapter by noting that PABLO variants have been applied successfully in the construction of preconditioners in settings that are different from the one presented in this thesis. A slightly modified version of PABLO has been used successfully in a parallel computing setting to build block diagonal preconditioners for difficult problems in computational fluid dynamics (CFD) and chemical engineering; see, e.g., [26]. The threshold variants TPABLO and XPABLO can be used as a method to find blocks with large entries in dense or sparse matrices.

Specifically, in [28], XPABLO is used to find such blocks to build preconditioners for the CSYM method [13] for solving systems with complex symmetric matrices. The situation there is that an efficient preconditioner has to be factorized into its complex symmetric singular value decomposition, so that block diagonal preconditioners seem to be the only practical way. In the examples presented in [28] the iteration count is reduced by up to 30% when using this approach.

3.1 Some Notes on Literature on Reorderings for Preconditioners

In addition to the references already mentioned, several authors have explored reorderings and partitioning techniques to improve the robustness and performance of different types of preconditioners.

The combination of reorderings and scalings with ILU preconditioners for solving highly indefinite and nonsymmetric systems is studied in [7]. The numerical experiments show that the reliability and performance of ILU preconditioned Krylov subspace methods can be drastically improved by applying nonsymmetric permutations to put large entries on the diagonal. The best results came from the maximum product transversal algorithm with scaling (MPS), the same algorithm we recommend and use as a preparation step in section 3.2. The experiments also show the usefulness of additionally using a symmetric reordering. It was found in [7] that the reverse Cuthill–McKee ordering (RCM) [21] gave good results in a majority of cases and was therefore chosen as the default ordering.

Symmetric reorderings for ILU preconditioners for symmetric or nearly symmetric matrices with zero-free diagonal are studied in [9]. The experiments in the paper cover the reorderings found by the Cuthill–McKee algorithm, the reverse Cuthill–McKee algorithm and the multiple minimum degree algorithm [32]. Although all these

algorithms were designed for sparse direct solvers, they are found to be useful to improve the performance of iterative (Krylov subspace) methods preconditioned with incomplete LU factorizations. In many cases RCM was found to be best and in many cases for which RCM was not best, it still gave good results. In this respect the results in [9] are consistent with those from [7].

The influence of unsymmetric permutations on solving linear systems originating from semiconductor device simulation and from circuit simulation is studied in [43]. The numerical results in [43] present both the effect on direct solvers and on various preconditioners for Krylov subspace solvers. The authors conclude that a maximum product transversal algorithm with scaling gives the best results and they especially point out the significant (positive) impact on ILU preconditioning. The combination of MPS and ILU was found to be on average the most efficient preconditioner for the test problems, a result very similar to the one in [7].

Orderings for factorized sparse approximate inverse preconditioners (AINV) preconditioners [41, p. 331–333] are studied in [10]. The numerical results show that for these preconditioners minimum degree is best followed by nested dissection, while reverse Cuthill–McKee and red-black orderings perform poorly. This shows, that the effect of reorderings on the performance of AINV is very different, almost opposite, to the effect on the performance of ILU.

A good overview on reordering for ILU preconditioning is also given in [41, p. 333–337]. In the example results from [41] RCM performs best of the three compared reordering techniques: RCM, minimum degree, and nested dissection.

For an overview–without numerical results–of the established algorithms in the field of reordering strategies for preconditioning we cite also the appropriate sections in the survey [4] and the technical report [25].

3.2 Nonsymmetric Permutations and Diagonal Scalings

The rationale for obtaining diagonals with large weight is heuristic: Large diagonals tend to decrease the need for pivoting in a direct elimination method. ILU-type preconditioners can also benefit greatly [7], [43]. Moreover, an iterative method with diagonal preconditioner may be expected to converge more rapidly if the diagonal is large compared to the off-diagonal part of the matrix.

Let $\mathcal{B} = (V_r, V_c, E)$ be the bipartite graph of A. An ordered subset $T \subseteq E$ is called a *matching* or *transversal* if no two edges in T are incident to the same node, i.e., T consist of edges $\{i, j\}$, $i \in \{r_1, \ldots, r_n\}$, $j \in \{c_1, \ldots, c_n\}$, such that each row node i and each column node j appears at most once. A transversal T is a *maximum transversal* (or maximum matching) if it has maximum cardinality, i.e., $|T| \geq |T'|$ for all possible transversals T'. If $A \in \mathbb{R}^{n\times n}$ is not structurally singular, then a maximum transversal T has cardinality $|T| = n$; see, e.g., [21]. In this case, T is called a *perfect matching* and defines an $n \times n$ permutation matrix Σ with

$$(\Sigma)_{ij} = \begin{cases} 1 & \text{if } \{i, j\} \in T, \\ 0 & \text{otherwise.} \end{cases}$$

Let σ be the permutation associated with Σ, i.e., for all $x \in \mathbb{R}^n$

$$(\Sigma x)_i = x_{\sigma(i)}, \quad i = 1, \ldots, n.$$

As a consequence, for a maximum transversal of a structurally nonsingular matrix $A \in \mathbb{R}^{n\times n}$, we have

$$(\Sigma A)_{ii} = a_{\sigma(i),i} \neq 0, \quad i = 1, \ldots, n.$$

In [22], two different types of transversals with respect to different notions of a "weight" were introduced according to the following definition.

Definition 3.1: A maximum transversal T with corresponding permutation σ

(i) is a *bottleneck transversal* if it maximizes

$$\min_{i=1,\ldots,n} |a_{\sigma(i),i}|$$

over all maximum transversals, and

(ii) it is a *maximum product transversal* if it maximizes

$$\prod_{i=1}^{n} |a_{\sigma(i),i}|$$

over all maximum transversals. $\diamond$

The paper [22] proposes and analyzes various algorithms to compute bottleneck transversals. It is reported there that extensive testing shows that in practice the computational complexity of these algorithms behaves like $\mathcal{O}(n + \text{nnz})$, although the theoretical worst case upper bound is $\mathcal{O}(n \cdot \text{nnz})$.

An algorithm for computing maximum product transversals is given in [23]. It has been incorporated in the Harwell Subroutine Library (HSL) [40] as algorithm MC64. As a by-product, when computing a maximum product transversal, MC64 also delivers diagonal scaling matrices $\hat{C}$ and $\hat{R}$ such that

$$\hat{A} = \Sigma(\hat{R}A\hat{C}) \tag{3.2}$$

is an I-matrix in the sense of [35]; i.e., $|\hat{A}_{ij}| \leq 1$ for all i, j and $\hat{A}_{ii} = 1$, $i = 1, \ldots, n$.

The time complexity bound for computing a maximum product transversal as implemented in MC64 is $\mathcal{O}(n(\text{nnz} + n) \log_2 n)$. The tests in [23, p. 987] show that finding a maximum product transversal is in almost all cases computationally more expensive than finding a bottleneck transversal. Further tests show that in many

cases this is compensated by $\hat{A}$ being easier to factorize or being better suited for iterative solvers.

In this chapter, we use maximum product transversals and we always apply the scaling computed by MC64; i.e., we always transform A into an I-matrix before applying XPABLO. For our test problems in section 3.7 we found this to be always superior to using the cheaper bottleneck transversal.

3.3 XPABLO: An Extension of PABLO and TPABLO

For a digraph $\mathcal{G} = (V, E)$, PABLO, TPABLO, and XPABLO produce a partition of V into q disjoint, nonempty subsets V_κ, $\kappa = 1, \ldots, q$ (the "blocks"). These blocks are built one at a time. We now describe how this is done, assuming the following situation: Blocks $V_1, \ldots, V_{\nu-1}$ have already been built. Given the current block V_ν and a candidate vertex $i \in \operatorname{adj}|_{\overline{V}}(V_\nu)$ with $\overline{V} = V \setminus \bigcup_{\kappa=1}^{\nu-1} V_\kappa$, the algorithms decide whether or not to incorporate i into the current block using a decision function (usually called criterion) τ. After finishing a block its nodes are removed from the graph. XPABLO uses three disjoint sets C, Q and B to hold the nodes still in the graph. The set B denotes the current block being built, i.e., the elements of B will become the νth block. The set $Q \subset \operatorname{adj}|_{\overline{V}}(B)$ contains the current candidate vertices, i.e., a candidate vertex i will always come from Q. Finally, the set C contains all the remaining vertices. Algorithm 3.1 (p. 35) and Algorithm 3.2 (p. 40) show in detail how candidate vertices are selected; see also [36].

3.3.1 XPABLO Criteria and Parameters

The following definitions describe some basic criteria. They can be combined logically to yield a variety of different criteria, including, as we shall see in section 3.3.2, the traditional PABLO and TPABLO criteria.

Definition 3.2: Let $\mathcal{G} = (V, E)$ be a digraph, $V' \subseteq V$, and $\mathcal{G}' = (V', E')$ be the subgraph induced by V'. The *fullness* $\phi(V')$ of V' is defined as

$$\phi(V') = \begin{cases} \dfrac{|E'|}{|V'|^2 - |V'|} & \text{if } |V'| > 1, \\ 0 & \text{if } |V'| \leq 1. \end{cases} \qquad \diamond$$

The thus defined fullness measures the number of edges in $\mathcal{G}'$ compared to the maximally possible number $|V'|^2 - |V'|$ for a complete digraph. The definition of $\phi(\{v\}) = 0$ follows PABLO in the interpretation that a graph with only one vertex is empty, but other definitions may be useful depending on the problems one needs to solve.

Given $\alpha > 0$, in our generic situation, we say (see [36]) that vertex i satisfies the *fullness criterion* (with fullness parameter α) if

$$\phi\big(V_\nu \cup \{i\}\big) \geq \alpha\, \phi(V_\nu). \tag{FC}$$

Note that (FC) can be fulfilled even when $\alpha > 1$.

Definition 3.3: If i is a vertex in a digraph $\mathcal{G} = (V, E)$ with $\deg(i) > 0$, and if $V' \subseteq V$, the *connectivity* of i with respect to V' is the fraction

$$\frac{\deg|_{V'}(i)}{\deg(i)}. \qquad \diamond$$

In our generic situation, given $\beta > 0$, we say that vertex i satisfies the *connectivity criterion* (with connectivity parameter β) if

$$\deg|_{V_\nu}(i) \geq \beta \deg|_{\overline{V}}(i). \tag{CC}$$

The connectivity criterion means that in $\mathcal{G}|_{\overline{V}}$ at least a fraction β of all edges incident with i have their other incident vertex in V_ν. The criterion is never met if $\beta > 1$ (and $\deg(i) > 0$).

Example 3.4: Figure 3.1 illustrates the fullness and the connectivity criterion for the digraph $\mathcal{G}(A)$ in terms of the pattern of the matrix A. The matrix is assumed to be already symmetrically permuted such that the three blocks built up so far appear first. The black diagonal entry corresponds to the candidate vertex which may end up to be included into the third block. The fullness criterion requires that the grey parts of row i and column i must not be too sparse. The connectivity criterion requires that the hatched parts of these rows and columns should not contain too many elements as compared to the grey parts. Row and column i are considered only together—not individually. The white parts of row i and column i are not taken into consideration when deciding on vertex i.

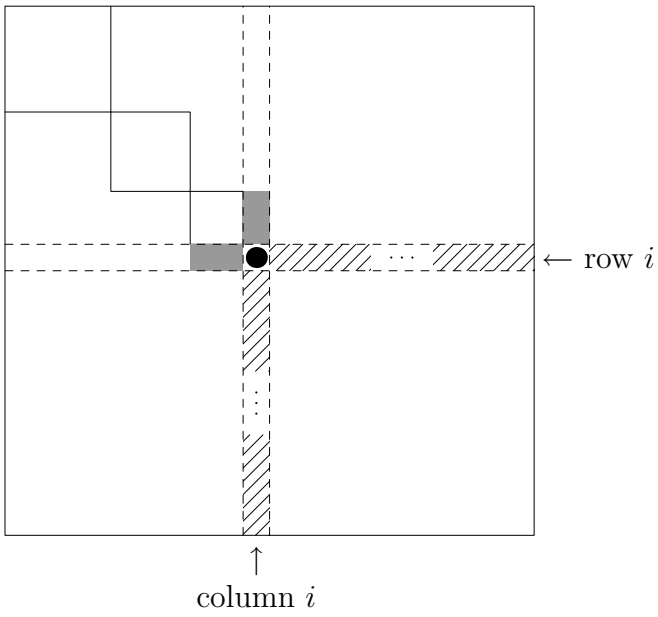


Figure 3.1. Illustration of the Fullness Criterion and the Connectivity Criterion

The PABLO algorithm of [36] adds a candidate vertex $i \in \text{adj}\,|_{\overline{V}}(V_\nu)$ to the current block V_ν if and only if v satisfies the fullness *or* the connectivity criterion for some prechosen parameters α and β. We formulate this by saying that PABLO uses the criterion τ defined as

$$\tau = \text{FC} \vee \text{CC}. \qquad \text{(PABLO)}$$

The combined PABLO criterion uses only structural information of A. In its threshold counterparts TPABLO1 and TPABLO2 (see [5], [14]), numerical values of the matrix entries are also taken into account. For a precise statement, and also to formulate our generalization XPABLO, we introduce additional notation and terminology. From now on, we consider the digraph $\mathcal{G}(A)$ to be an edge-weighted digraph where edge $e = (i, j)$ has weight $w(e) = |a_{ij}|$. An edge $e = (i, j)$ is called *large* if its weight $w(e) = |a_{ij}|$ is larger than a given threshold $\gamma > 0$, i.e., e is large if $w(e) > \gamma$. In the same manner, we call matrix entries large if their magnitude is larger than γ.

Definition 3.5: Given $A \in \mathbb{R}^{n\times n}$ and $\gamma > 0$, we write $A^{>\gamma} \in \mathbb{R}^{n\times n}$ for the matrix

$$(A^{>\gamma})_{ij} = \begin{cases} a_{ij} & \text{if } |a_{ij}| > \gamma, \\ 0 & \text{otherwise.} \end{cases} \qquad \diamond$$

We use the superscript notation $^{>\gamma}$ in an intuitive manner at various places. For example, if $\mathcal{G} = \mathcal{G}(A)$ is the digraph of A, then $\mathcal{G}^{>\gamma} = \mathcal{G}(A^{>\gamma})$. Another example is the following definition.

Definition 3.6: Let $\mathcal{G} = (V, E)$ be a digraph, $V' \subseteq V$, and $\gamma \geq 0$. The *threshold fullness* $\phi^{>\gamma}(V')$ of V' is the fullness $\phi(V')$ of V' in the graph $\mathcal{G}^{>\gamma}$. $\diamond$

In our generic situation, given a threshold parameter $\gamma > 0$ and a threshold fullness parameter $\vartheta \in [0, 1]$, we say that i satisfies the *threshold fullness criterion* if

$$\phi^{>\gamma}\big(V_\nu \cup \{i\}\big) \geq \vartheta. \tag{TFC}$$

In contrast to the plain fullness criterion, the threshold fullness criterion just measures the fullness the new block $V_\nu \cup \{i\}$ has in $A^{>\gamma}$ without relating it to the fullness of V_ν.

Definition 3.7: Let $\mathcal{G} = (V, E)$ be a digraph, $V' \subseteq V$, and $\gamma \geq 0$. If i is a vertex with $\deg|_{V'}(i) > 0$, the *threshold connectivity* of i with respect to V' is the fraction

$$\frac{\deg|^{>\gamma}_{V'}(i)}{\deg|_{V'}(i)}. \qquad \diamond$$

The threshold connectivity compares the number of large edges incident to i in $\mathcal{G}|_{V'}$ to all such edges. As opposed to plain connectivity according to Definition 3.3, it does not consider any edges going to nodes outside V'.

In our generic situation, given a threshold connectivity parameter ζ, we say that the *threshold connectivity criterion* holds if

$$\deg|_{V_\nu}^{>\gamma}(i) \geq \zeta \deg|_{V_\nu}(i). \tag{TCC}$$

This criterion measures how many of the new entries of the current block are large. An important choice for the threshold connectivity parameter is $\zeta = 1/(2n)$. For this value of ζ, the right-hand side of (TCC) is always strictly less than one since $\deg|_{V_\nu}(i) < 2n$ for all $i \in V$. Hence for $\zeta = 1/(2n)$, the threshold connectivity criterion holds if and only if $\deg|_{V_\nu}^{>\gamma}(i) \geq 1$, i.e., if and only if there is at least one edge between i and the current block V_ν with edge weight larger than γ.

In addition to the threshold parameter γ, XPABLO also accepts an additional threshold parameter δ with $0 \leq \delta < \gamma$, which is used to filter out nonzero matrix entries with very small magnitude; i.e., XPABLO really works on the graph $\mathcal{G}(A^{>\delta})$. Note that δ changes only the graph on which XPABLO is operating and not the underlying matrix.

Algorithm 3.1 shows the details of the way XPABLO works. In the loops over all edges incident with a vertex i, a vertex j adjacent to i can appear twice, as j can be adjacent to i through up to two edges. For the correctness of the updates it is actually important to have j appear twice if there are two edges between i and j.

In our implementation of XPABLO we have two additional parameters to control the size of blocks found by XPABLO. With *minbs* we can set the desired minimum blocks size and with *maxbs* we can enforce a maximum block size. In particular enforcing a maximum block size is important in many applications, as it

```
1:  input: a digraph G(A) = (V, E) and a criterion τ
2:  output: a partitioning {V_1, ..., V_q} of V
3:  C := V,   Q := ∅,   B := ∅,   q := 1
4:  while C ≠ ∅ do
5:      remove a vertex i from C, and place it in B
6:      for all edges e in inc(i) do
7:          let j be the vertex adjacent to i through e
8:          move j from C to Q if j ∈ C
9:      end for
10:     while Q ≠ ∅ do
11:         remove a vertex i from Q
12:         if i fulfills the XPABLO criterion τ then
13:             insert i into B
14:             for all edges e in inc(i) do
15:                 let j be the vertex adjacent to i through e
16:                 move j from C to Q if j ∈ C
17:             end for
18:             if |B| > maxbs then
19:                 move all nodes in Q to C
20:             end if
21:         else
22:             insert vertex i into C
23:         end if
24:     end while
25:     set V_q := B and q := q + 1        {finish current block}
26:     G := G|_C
27:     set B := ∅
28: end while
29: merge adjacent blocks of size < minbs if the size of the resulting block is ≥ maxbs
```

Algorithm 3.1. XPABLO

may otherwise happen that with the criteria and the parameters in use, the algorithm would produce only one block, the whole matrix, and this is clearly not useful. To enforce a maximum block size of *maxbs* we modify XPABLO so that it closes the current block B as soon as it reaches this maximum size, which can easily done by moving all nodes still in Q to C, see lines 18–20 in Algorithm 3.1.

We often also fix a minimum block size *minbs*. This minimum size is achieved by merging adjacent blocks which are too small. Note that we do not merge blocks if we would exceed the maximum block size, even if we retain small blocks.

Example 3.8: Figure 3.2 shows the blocks found by XPABLO for the graph of the matrix `LSQ_2d_1000`. The block size was forced to be 250 by setting $minbs = maxbs = 250$. The blocks were found by XPABLO in the following order: black, blue, red, green. Notice that the first three blocks (black, blue, and red) are connected, but the last block (green) consist of five connected components. This comes from merging five small blocks together.

Figure 3.3 shows a spy plot of the matrix after being permuted according to the blocks found by XPABLO. For comparison we refer to the non-permuted spy plot shown in Figure 1.3.

3.3.2 XPABLO as a Generalization of PABLO and TPABLO

We now have four criteria at hand upon which we can decide whether to include a new vertex to a current block or not. These criteria can be logically combined resulting in various XPABLO criteria, denoted by τ. The following list shows how XPABLO is reduced to PABLO or TPABLO by choosing a specific criterion τ and specific values for some of the parameters.

1. With $\tau = \mathrm{FC} \vee \mathrm{CC}$, XPABLO reduces to PABLO.
2. With $\tau = (\mathrm{FC} \vee \mathrm{CC}) \wedge \mathrm{TCC}$ and $\zeta = 1/(2n)$, XPABLO reduces to TPABLO1.

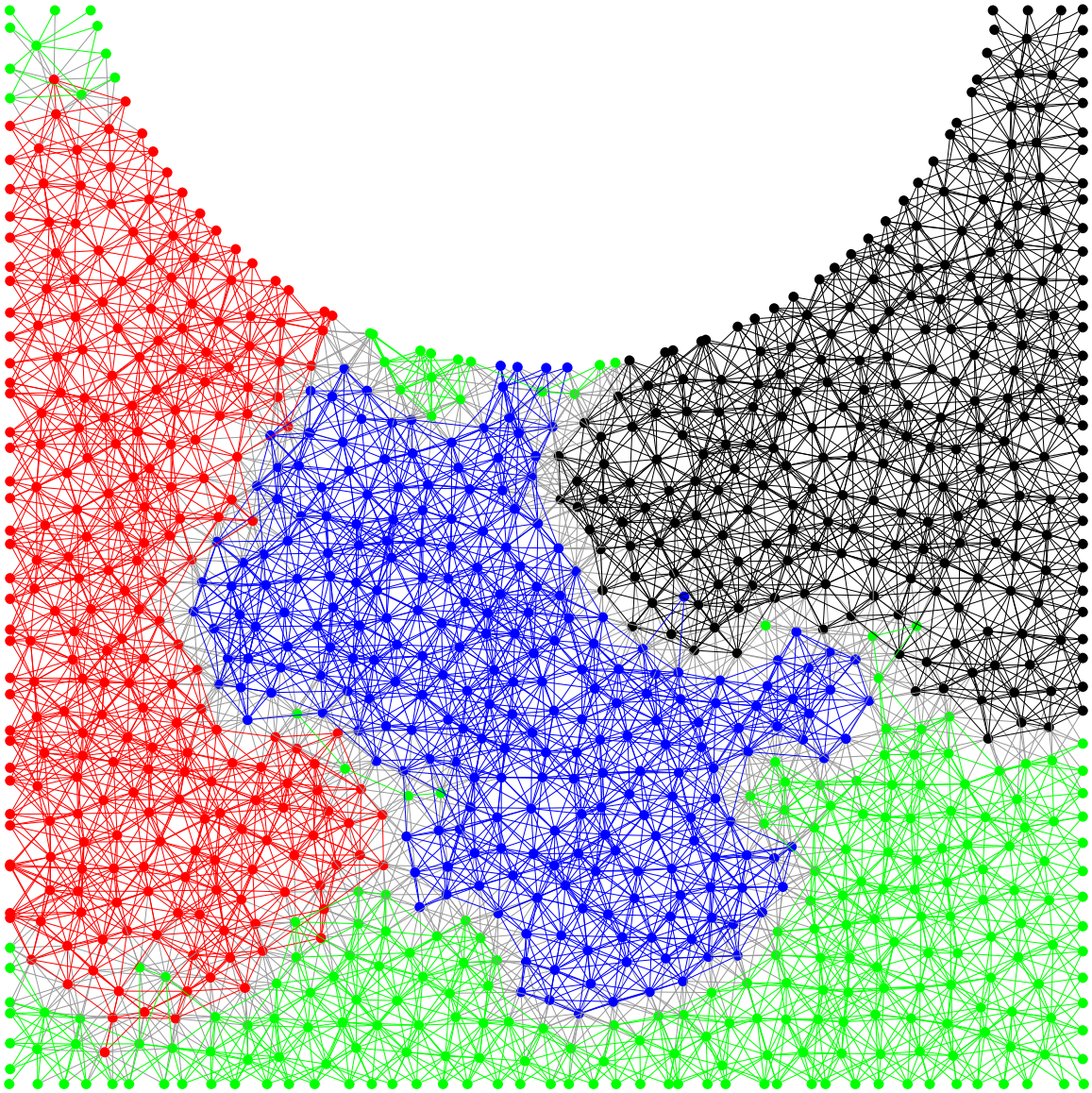


Figure 3.2. Graph of `LSQ_2D_1000` Showing XPABLO Blocks

The graph shows the four blocks found by XPABLO ($minbs = 250$, $maxbs = 250$). Edges inside a block are colored in the block color (black, blue, red, green), edges between blocks are colored in grey.

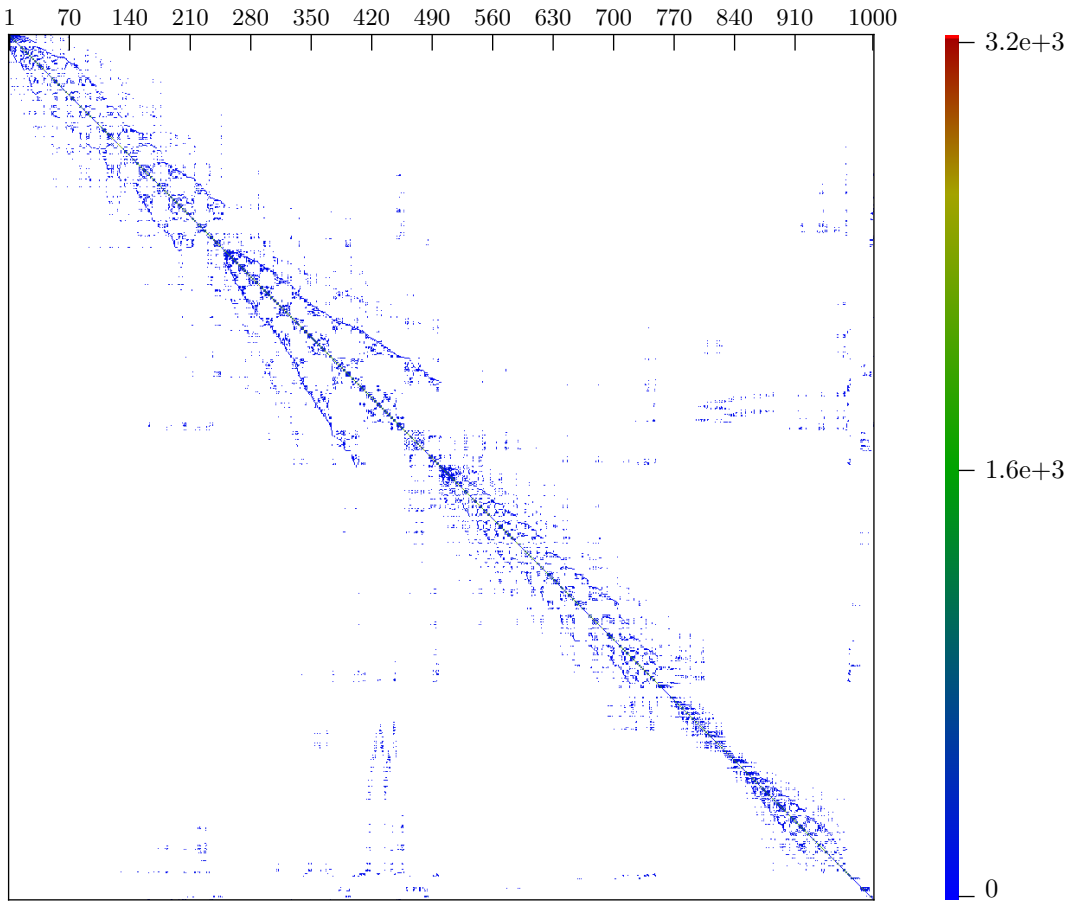


Figure 3.3. Spy Plot of `LSQ_2D_1000` After XPABLO Permutation
Compare with Figure 1.3.

3. With $\tau = (\mathrm{FC} \vee \mathrm{CC}) \wedge \mathrm{TFC}$ and $\vartheta = 1$, XPABLO reduces to TPABLO2.

Based on a series of numerical experiments, our default criterion for XPABLO is $\tau = \mathrm{FC} \vee \mathrm{CC} \vee \mathrm{TCC}$ with $\zeta = 1/(2n)$; see section 3.5.2 for a more detailed discussion of XPABLO default parameters.

Once XPABLO has determined a partitioning of V into blocks $V_1, \ldots, V_q$, let π be a permutation that groups the vertices $1, \ldots, n$, from V into these blocks; i.e., $V_1 = \{\pi(1), \ldots, \pi(k_1)\}$, $V_2 = \{\pi(k_1{+}1), \ldots, \pi(k_2)\}$, $\ldots$, $V_q = \{\pi(k_{q-1}{+}1), \ldots, \pi(k_q)\}$, with $k_i = \sum_{i=j}^{i} |V_j|$. Then the permuted matrix PAP^T is naturally partitioned into q blocks of size $|V_j|$, the jth diagonal block corresponding to the vertex set V_j. The threshold criteria may now be interpreted as placing large entries on these diagonal blocks. This is made more precise in the following proposition which follows immediately upon inspection of the various criteria.

Proposition 3.9:

1. If $\tau = (\mathrm{FC} \vee \mathrm{CC}) \vee \mathrm{TCC}$ and $\zeta = 1/(2n)$, then all entries in the off-diagonal blocks of PAP^T have modulus less than γ.
2. If $\tau = (\mathrm{FC} \vee \mathrm{CC}) \wedge \mathrm{TCC}$ and $\zeta = 1$, then all nonzero off-diagonal entries of every diagonal block of PAP^T have modulus greater than or equal to γ.

3.3.3 Implementation Details

Algorithm 3.2 now describes the XPABLO algorithm in more detail, cf. Algorithm 3.1. In this section we will discuss several implementation details, many of them crucial for the complexity analysis in section 3.4. Line numbers in this section refer to Algorithm 3.2.

We first note that quantities needed for τ, such as $\deg|_B$, $\deg|_B^{\geq\gamma}$, etc., are always updated immediately. This allows a fast evaluation of τ which is very important for the overall performance. Since the nodes of the graph are stored in C, Q and B we access the original node set V only once at the very beginning of the algorithm to

1: **input:** a digraph $\mathcal{G}(A) = (V, E)$ and a criterion τ
2: **output:** a partitioning $\{V_1, \ldots, V_q\}$ of V
3: $C := V, \quad Q := \emptyset, \quad B := \emptyset, \quad q := 1$
4: set $\deg|_B(i) := 0$ and $\deg|_B^{>\gamma}(i) := 0$ for all $i \in V$
5: **while** $C \neq \emptyset$ **do**
6: remove a vertex i from C, and place it in B
7: set $|E|_B := 0$, $|E|_B^{>\gamma} := 0$, $\phi(B) := 0$, $\phi^{>\gamma}(B) := 0$
8: **for all** edges e in $\mathrm{inc}(i)$ **do**
9: let j be the vertex adjacent to i through e
10: move j from C to Q if $j \in C$
11: $\deg|_B(j) := \deg|_B(j) + 1$
12: $\deg|_B^{>\gamma}(j) := \deg|_B^{>\gamma}(j) + 1$ if $w(e) > \gamma$
13: **end for**
14: **while** $Q \neq \emptyset$ **do**
15: remove a vertex i from Q
16: **if** i fulfills the XPABLO criterion τ **then**
17: insert i into B
18: set $|E|_B := |E|_B + \deg|_B(i)$
19: set $|E|_B^{>\gamma} := |E|_B^{>\gamma} + \deg|_B^{>\gamma}(i)$
20: update $\phi(B)$ and $\phi^{>\gamma}(B)$ {this requires $|E|_B$ and $|E|_B^{>\gamma}$}
21: **for all** edges e in $\mathrm{inc}(i)$ **do**
22: let j be the vertex adjacent to i through e
23: move j from C to Q if $j \in C$
24: $\deg|_B(j) := \deg|_B(j) + 1$
25: $\deg|_B^{>\gamma}(j) := \deg|_B^{>\gamma}(j) + 1$ if $w(e) > \gamma$
26: **end for**
27: **if** $|B| > \mathit{maxbs}$ **then**
28: move all nodes in Q to C
29: **end if**
30: **else**
31: insert vertex i into C
32: **end if**
33: **end while**
34: set $V_q := B$ and $q := q + 1$ {finish current block}
35: **for all** vertices $i \in B$ **do**
36: **for all** edges e in $\mathrm{inc}(i)$ **do**
37: let j be the vertex adjacent to i through e
38: $\deg|_B(j) := 0, \quad \deg|_B^{>\gamma}(j) := 0, \quad \deg(j) := \deg(j) - 1$
39: **end for**
40: **end for**
41: $\mathcal{G} := \mathcal{G}|_C, \quad B := \emptyset$
42: **end while**
43: merge adjacent blocks of size $< \mathit{minbs}$ if the size of the resulting block is $\geq \mathit{maxbs}$

Algorithm 3.2. XPABLO with Implementation Details

copy it. Therefore, we do not need to reduce V during the algorithm. Morover, with explicit updates of the degrees we notice that the algorithm is still correct if we leave the graph unchanged, i.e., if we remove line 41. And in fact, in our implementation we do not explicitly reduce the graph.

For the implementation of Q we use a queue, hence the symbol. This choice is an implementation detail and not an intrinsic property of XPABLO. The only operations performed with Q are insertions and removal of nodes. Which node gets removed is up to the intrinsic properties of the data structure, i.e., XPABLO does not need to remove a specific node or a node with some specific property. A queue has time complexity $\mathcal{O}(1)$ for both operations (insert and remove). There are other data structures, such as a stack, which would allow the same time complexity. We have run internal experiments to compare using a queue and a stack to store Q. Although the results for a specific problem were sometimes quite different, there was no general trend to prefer a stack over a queue for implementing Q. Therefore, we decided to continue to use a queue, which was already the data structure used for Q in PABLO and TPABLO.

If we look carefully at the use of C we will notice that we have several operations which need to be performed very fast:

- Adding a node to C.
- Removing a (randomly selected) node from C.
- Removing a specific node from C.
- Checking whether a specific node is an element in C.

In our implementation we use a linked list with an external index, i.e., an index which contains for each node in the graph a pointer to the position in C or the information that the node is not an element of C. In this way, all the operations above have time complexity $\mathcal{O}(1)$.

3.4 Analysis of XPABLO

Looking at Algorithm 3.2 we see two nested loops, the outer loop running until C is empty and the inner loop running until Q is empty. Since Q can grow inside the inner loop, we have to check if and when the loop and hence the algorithm terminates.

Lemma 3.10: The algorithm XPABLO (Algorithm 3.2) always terminates.

Proof: We basically follow the proof from [36, p. 816]. At any time in the algorithm we have

$$|Q| \leq n, \qquad |C| \leq n, \qquad \text{and} \qquad |Q \cup C| = |Q| + |C| \leq n.$$

The outer loop (Algorithm 3.2, lines 5–42) terminates because the size of C decreases by at least one in each iteration (cf. line 7).

In the inner loop (lines 14–33), the number $|Q \cup C|$ does not increase, since nodes moved to C come from Q and vice versa. Moreover, after each pass through the inner loop, we have decreased either $|Q \cup C|$ or $|Q|$. In the first case we have moved a vertex to B which can cause Q to increase. In the second case we have moved a vertex from Q back to C.

With $|Q| < n$, there can be at most $n-1$ consecutive iterations of the inner loop in which we decrease $|Q|$ and leave $|Q \cup C|$ unchanged. Therefore, in n consecutive iterations of the inner loop $|Q \cup C|$ gets decreased at least once. At the start of the inner loop we have $|Q \cup C| < n$. Thus after at most n^2 iterations the inner loop must terminate. □

The time complexity of Algorithm 3.2 is given in the following result.

Theorem 3.11: The XPABLO algorithm can be implemented with time complexity $\mathcal{O}\big(n + \operatorname{nnz}(A)\big)$ whenever an evaluation of criterion τ has cost $\mathcal{O}(1)$.

Proof: The general idea behind this proof originates from [36], where it was shown that the entire PABLO algorithm has time complexity $\mathcal{O}\big(n + \operatorname{nnz}(A)\big)$. Here we will show that this time complexity also holds for XPABLO and hence also for TPABLO. The line numbers cited refer to Algorithm 3.2.

We assume that $\mathcal{G} = \mathcal{G}(A)$ is given stored in an adjacency list representation. For the common sparse matrix formats we can obtain $\mathcal{G}(A)$ with cost $\mathcal{O}\big(n + \operatorname{nnz}(A)\big)$; see, e.g., [19] and [21] for more information about sparse matrix formats.

Taking Q as a queue, C as a doubly linked list with an external index, and B as a singly linked list, all insert and remove operations we use with these lists can be done with cost $\mathcal{O}(1)$. Moving the contents of B to the newly created block V_ν, $\nu = 1, \ldots, q$, can be done with cost $\mathcal{O}(|V_\nu|)$. In total for all blocks together the extra cost is $\mathcal{O}(n)$.

The main observation to measure the time complexity of the inner loop is to realize that each node v is inserted into Q at most $\deg(v)$ times: Vertices are added to Q only when they are adjacent to some vertex i just inserted into B. A vertex i never returns to Q or C once it is inserted into B. Therefore each vertex v in the graph will be inserted at most $\deg(v)$ times into Q. Altogether there can be at most $\sum_{w \in V} \deg(w) = 2 \operatorname{nnz}(A)$ insertions into Q. Since each insert can be done in $\mathcal{O}(1)$ operations, this sums up to $\mathcal{O}\big(\operatorname{nnz}(A)\big)$ operations.

Consequently, moving vertices from Q back to C can be done in $\mathcal{O}(\operatorname{nnz}(A))$ operations, too.

XPABLO inserts a vertex i into B exactly n times. Every time a vertex i is inserted into B, the values of $\deg|_B(i)$ are updated for all $j \in Q$ adjacent to i (cf. lines 11 and 24). Analogously to the insertions into Q discussed above, in total at most $2 \operatorname{nnz}(A)$ updates have to be done. Since the whole adjacency list of i is traversed, each update can be done in $\mathcal{O}(1)$ operations.

When XPABLO ends the inner loop, the vertices in B are removed from the graph. This is done by updating the $\deg(w)$ value for all vertices w adjacent to any vertex $v \in B$. This also is done in $\mathcal{O}(\mathrm{nnz}(A))$.

By assumption, the cost for testing criterion τ is $\mathcal{O}(1)$ as it is in PABLO. Therefore, the additional work in XPABLO as compared to PABLO is that we have to update the new quantities $|E|_B^{>\gamma}$, $\phi^{>\gamma}(B)$, and $\deg|_B^{>\gamma}$ used in (TCC) and (TFC); see lines 12 and 25 (updates of $\deg|_B^{>\gamma}$), line 19 (update of $|E|_B^{>\gamma}$) and line 20 (update of $\phi^{>\gamma}(B)$). These updates follow the ones to be done for $|E|_B$, $\phi(B)$, and $\deg|_B$ by PABLO. In total, this means $\mathcal{O}(n)$ extra operations to update $|E|_B^{>\gamma}$ and $\phi^{>\gamma}(B)$ and $\mathcal{O}\big(\mathrm{nnz}(A)\big)$ extra operations to update $\deg|_B^{>\gamma}$. This results in an additional cost of $\mathcal{O}\big(n + \mathrm{nnz}(A)\big)$.

The following table summarizes the time complexities of the various operations:

Operations	Time complexity
Inserting vertices into Q	$\mathcal{O}(\mathrm{nnz}(A))$
Moving vertices back to C	$\mathcal{O}(\mathrm{nnz}(A))$
Inserting vertices into B	$\mathcal{O}(n)$
Updating $\lvert E\rvert_B$, $\lvert E\rvert_B^{>\gamma}$	$\mathcal{O}(n)$
Updating deg, $\deg_B$, $\deg_B^{>\gamma}$	$\mathcal{O}(\mathrm{nnz}(A))$

The total time complexity is then $\mathcal{O}\big(n + \mathrm{nnz}(A)\big)$, which establishes the theorem. □

Let us note that in our practical implementations we indeed update $|E|_B^{>\gamma}$, $\deg|_B^{>\gamma}$ explicitly. This is in contrast to the implementation of TPABLO in [14]. As is explained in [14], the implementation there is not more than d times as costly as PABLO when d is the maximum degree in $\mathcal{G}$. In our implementation we need a little

more memory, but require only about twice the work that PABLO does, independent of d. Therefore, our implementation is in most cases faster than that in [14].

We defer a detailed discussion on how to choose an XPABLO criterion τ and the various parameters (α, β, γ, ϑ, ζ) to the sections on practical issues and on numerical results (sections 3.5 and 3.7).

3.5 Practical Issues

Before reporting on our numerical experiments, we address some practical issues when using XPABLO in a preconditioning framework. We concentrate on the two main issues, namely, numerical stability and the choice of a good set of values for all the different XPABLO parameters.

3.5.1 Robustness

We consider the issue of numerical instability (or even the singularity) of the diagonal blocks D_i, $i = 1, \ldots, q$, in the block triangular preconditioner M. If A is a general nonsingular matrix, there is no guarantee that all the blocks D_i are nonsingular, nor that M is nonsingular. And even if all D_i are nonsingular, they may still be highly ill-conditioned. We mention that difficulties with singular or nearly singular blocks were very rare in our numerical experiments using the recommended parameters, as described in section 3.5.2. Problems with singular and nearly singular blocks are handled as follows: Let $D_i \in \mathbb{R}^{n_i \times n_i}$, $i \in \{1, \ldots, q\}$, be a problematic block. We then go back to the XPABLO blocks and replace the corresponding ith XPABLO block $V_i = \{v_1, \ldots, v_{n_i}\}$ by n_i new blocks $\{v_1\}$, $\ldots$, $\{v_{n_i}\}$. Since we have scaled and permuted A to be an I-matrix, the diagonal entries of A are ± 1 and hence the new diagonal blocks are all nonsingular.

If M is the lower block triangular part of A, this is equivalent to replacing D_i (in M) by the lower triangular part of D_i. In the same way we replace D_i by

its upper triangular part for backward block Gauss–Seidel preconditioning and by its diagonal for block Jacobi preconditioning. This is also summarized in Table 3.1. Since A was scaled and permuted to be an I-matrix, the lower triangular part, the upper triangular part, and the diagonal part of D_i are all I-matrices and hence nonsingular, even if D_i is singular.

Table 3.1. Replacement of Singular or Nearly Singular Blocks

Preconditioner	D_i replaced by...
block Jacobi	diagonal part
forward block Gauss–Seidel	lower triangular part
backward block Gauss–Seidel	upper triangular part

In our MATLAB implementation we use a simple but cheap test to determine if we consider D_i to be too ill-conditioned. Let $e = (1, \ldots, 1)^T \in \mathbb{R}^{n_i}$ be the vector of all ones and let $b = D_i e$. Then we consider D_i to be too ill-conditioned if

$$\left| 1 - \frac{\|\tilde{D}_i^{-1} b\|}{\|e\|} \right| > \sqrt{\epsilon_M},$$

where $x = \tilde{D}_i^{-1} b$ is computed by using the LU decomposition of D_i to determine the solution x of $D_i x = b$. With ϵ_M we denote the machine precision (machine epsilon).

The problems described cannot occur for the class of spd matrices or the class of H-matrices. Recall from Definition 2.3 that a matrix A is termed an H-matrix if there exist weights $u_j > 0$, $j = 1, \ldots, n$, such that for all $i = 1, \ldots, n$

$$|a_{ii}| \cdot u_i \; > \sum_{\substack{j=1,\ldots,n \\ j \neq i}} |a_{ij}| \cdot u_j.$$

The class of H-matrices contains the class of M-matrices (see Theorem 2.8), and they arise, e.g., in certain discretizations of (elliptic) boundary value problems; see, e.g., [11]. We have the following result.

Theorem 3.12: Assume that $A \in \mathbb{R}^{n\times n}$ is an H-matrix or that A is symmetric and positive definite. Then M in (3.4) is nonsingular; i.e., D_ν is nonsingular for each $\nu = 1, \ldots, q$.

Proof: Assume first that A is an H-matrix. Then M is also an H-matrix (with the same weights u_j as for A). Therefore, the diagonal blocks D_ν of M are H-matrices as well and they are thus nonsingular. Moreover, their LU factorizations can be performed without pivoting; see, e.g., [1].

If A is symmetric positive definite, each of the blocks D_ν, which are principal submatrices of A, are symmetric positive definite as well. In particular, they are all nonsingular and they admit a Cholesky factorization and an LU factorization without pivoting. □

Remark 3.13: The diagonal blocks D_ν of A in Theorem 3.12 are as "well conditioned" as A in the following sense:

If A is an H-matrix, then, by definition, $|a_{ii}|u_i - \sum_{j=1}^{n} |a_{ij}|u_j > 0$ for all $i = 1, \ldots, n$. If we replace A by $D = \mathrm{diag}(D_1, \ldots, D_q)$, then the left-hand side can increase, but can not decrease, i.e.,

$$|a_{ii}|u_i - \sum_{j\in V_\nu} |a_{ij}|u_j \;\geq\; |a_{ii}|u_i - \sum_{j\,\in\,\{1,\ldots,n\}} |a_{ij}|u_j \;>\; 0, \quad \text{for all } i \in V_\nu,\ \nu = 1, \ldots, q. \tag{3.3}$$

Let $\langle A\rangle$ denote the comparison matrix of A (see Definition 2.4). If we let $u = (u_1, \ldots, u_n)^T$ to be the vector of the H-matrix weights of A, then

$$\langle D\rangle u \;\geq\; \langle A\rangle u \;>\; 0.$$

Note that $\langle A \rangle u$ can be seen as a measure of how strong the diagonal dominance of AU with $U = \mathrm{diag}(u_1, \ldots, u_n)$ is.

If A is symmetric positive definite, then $x^T A x \geq \lambda_{\min} \|x\|_2^2$ for $x \neq 0$, where $\lambda_{\min}$ is the smallest eigenvalue of A. For a vector $x_\nu \in \mathbb{R}^{n_\nu}$ let $\hat{x}_\nu \in \mathbb{R}^n$ such that x_ν is the νth block subvector of $\hat{x}_\nu$, the remaining entries being zero. Then

$$x_\nu^T D_\nu x_\nu = \hat{x}_\nu^T A \hat{x}_\nu \geq \lambda_{\min} \|\hat{x}_\nu\|_2^2 = \lambda_{\min} \|x_\nu\|_2^2,$$

i.e., the smallest eigenvalue of D_ν, $\nu = 1, \ldots, q$ is larger or equal than $\lambda_{\min}$.

In view of the maximum transversal transformation $A \rightarrow \Sigma(\hat{R}A\hat{C})$ and the two-sided XPABLO permutation $A \rightarrow PAP^T$, it is important to notice that the H-matrix property, symmetry, and positive definiteness are preserved under any symmetric permutation and hence under the XPABLO permutation. In general, however, symmetry and positive definiteness will not be conserved under the transversal transformation. Morover, both the scaling and the unsymmetric permutation on its own can destroy symmetry and positive definiteness.

The H-matrix property, on the other hand, is preserved under the I-matrix scaling of MC64:

Theorem 3.14: Let A be an H-matrix and let Σ be a permutation matrix and let $\hat{R}$ and $\hat{C}$ be scaling matrices such that $\hat{A} = \Sigma(\hat{R}A\hat{C})$ is an I-matrix. Then $\hat{A}$ is an H-matrix.

Proof: We first note that the H-matrix property is preserved under the row permutation Σ, which permutes the rows of A. It is also preserved under the row-scaling $\hat{R}$, since no row is scaled with zero. It remains to show that the column scaling $\hat{C}$ preserves the H-matrix property. In our case it is sufficient to show that $A\hat{C}$ is an H-matrix. Let u_j, $j = 1, \ldots, n$ be the H-matrix weights of A. Now, let $\hat{u}_j = u_j / |\hat{c}_{jj}|$,

$j = 1, \ldots, n$. Then

$$|a_{ii}\hat{c}_{ii}| \cdot \hat{u}_i = |a_{ii}| \cdot u_i > \sum_{\substack{j=1,\ldots,n \\ j \neq i}} |a_{ij}| \cdot u_j = \sum_{\substack{j=1,\ldots,n \\ j \neq i}} |a_{ij}\hat{c}_{jj}| \cdot \hat{u}_j. \qquad \square$$

3.5.2 Choosing Parameters

We also have to carefully consider the choice of the XPABLO parameters and the actual criterion τ. In practice, a first important point is that of fixing a minimum and a maximum block size in Algorithm 3.2. With the following recommendations for minimum and maximum block size we aim to improve the total solving time in serial execution. Therefore, we do not need different blocks to be of the same or similar size to each other. Moreover, we do not aim to have a fixed number of blocks. To optimize the total time we have to find the right balance between a large block size, which allows us to capture large parts of the matrix, and a small block size, which allows us to factorize the block very fast. Notice, that the optimal block size—optimal in having the smallest total execution time—can depend heavily on the computer used. Even when we fix the problem and all involved algorithms and parameters. Thus, the recommendations can only be a weak guidance. Modern sparse direct solvers are often better suited to take advantage of a growing number of computation cores or a growing cache size than the employed Krylov subspace solvers. Therefore, we expect the optimal block sizes to increase in general if a newer and faster computer with more memory is used. In this chapter we will present results using $minbs = 200$ and $maxbs = 1000$, as we already did in [30].

The next, crucial, issue is that of finding adequate parameters and a suitable XPABLO criterion τ. We performed a long series of computations on many test problems, and we can give suggested values for most of the parameters resulting in run times of the preconditioned iterative methods which are quite close to the

Table 3.2. Recommended XPABLO Criterion τ and Parameter Values

τ	α	β	γ	δ	ζ
FC $\vee$ CC $\vee$ TCC	1.1	0.6	$\left(\sum \lvert a_{ij} \rvert\right) / \operatorname{nnz}(A)$	0.05	$1/(2n)$

individually best ones. For each given matrix better parameters could be found by trying a variety of different parameter settings and then choosing the best one (with respect to the resources being used, time, and storage). Of course, this is too costly in practice.

The suggested choices for both block Jacobi and block Gauss–Seidel preconditioning are presented in Table 3.2. If these recommendations are used, the only difference between XPABLO and TPABLO1 is a change in τ. XPABLO has $\tau =$ FC $\vee$ CC $\vee$ TCC and TPABLO1 has $\tau =$ (FC $\vee$ CC) $\wedge$ TCC. The difference between XPABLO and TPABLO1 might appear small, but in almost all cases XPABLO is clearly to be preferred compared to TPABLO1.

As a last issue, we further discuss our choice for γ, the threshold parameter used in (TCC) and (TFC). Of all the parameter recommendations, the one for γ is the weakest in the sense that it is the most likely candidate for not being a good choice for a particular problem. Therefore, we will present two different recommendations. As shown in Table 3.2, the default choice of γ for XPABLO is the average of the magnitudes of the nonzero entries, a quantity that can easily be computed and proved to give good overall results in our experiments.

The second recommendation—which also gives good overall results and for some problems even better results than the default choice—is the following. After having scaled (and permuted) the matrix according to (3.2), it seems reasonable to use γ as a "percentage" parameter; i.e., we choose γ such that a given fraction $\gamma' \in [0, 1]$ of the nonzero elements are dropped when passing from A to $A^{>\gamma}$. Since

we know the number $|E|$ of nonzeros in A, finding the corresponding γ is an instance of the *kth largest element problem*: γ is the value of the kth largest element of A with $k = \lfloor \gamma' |E| \rfloor$. Note that the kth largest element problem can be solved with linear complexity; i.e., its cost is $\mathcal{O}(|E|)$; see, e.g., [16].

3.6 Efficient Block Gauss–Seidel Preconditioning

Let A denote the matrix obtained after the preprocessing steps of section 3.2 and after applying the XPABLO permutation P of section 3.3, i.e., $A := P(\Sigma \hat{R} A \hat{C}) P^T$, where Σ is the permutation matrix found by MC64 and $\hat{R}$ and $\hat{C}$ are the scaling matrices found by MC64, cf. (3.2). Then A has a block structure

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & \ddots & \vdots \\ A_{q1} & \cdots & A_{qq} \end{bmatrix}, \qquad A_{ij} \in \mathbb{R}^{n_i \times n_j},$$

with $n_i = |V_i|$ being the size of the blocks corresponding to the XPABLO permutation P. Each diagonal block $D_i = A_{ii}$ in the block diagonal $D = \operatorname{diag}(D_1, \ldots, D_q)$ should be (relatively) full and should have (relatively) large nonzero entries. Even with A nonsingular, some of the diagonal blocks D_i can be singular; see section 3.5.1 for a discussion of this issue. For the moment we assume all D_i to be nonsingular.

We can compute the (row pivoted) LU factorization $D_i = \Pi_i L_i' U_i'$ for each block, so that $D = \Pi\, L'\, U'$, $\Pi = \operatorname{diag}(\Pi_1, \ldots, \Pi_q)$, $L' = \operatorname{diag}(L_1', \ldots, L_q')$, and $U' = \operatorname{diag}(U_1', \ldots, U_q')$, and we use this LU factorization of D when it comes to solving linear systems of the form

$$Ds = r$$

in a preconditioned iterative method with preconditioner D. This block diagonal preconditioning approach (with PABLO, TPABLO1, and TPABLO2) was used in [15], where the numerical results were quite encouraging. Note that when the diagonal

block D_i is dense, we could use the optimized dense matrix linear algebra code from LAPACK to perform these factorizations very efficiently [2]. When they are sparse, it is highly recommended to use a sparse factorization. For simplicity, we always use the sparse factorization provided by UMFPACK [18]. This is motivated by the observation that in our experiments the diagonal blocks are mostly "very" sparse, since small dense blocks are usually merged into larger sparse ones to obtain blocks of size *minbs* or larger.

An obvious extension of this idea is to use block Gauss–Seidel preconditioning, i.e., to include a block triangular part of A into the preconditioner. In the following discussion we will use the lower block triangular part, i.e., a forward block Gauss–Seidel iteration, as the preconditioner. It is easy to see that Proposition 3.15 below also holds when the preconditioner M is the upper block triangular part of A.

Note that block Gauss–Seidel iteration without acceleration by a Krylov subspace method was used for the numerical experiments in the original PABLO paper [36]. Using the established block structure the block Gauss–Seidel preconditioner M_{GS} can be written as

$$M_{\mathrm{GS}} = \begin{bmatrix} D_1 & 0 & \cdots & 0 \\ A_{21} & D_2 & & \vdots \\ \vdots & & \ddots & 0 \\ A_{q1} & A_{q2} & \cdots & D_q \end{bmatrix}. \tag{3.4}$$

In a (left) preconditioned sparse iterative solver the preconditioned matrix-vector multiplication

$$z = M^{-1}Av \tag{3.5}$$

with preconditioner M^{-1} is usually the most dominant operation in terms of execution time. It is common to separate this operation into a matrix-vector multiplication $y = Av$ and a solve operation $z = M^{-1}y$. This is done, e.g., in the MATLAB interfaces to Krylov subspace methods like CG and GMRES. When we compute (3.5)

in this way, applying the block Gauss–Seidel preconditioner by computing $z = M_{\mathrm{GS}}^{-1} y$ is much more expensive than applying the block Jacobi preconditioner by computing $z = D^{-1} y$.

However, it is sometimes possible to compute (3.5) in a much more efficient way if the matrix-vector product and the application of the preconditioner are not separated. In the next result we show that this is the case for block Gauss–Seidel preconditioning. The result is very similar to *Eisenstat's trick* [27] to implement an efficient preconditioned conjugate gradient (PCG) method for the class of preconditioners of the form

$$M = (\tilde{D} - L)\tilde{D}^{-1}(\tilde{D} - L)^T$$

in which $-L$ is the strict lower triangular part of A ($A = D - L - L^T$) and $\tilde{D}$ is a positive diagonal matrix, not necessarily the diagonal of A. This class includes, e.g., the symmetric Gauss–Seidel preconditioner $M = (D-L)D^{-1}(D-L)^T$. Several other similar ideas are discussed in [37]. See also [38, pp. 208, 225] and [41, pp. 265ff].

Theorem 3.15: Let $A \in \mathbb{R}^{n \times n}$ be a matrix with a $q \times q$ block structure such that all diagonal blocks $D_i = A_{ii}$, $i = 1, \ldots, q$, are nonsingular. We denote $D = \mathrm{diag}(D_1, \ldots, D_q)$ to be the block diagonal part and M_{GS} to be the lower block triangular part (including the diagonal blocks) of A. The preconditioned matrix-vector multiplications $D^{-1}Av$ and $M_{\mathrm{GS}}^{-1}Av$ can both be done using exactly the same number of operations.

Proof: Let $z = M_{\mathrm{GS}}^{-1}Av$ and $\tilde{z} = D^{-1}Av$. For any vector $w \in \mathbb{R}^n$ we write w_i to denote the ith block, conformal with the block structure of A. We split A as $A = D - L - U$ into the block diagonal part $D = \mathrm{diag}(D_1, \ldots, D_q)$ (same as before), the block lower triangular part L with $L_{ij} = -A_{ij}$ for $i > j$, and the block upper triangular part U with $U_{ij} = -A_{ij}$ for $i < j$. Both L and U have a zero block diagonal.

Using this notation the block (lower) triangular preconditioner is $M_{\mathrm{GS}} = D - L$, and we can rewrite the preconditioned matrix-vector multiplication as

$$\begin{aligned} M_{\mathrm{GS}}^{-1} A v &= (D-L)^{-1}(D-L-U)v \\ &= (D-L)^{-1}(D-L)v - (D-L)^{-1}Uv \\ &= v - (D-L)^{-1}Uv. \end{aligned}$$

If we set $w = Uv$ and $y = (D-L)^{-1}w$, we get $z_i = v_i - y_i$, $i = 1, \ldots, q$, where

$$y_i = D_i^{-1}\left(w_i + \sum_{j=1}^{i-1} L_{ij} y_j\right) \qquad \text{and} \qquad w_i = \sum_{j=i+1}^{q} U_{ij} v_j.$$

We can further simplify the formula for y_i to

$$y_i = -D_i^{-1}\left(\sum_{j=1}^{i-1} A_{ij} y_j + \sum_{j=i+1}^{q} A_{ij} v_j\right). \tag{3.6}$$

Following the same procedure, we write $\tilde{z} = D^{-1}Av = v - \tilde{y}$ and get $\tilde{z}_i = v_i - \tilde{y}_i$, where

$$\tilde{y}_i = -D_i^{-1}\left(\sum_{j=1}^{i-1} A_{ij} v_j + \sum_{j=i+1}^{q} A_{ij} v_j\right). \tag{3.7}$$

But the computations (3.6) and (3.7) use the exact same number of operations if we assume that dense vectors are employed. □

3.7 Numerical Experiments

In this section we present numerical results using XPABLO as a tool for preconditioning. We start by showing the improvements of XPABLO over PABLO and TPABLO, and then provide a detailed comparison of XPABLO and ILUTP as a tool for preconditioning. All experiments were run on an Intel Core 2 processor. The test programs are written in a mixture of MATLAB (version 7.7, release R2008b), C, C++, and Fortran. The non-MATLAB code was compiled using version 4.1 of the GNU Compiler

Collection (GCC), identical optimization flags were used in all cases. For iterative solving we use MATLAB's `gmres` function, which uses Householder transformations for the orthogonalization in the Arnoldi process, see also [49]. For computations that are the same in the different preconditioned solvers, we use the same code—whether it is compiled C/C++/Fortran code or MATLAB code. Therefore, we can somewhat compare the execution times and not only the iteration counts. Note, however, that the timings are not fully reliable. The experiments were run on a fairly standard Laptop running Linux. Therefore, we always have several dozen other applications running in parallel with our code and their behavior can influence the measured time for our experiments, e.g., by filling the cache of the processor(s) with their own code and/or data. Additionally, with issues like swapping and today's complicated cache hierarchies, it is possible that an experiment can influence the timings of later experiments. We could possible eliminate most of these effects by rebooting the computer after each experiment. In practice the observed unreliabilities in the timings were way too small to justify such countermeasures.

Furthermore, we notice that the timings presented here can not be compared with an optimized and fully compiled implementation. This is mostly due to the involved MATLAB code. We give here just some of the factors contributing to MATLAB's performance being far from optimal:

- The code is often interpreted, even though a just-in-time compiler is employed in modern MATLAB versions.
- Loops involve a considerably overhead compared to compiled languages so that, e.g., a growing number of blocks from XPABLO brings a growing overhead in the construction of the preconditioner and in each iteration step.
- The interface between MATLAB and the C/C++/Fortran part of our code often requires otherwise unnecessary copies of data.

An overview of the test problems is given in section 1.1. Preliminary tests have shown that the XPABLO framework can perform well for problems stemming from computational fluid dynamics (CFD) and semiconductor device simulation. Therefore, some test matrices were specifically chosen from these application areas. See the application area column in Table 1.1. We note that PABLO was not designed specifically to solve these kinds of problems. As noted in the introduction of this chapter, a modified version of PABLO has been used before for problems in CFD [26]. More information on solving systems from semiconductor device simulation can be found in [43].

To illustrate the different phases of the XPABLO framework, Figure 3.4 shows matrix plots of the matrix `GARON1` during four different phases. The bottom right plot shows the matrix of the system we are actually solving.

3.7.1 Comparison of PABLO, TPABLO1, and XPABLO

Table 3.6 shows the execution times and iteration counts using the XPABLO framework compared to PABLO- and TPABLO1-based preconditioning. If the iteration count is given in parentheses, this indicates that we did not observe convergence after 1000 iterations. In this case we show the iteration number for which the smallest preconditioned relative residual norm was achieved for the first time, i.e., the iteration number shown in parenthesis may be smaller than 1000 although we have computed 1000 iterations. Note that in such a case we essentially have stagnation, and the code gives the "solution" computed at the last iteration number before stagnating. More detailed results, including the relative residual norms $\|b - Ax\|/\|b\|$ (Rel. Res.) and, if the exact solution x_* is known, the relative error norms $\|x_* - x\|/\|x_*\|$ (Rel. Err.), are given in Table 3.3 (PABLO), Table 3.4 (TPABLO1), and Table 3.5 (XPABLO). Note that we report *unpreconditioned* relative residuals in the Rel. Res. column. They

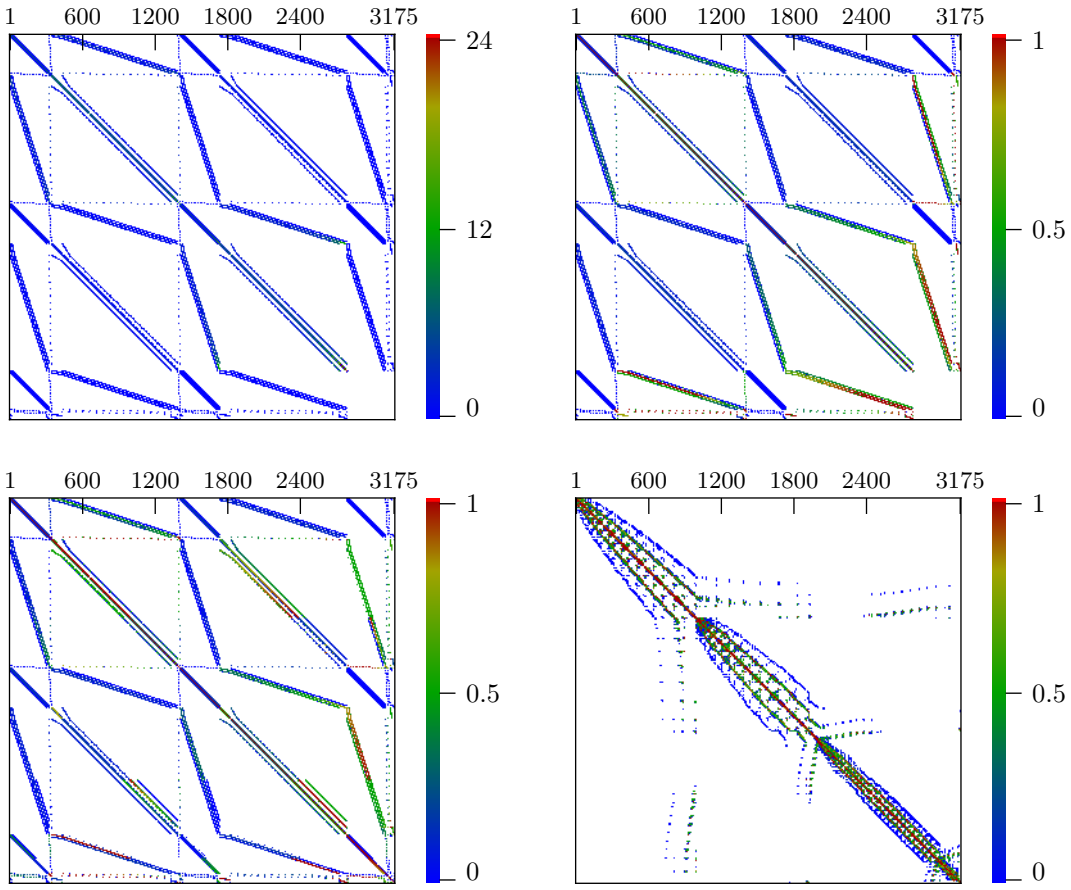


Figure 3.4. Spy Plots of `GARON1` During Four Phases of the XPABLO Framework

Top left: The original matrix. Top right: The matrix scaled by MC64, but not yet permuted to be an I-matrix. Bottom left: After scaling and permuting by MC64. The matrix is now an I-matrix. Bottom right: The matrix after the XPABLO permutation. Note the difference in scale between the top left and the other plots.

can not be compared with the preconditioned relative residuals used in the stopping criterion of the iterative solver.

All parameters are set to the values described in section 3.5; in particular, γ is set to the average of all nonzero entries. The block sizes were controlled by $minbs = 200$ and $maxbs = 1000$. Only the criterion τ is varied. In all cases, we first use MC64 to scale and permute the linear system. GMRES(50) is used as the solver and we always use $x_0 = 0$ as the initial approximation. The "preconditioner" is the lower block triangular part of the scaled and permuted matrix, using the blocks found by (T/X)PABLO. The real preconditioner is more complicated: Let M be the lower block triangular part of the scaled and permuted matrix $P\Sigma\hat{R}A\hat{C}P^T$, where P is the XPABLO permutation, Σ is the permutation found by MC64 and $\hat{R}$ and $\hat{C}$ are the scaling matrices found by MC64. In our experiments we then use GMRES(50) to solve

$$\underbrace{M^{-1}P\Sigma\hat{R}}_{M_L^{-1}} A \underbrace{\hat{C}P^T}_{M_R^{-1}} y = \underbrace{M^{-1}P\Sigma\hat{R}}_{M_L^{-1}} b, \quad x = \underbrace{\hat{C}P^T}_{M_R^{-1}} y,$$

i.e., we actually use a split preconditioner. Let $x_k = M_R^{-1}y_k$ and y_k be the kth approximation to x and y, respectively. The (preconditioned) relative residual $\|r_k\|/\|r_0\|$ is then

$$\text{preconditioned relative residual} = \frac{\|r_k\|}{\|r_0\|} = \frac{\|M_L^{-1}(b - Ax_k)\|}{\|M_L^{-1}b\|}.$$

For comparison, the unpreconditioned relative residual norm reported in the Rel. Res. column is

$$\text{Rel. Res.} = \frac{\|b - Ax_k\|}{\|b\|}.$$

Note that these two relative residual norms can be very different if M_L^{-1} is ill-conditioned.

Table 3.3. PABLO Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	3.35	346	$6.992 \cdot 10^{-8}$	$1.158 \cdot 10^{-5}$
CAVITY26	0.195	4	$1.342 \cdot 10^{+0}$	$2.917 \cdot 10^{+1}$
EX19	24.3	900	$3.691 \cdot 10^{-8}$	$1.520 \cdot 10^{+0}$
EX35	3.95	78	$5.990 \cdot 10^{-5}$	$5.378 \cdot 10^{+0}$
GARON1	1.92	265	$2.096 \cdot 10^{-8}$	$7.825 \cdot 10^{-7}$
GARON2	33.5	(1000)	$1.788 \cdot 10^{-4}$	$4.099 \cdot 10^{-2}$
RAEFSKY2	1.6	141	$7.042 \cdot 10^{-9}$	na
RAEFSKY3	66.2	(1000)	$1.792 \cdot 10^{-10}$	na
SHYY41	0.083	6	$7.526 \cdot 10^{-2}$	$1.381 \cdot 10^{+0}$
SHYY161	384.9	(200)	$1.306 \cdot 10^{+10}$	$1.156 \cdot 10^{+14}$
IGBT3	9.99	430	$2.328 \cdot 10^{-4}$	na
NMOS3	9.3	194	$2.930 \cdot 10^{-5}$	na
BARRIER2-1	869.5	(1000)	$3.229 \cdot 10^{-3}$	na
PARA-4	1031.9	692	$5.002 \cdot 10^{-5}$	na
PARA-8	967.6	621	$6.409 \cdot 10^{-10}$	na
OHNE2	2062.5	(1000)	$6.501 \cdot 10^{-3}$	na
2D_54019_HIGHK	197.8	(1000)	$7.628 \cdot 10^{-8}$	na
3D_51448_3D	168.4	816	$2.797 \cdot 10^{-12}$	na
IBM_MATRIX_2	206.0	(1000)	$3.991 \cdot 10^{-12}$	na
MATRIX_9	195.4	265	$5.113 \cdot 10^{-8}$	na
MATRIX-NEW_3	648.0	(1000)	$5.859 \cdot 10^{-8}$	na
LSQ_2D_1000	0.024	1	$2.756 \cdot 10^{-10}$	$6.534 \cdot 10^{-14}$
LSQ_2D_2000	0.116	21	$3.758 \cdot 10^{-5}$	$1.606 \cdot 10^{-8}$
LSQ_2D_5000	0.478	39	$6.469 \cdot 10^{-5}$	$2.485 \cdot 10^{-8}$
LSQ_2D_10000	1.39	51	$1.709 \cdot 10^{-4}$	$6.226 \cdot 10^{-8}$
LSQ_2D_50000	32.1	148	$5.590 \cdot 10^{-4}$	$4.619 \cdot 10^{-7}$
LSQ_2D_100000	144.8	204	$1.080 \cdot 10^{-3}$	$7.570 \cdot 10^{-7}$
LSQ_2D_200000	699.3	280	$2.159 \cdot 10^{-3}$	$3.396 \cdot 10^{-6}$
LSQ_2D_400000	5334.3	583	$3.121 \cdot 10^{-3}$	$5.915 \cdot 10^{-6}$
MPS_2D_10000	1.58	77	$2.623 \cdot 10^{-4}$	$4.610 \cdot 10^{-7}$
MPS_2D_50000	39	197	$1.284 \cdot 10^{-3}$	$9.466 \cdot 10^{-7}$
MPS_2D_100000	187.1	278	$2.163 \cdot 10^{-3}$	$3.410 \cdot 10^{-6}$
MPS_2D_200000	1261.2	517	$2.153 \cdot 10^{-3}$	$5.752 \cdot 10^{-6}$
MPS_2D_400000	7813.3	871	$2.713 \cdot 10^{-3}$	$1.115 \cdot 10^{-5}$
LSQ_3D_10000	0.68	21	$6.939 \cdot 10^{-6}$	$2.658 \cdot 10^{-8}$
LSQ_3D_50000	9.28	35	$2.642 \cdot 10^{-5}$	$5.426 \cdot 10^{-8}$
LSQ_3D_100000	36.6	44	$3.119 \cdot 10^{-5}$	$9.324 \cdot 10^{-8}$
LSQ_3D_200000	152.2	54	$6.115 \cdot 10^{-5}$	$1.239 \cdot 10^{-7}$
MPS_3D_10000	0.643	29	$1.137 \cdot 10^{-5}$	$4.636 \cdot 10^{-8}$
MPS_3D_50000	10.4	50	$5.710 \cdot 10^{-5}$	$1.378 \cdot 10^{-7}$
MPS_3D_100000	44.8	64	$7.976 \cdot 10^{-5}$	$1.587 \cdot 10^{-7}$
MPS_3D_200000	212.0	86	$1.362 \cdot 10^{-4}$	$6.861 \cdot 10^{-7}$

Table 3.4. TPABLO1 Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	2.36	241	$5.696 \cdot 10^{-8}$	$8.019 \cdot 10^{-6}$
CAVITY26	1.83	174	$1.770 \cdot 10^{-3}$	$1.040 \cdot 10^{+0}$
EX19	27.0	(950)	$5.676 \cdot 10^{-9}$	$3.446 \cdot 10^{-1}$
EX35	4.23	82	$6.949 \cdot 10^{-5}$	$1.094 \cdot 10^{+0}$
GARON1	1.66	228	$2.518 \cdot 10^{-8}$	$1.257 \cdot 10^{-6}$
GARON2	33.5	(1000)	$1.129 \cdot 10^{-6}$	$1.639 \cdot 10^{-4}$
RAEFSKY2	1.39	117	$7.728 \cdot 10^{-9}$	na
RAEFSKY3	66.9	(1000)	$3.432 \cdot 10^{-10}$	na
SHYY41	0.090	8	$2.292 \cdot 10^{-2}$	$1.916 \cdot 10^{+0}$
SHYY161	384.4	(1000)	$4.485 \cdot 10^{+3}$	$1.382 \cdot 10^{+7}$
IGBT3	15.3	619	$7.567 \cdot 10^{-4}$	na
NMOS3	34.6	733	$1.542 \cdot 10^{-6}$	na
BARRIER2-1	768.7	(1000)	$8.220 \cdot 10^{-4}$	na
PARA-4	703.1	539	$4.080 \cdot 10^{-5}$	na
PARA-8	659.1	483	$7.799 \cdot 10^{-10}$	na
OHNE2	1974.1	989	$1.542 \cdot 10^{-3}$	na
2D_54019_HIGHK	188.4	969	$2.235 \cdot 10^{-9}$	na
3D_51448_3D	89.9	484	$8.200 \cdot 10^{-12}$	na
IBM_MATRIX_2	81.3	439	$9.072 \cdot 10^{-12}$	na
MATRIX_9	211.2	298	$7.014 \cdot 10^{-8}$	na
MATRIX-NEW_3	602.0	(1000)	$1.641 \cdot 10^{-9}$	na
LSQ_2D_1000	0.023	1	$2.755 \cdot 10^{-10}$	$6.550 \cdot 10^{-14}$
LSQ_2D_2000	0.139	27	$1.482 \cdot 10^{-5}$	$8.283 \cdot 10^{-9}$
LSQ_2D_5000	0.486	38	$6.298 \cdot 10^{-5}$	$2.259 \cdot 10^{-8}$
LSQ_2D_10000	1.54	60	$1.052 \cdot 10^{-4}$	$1.413 \cdot 10^{-7}$
LSQ_2D_50000	35.3	162	$4.615 \cdot 10^{-4}$	$8.076 \cdot 10^{-7}$
LSQ_2D_100000	154.4	221	$9.439 \cdot 10^{-4}$	$1.188 \cdot 10^{-6}$
LSQ_2D_200000	836.9	340	$1.736 \cdot 10^{-3}$	$3.335 \cdot 10^{-6}$
LSQ_2D_400000	5866.6	653	$2.899 \cdot 10^{-3}$	$4.776 \cdot 10^{-6}$
MPS_2D_10000	1.94	97	$2.325 \cdot 10^{-4}$	$5.015 \cdot 10^{-7}$
MPS_2D_50000	46.5	265	$1.047 \cdot 10^{-3}$	$2.991 \cdot 10^{-6}$
MPS_2D_100000	288.5	495	$2.070 \cdot 10^{-3}$	$3.475 \cdot 10^{-6}$
MPS_2D_200000	1779.7	854	$2.052 \cdot 10^{-3}$	$7.978 \cdot 10^{-6}$
MPS_2D_400000	7648.7	(1000)	$9.084 \cdot 10^{-2}$	$5.396 \cdot 10^{-4}$
LSQ_3D_10000	0.602	20	$7.752 \cdot 10^{-6}$	$1.908 \cdot 10^{-8}$
LSQ_3D_50000	7.94	35	$1.609 \cdot 10^{-5}$	$4.523 \cdot 10^{-8}$
LSQ_3D_100000	30.5	43	$3.014 \cdot 10^{-5}$	$5.987 \cdot 10^{-8}$
LSQ_3D_200000	129.7	53	$5.574 \cdot 10^{-5}$	$9.389 \cdot 10^{-8}$
MPS_3D_10000	0.533	30	$1.641 \cdot 10^{-5}$	$3.902 \cdot 10^{-8}$
MPS_3D_50000	8.38	50	$5.182 \cdot 10^{-5}$	$1.114 \cdot 10^{-7}$
MPS_3D_100000	36.9	68	$8.528 \cdot 10^{-5}$	$2.246 \cdot 10^{-7}$
MPS_3D_200000	170.0	86	$1.154 \cdot 10^{-4}$	$4.814 \cdot 10^{-7}$

Table 3.5. XPABLO Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	0.601	49	$4.544 \cdot 10^{-8}$	$1.718 \cdot 10^{-6}$
CAVITY26	0.61	45	$9.208 \cdot 10^{-5}$	$3.290 \cdot 10^{-2}$
EX19	6.66	252	$2.172 \cdot 10^{-9}$	$3.168 \cdot 10^{-1}$
EX35	2.48	28	$2.606 \cdot 10^{-4}$	$3.905 \cdot 10^{-1}$
GARON1	0.333	37	$2.294 \cdot 10^{-8}$	$8.023 \cdot 10^{-7}$
GARON2	5.91	163	$1.077 \cdot 10^{-8}$	$1.744 \cdot 10^{-6}$
RAEFSKY2	0.665	39	$1.106 \cdot 10^{-8}$	na
RAEFSKY3	31.8	390	$3.555 \cdot 10^{-10}$	na
SHYY41	0.112	6	$7.738 \cdot 10^{-2}$	$9.402 \cdot 10^{-1}$
SHYY161	9.14	19	$4.032 \cdot 10^{+14}$	$6.795 \cdot 10^{+17}$
IGBT3	3.29	120	$2.492 \cdot 10^{-3}$	na
NMOS3	4.99	76	$2.490 \cdot 10^{-5}$	na
BARRIER2-1	854.5	680	$2.845 \cdot 10^{-5}$	na
PARA-4	519.0	210	$2.241 \cdot 10^{-5}$	na
PARA-8	196.8	72	$3.205 \cdot 10^{-10}$	na
OHNE2	1339.2	517	$8.692 \cdot 10^{-4}$	na
2D_54019_HIGHK	49.4	177	$8.701 \cdot 10^{-10}$	na
3D_51448_3D	16.2	44	$3.762 \cdot 10^{-11}$	na
IBM_MATRIX_2	15.2	42	$1.114 \cdot 10^{-10}$	na
MATRIX_9	139.5	142	$1.629 \cdot 10^{-7}$	na
MATRIX-NEW_3	80.9	93	$5.423 \cdot 10^{-11}$	na
LSQ_2D_1000	0.024	1	$2.781 \cdot 10^{-10}$	$6.581 \cdot 10^{-14}$
LSQ_2D_2000	0.084	15	$3.824 \cdot 10^{-5}$	$7.961 \cdot 10^{-9}$
LSQ_2D_5000	0.369	24	$9.347 \cdot 10^{-5}$	$1.728 \cdot 10^{-8}$
LSQ_2D_10000	1.04	34	$2.007 \cdot 10^{-4}$	$3.845 \cdot 10^{-8}$
LSQ_2D_50000	18.4	75	$5.253 \cdot 10^{-4}$	$5.178 \cdot 10^{-7}$
LSQ_2D_100000	88.4	111	$1.049 \cdot 10^{-3}$	$6.596 \cdot 10^{-7}$
LSQ_2D_200000	525.0	188	$1.715 \cdot 10^{-3}$	$9.223 \cdot 10^{-7}$
LSQ_2D_400000	2467.8	240	$2.415 \cdot 10^{-3}$	$2.582 \cdot 10^{-6}$
MPS_2D_10000	1.56	73	$2.874 \cdot 10^{-4}$	$4.996 \cdot 10^{-7}$
MPS_2D_50000	36.6	185	$1.096 \cdot 10^{-3}$	$1.460 \cdot 10^{-6}$
MPS_2D_100000	172.8	257	$1.869 \cdot 10^{-3}$	$3.495 \cdot 10^{-6}$
MPS_2D_200000	1019.8	418	$1.943 \cdot 10^{-3}$	$3.972 \cdot 10^{-6}$
MPS_2D_400000	6889.9	759	$2.386 \cdot 10^{-3}$	$8.651 \cdot 10^{-6}$
LSQ_3D_10000	0.779	18	$3.247 \cdot 10^{-6}$	$6.528 \cdot 10^{-9}$
LSQ_3D_50000	9.49	26	$1.106 \cdot 10^{-5}$	$2.074 \cdot 10^{-8}$
LSQ_3D_100000	33.2	31	$2.626 \cdot 10^{-5}$	$4.299 \cdot 10^{-8}$
LSQ_3D_200000	127.9	38	$3.329 \cdot 10^{-5}$	$3.919 \cdot 10^{-8}$
MPS_3D_10000	0.599	27	$1.671 \cdot 10^{-5}$	$5.317 \cdot 10^{-8}$
MPS_3D_50000	9.18	44	$4.791 \cdot 10^{-5}$	$7.248 \cdot 10^{-8}$
MPS_3D_100000	39.4	55	$6.064 \cdot 10^{-5}$	$1.202 \cdot 10^{-7}$
MPS_3D_200000	176.1	70	$1.125 \cdot 10^{-4}$	$2.599 \cdot 10^{-7}$

Table 3.6. Comparison of PABLO, TPABLO1, and XPABLO Results

Matrix	PABLO		TPABLO1		XPABLO	
	Time	Iter	Time	Iter	Time	Iter
CAVITY16	3.35	346	2.36	241	**0.601**	**49**
CAVITY26	**0.195**	**4**	1.83	174	0.61	45
EX19	24.3	900	27.0	(950)	**6.66**	**252**
EX35	3.95	78	4.23	82	**2.48**	**28**
GARON1	1.92	265	1.66	228	**0.333**	**37**
GARON2	33.5	(1000)	33.5	(1000)	**5.91**	**163**
RAEFSKY2	1.6	141	1.39	117	**0.665**	**39**
RAEFSKY3	66.2	(1000)	66.9	(1000)	**31.8**	**390**
SHYY41	**0.083**	**6**	0.090	8	0.112	**6**
SHYY161	384.9	(200)	384.4	(1000)	**9.14**	**19**
IGBT3	9.99	430	15.3	619	**3.29**	**120**
NMOS3	9.3	194	34.6	733	**4.99**	**76**
BARRIER2-1	869.5	(1000)	768.7	(1000)	**854.5**	**680**
PARA-4	1031.9	692	703.1	539	**519.0**	**210**
PARA-8	967.6	621	659.1	483	**196.8**	**72**
OHNE2	2062.5	(1000)	1974.1	989	**1339.2**	**517**
2D_54019_HIGHK	197.8	(1000)	188.4	969	**49.4**	**177**
3D_51448_3D	168.4	816	89.9	484	**16.2**	**44**
IBM_MATRIX_2	206.0	(1000)	81.3	439	**15.2**	**42**
MATRIX_9	195.4	265	211.2	298	**139.5**	**142**
MATRIX-NEW_3	648.0	(1000)	602.0	(1000)	**80.9**	**93**
LSQ_2D_1000	0.024	**1**	**0.023**	**1**	0.024	**1**
LSQ_2D_2000	0.116	21	0.139	27	**0.084**	**15**
LSQ_2D_5000	0.478	39	0.486	38	**0.369**	**24**
LSQ_2D_10000	1.39	51	1.54	60	**1.04**	**34**
LSQ_2D_50000	32.1	148	35.3	162	**18.4**	**75**
LSQ_2D_100000	144.8	204	154.4	221	**88.4**	**111**
LSQ_2D_200000	699.3	280	836.9	340	**525.0**	**188**
LSQ_2D_400000	5334.3	583	5866.6	653	**2467.8**	**240**
MPS_2D_10000	1.58	77	1.94	97	**1.56**	**73**
MPS_2D_50000	39	197	46.5	265	**36.6**	**185**
MPS_2D_100000	187.1	278	288.5	495	**172.8**	**257**
MPS_2D_200000	1261.2	517	1779.7	854	**1019.8**	**418**
MPS_2D_400000	7813.3	871	7648.7	(1000)	6889.9	759
LSQ_3D_10000	0.68	21	**0.602**	20	0.779	**18**
LSQ_3D_50000	9.28	35	**7.94**	35	9.49	**26**
LSQ_3D_100000	36.6	44	**30.5**	43	33.2	**31**
LSQ_3D_200000	152.2	54	129.7	53	**127.9**	**38**
MPS_3D_10000	0.643	29	**0.533**	30	0.599	**27**
MPS_3D_50000	10.4	50	**8.38**	50	9.18	**44**
MPS_3D_100000	44.8	64	**36.9**	68	39.4	**55**
MPS_3D_200000	212.0	86	**170.0**	86	176.1	**70**

If the iteration count is given in parentheses, this indicates that we did not observe convergence after 20 cycles of GMRES(50), i.e., after 1000 iterations. In this case we show the iteration number for which the smallest preconditioned relative residual norm was achieved, i.e., the iteration number shown in parenthesis may be smaller than 1000 although we have computed 1000 iterations. The "solution" taken for the computation of the (non-preconditioned) relative residual and relative error norms is the approximation computed at the given iteration. For each matrix in Table 3.6, the numbers in bold indicate the smallest execution time and the lowest iteration number. More details on the stopping criterion are given later. Note that PABLO-, TPABLO1-, or XPABLO-based preconditioners need only one iteration if only one block is found by (T/X)PABLO. This happens, e.g., for the `LSQ_2D_1000` matrix. The robustness and the performance of XPABLO compared to PABLO and TPABLO1 can be readily appreciated.

3.7.2 Comparison of XPABLO and ILU

We now describe how we compare XPABLO with ILUTP as a tool for preconditioning. See section 2.2.1 for a short description of the ILUTP preconditioner used in the experiments. In preliminary tests the three drop tolerances used in the experiments (10^{-2}, 10^{-3}, and 10^{-4}) have been found to be good values for the selected problems. We use both XPABLO and ILUTP in conjunction with MC64; see section 3.2. We additionally precede ILUTP with a reverse Cuthill–McKee ordering (RCM); this follows the recommendations in [8], [9], and [43]. We do not use RCM together with XPABLO as our experiments have indicated that it does not further improve the convergence.

As already mentioned, we use GMRES(50) as the iterative solver in all our experiments. The stopping criterion is

$$\|r_k\|/\|\hat{b}\| < \sqrt{\epsilon_M}, \tag{3.8}$$

where r_k is the (preconditioned) residual at the kth iteration, $\hat{b}$ is the preconditioned right-hand side and ϵ_M is the machine precision (machine epsilon), i.e., $\sqrt{\epsilon_M} \approx 10^{-8}$, since we use IEEE 754 double precision arithmetic. We also did tests with the alternative stopping criterion

$$\frac{\|r_k\|_2}{\sqrt{\|A\|_\infty \|A\|_1}\,\|x_k\|_2 + \|\hat{b}\|_2} \leq \sqrt{\epsilon_M},$$

which is the default criterion in ILUPACK [12], based on the analysis in [3]; see also [39]. Since the results were very similar and the full test cannot be done at each iteration, we only show results using the relative residual stopping criterion (3.8). The solver was stopped if no convergence is reached after 20 cycles of GMRES(50), i.e., after 1000 iterations.

The results are summarized in Table 3.7. More detailed results, including the relative residual and relative error norms, are given in Table 3.5 for XPABLO, Table 3.8 for ILUTP(10^{-2}), Table 3.9 for ILUTP(10^{-3}), and Table 3.10 for ILUTP(10^{-4}). For the XPABLO-based preconditioners we use the recommended parameters described in section 3.5. The block size limits were $minbs = 200$ and $maxbs = 1000$. The notation is the same as in previous tables, see section 3.7.1 for a detailed description. A dagger (†) indicates that the incomplete LU decomposition failed. A double dagger (‡) indicates that GMRES found the preconditioner to be too ill-conditioned. In Table 3.7, the bold numbers indicate the smallest execution time and the lowest iteration number in each row. The solving time is measured in seconds.

Table 3.7. Comparison of XPABLO and ILUTP Results

Matrix	ILUTP(10^{-2})		ILUTP(10^{-3})		ILUTP(10^{-4})		XPABLO	
	Time	Iter	Time	Iter	Time	Iter	Time	Iter
CAVITY16	1.6	150	0.958	9	1.19	**5**	**0.601**	49
CAVITY26	0.802	25	0.989	7	1.24	**4**	**0.61**	45
EX19	15.9	(350)	16.3	(800)	23.8	(650)	**6.66**	**252**
EX35	22.3	(250)	23.1	(550)	23.1	(500)	**2.48**	**28**
GARON1	**0.209**	22	0.286	9	0.369	**5**	0.333	37
GARON2	4.25	194	**1.93**	19	3.49	**8**	5.91	163
RAEFSKY2	9.47	(1000)	2.17	20	3.62	**7**	**0.665**	39
RAEFSKY3	76	(300)	71.9	(1000)	**28.9**	**20**	31.8	390
SHYY41	–	†	–	†	1.23	970	**0.112**	**6**
SHYY161	–	†	–	†	–	†	**9.14**	**19**
IGBT3	–	†	0.468	35	**0.332**	**13**	3.29	120
NMOS3	18.5	(800)	20.8	(1000)	**1.19**	**15**	4.99	76
BARRIER2-1	143.9	(850)	179.3	(500)	300.9	(1000)	**854.5**	**680**
PARA-4	297.6	(350)	265.5	(950)	461.1	(1000)	**519.0**	**210**
PARA-8	–	†	–	†	383.5	(1000)	**196.8**	**72**
OHNE2	–	‡	602.0	(550)	802.7	(1000)	1339.2	517
2D_54019_HIGHK	50	(650)	63.7	(1000)	**14.6**	**29**	49.4	177
3D_51448_3D	49.7	(100)	**15.6**	44	47.0	**16**	16.2	44
IBM_MATRIX_2	50.5	(100)	16.1	44	48.2	**16**	**15.2**	42
MATRIX_9	153.3	(650)	**129.0**	297	282.2	**25**	139.5	142
MATRIX-NEW_3	–	†	223.9	(1000)	199.1	**27**	**80.9**	93
LSQ_2D_1000	**0.024**	10	0.026	6	0.032	4	0.024	1
LSQ_2D_2000	**0.051**	12	0.063	7	0.085	**4**	0.084	15
LSQ_2D_5000	**0.148**	15	0.22	9	0.352	**5**	0.369	24
LSQ_2D_10000	**0.359**	18	0.591	11	0.995	**6**	1.04	34
LSQ_2D_50000	**2.87**	32	4.34	17	9.76	**10**	18.4	75
LSQ_2D_100000	**8.32**	41	10.8	21	24.6	**13**	88.4	111
LSQ_2D_200000	**24.1**	56	26.0	26	66.8	**16**	525.0	188
LSQ_2D_400000	71.2	88	**61.9**	33	134.4	**19**	2467.8	240
MPS_2D_10000	**0.181**	21	0.189	11	0.32	**7**	1.56	73
MPS_2D_50000	1.95	39	**1.56**	18	2.65	**11**	36.6	185
MPS_2D_100000	6.63	49	**4.02**	22	6.65	**13**	172.8	257
MPS_2D_200000	32.7	116	**10.0**	26	16.2	**16**	1019.8	418
MPS_2D_400000	593.2	(1000)	**25.8**	32	37.6	**19**	6889.9	759
LSQ_3D_10000	0.939	12	2.0	8	4.17	**5**	**0.779**	18
LSQ_3D_50000	13.5	19	45.5	11	130.6	**7**	**9.49**	26
LSQ_3D_100000	44.0	22	176.0	12	510.9	**8**	**33.2**	31
LSQ_3D_200000	136.1	60	606.8	14	1743.7	**9**	**127.9**	38
MPS_3D_10000	**0.242**	12	0.565	7	1.13	**5**	0.599	27
MPS_3D_50000	**2.96**	20	10.9	11	28.8	**7**	9.18	44
MPS_3D_100000	**9.33**	30	36.4	13	119.3	**8**	39.4	55
MPS_3D_200000	**55.0**	132	101.7	16	399.7	**10**	176.1	70

3.8 Discussion

The comparison of PABLO, TPABLO1, and XPABLO (see Table 3.6) shows that XPABLO is almost always the superior PABLO-based reordering for block Gauss–Seidel preconditioning. This is true for both robustness and speed. Moreover, this is not achieved using more memory, since the parameters and most importantly the block size limits are the same in all (T/X)PABLO-based experiments.

If we compare the XPABLO solve results in Table 3.5 with the direct solve results in Table 1.2, then the direct solver seems to be superior: The direct solver is faster in almost all cases and in most cases by a factor of two or more. An exception are the `LSQ_3D_`$*$ and `MPS_3D_`$*$ matrices. For these problems the XPABLO-based solver is in general faster. On the other hand, the XPABLO-based solver usually requires less memory than the direct solver. In the experiments the direct solver failed for six of the test problems. In all six cases the solver failed because the available memory was not sufficiently large. With the XPABLO-based approach the amount of available memory was sufficient for all test problems.

As observed before, for small problems it can happen that XPABLO finds only one block. Naturally, we recommend to use a direct solver is such a case. In general, a direct solver should be used if the order of the matrix is less than or equal to the maximum block size *maxbs*. Note that XPABLO could find more than one block for a matrix of order *maxbs*. Even then we recommend to use a direct solver.

As it can be observed in Table 3.7, the XPABLO-based preconditioners can perform better than the ILU-based ones in many cases, and in some cases much better. In some examples the XPABLO-based preconditioners perform better, although the iteration counts of the ILUTP-based preconditioners are lower. The reason for this is that times for finding and, more importantly, factorizing the preconditioner are not the same in both cases. Moreover, in many cases with ILUTP each iteration step is

more costly. With ILUTP we have to do a matrix-vector multiplication with A and a preconditioning step with the incomplete factorization in each iteration. Whereas, in the block Gauss-Seidel preconditioning we can employ an optimized preconditioned matrix-vector multiplication as discussed in section 3.6.

On the other hand, for the meshfree discretizations of Poisson's equation the XPABLO approach is not doing as well. For these problems ILU-based preconditioners seem to work very well, even compared to the results using a direct solver, cf. Table 1.2. For the two-dimensional problems (`LSQ_2D_`$*$ and `MPS_2D_`$*$) the direct solver is the best, by being both fast and robust, followed closely by ILUTP(10^{-3}). XPABLO, on the other hand, is not really competitive for these problems. For the three-dimensional problems (`LSQ_3D_`$*$ and `MPS_3D_`$*$) the situation is different. Here, ILUTP(10^{-2}) seems to be the best overall choice, being consistently faster than the direct solver. Moreover, the direct solver was not able to solve the largest problems, because the available memory was not sufficient. ILUTP(10^{-2}) did not suffer from these problems. Moreover, the XPABLO-based preconditioners are much more competitive for the three-dimensional problems than for the two-dimensional problems. We note that XPABLO is the fastest for the `LSQ_3D_`$*$ problems and not far behind ILUTP(10^{-2}) for the three-dimensional MPS discretizations.

While XPABLO is not always competitive with sophisticated ILU preconditioners like ILUTP, we have seen that XPABLO can perform better than ILUT for many CFD and semiconductor device simulation problems. XPABLO also gave encouraging results for the discretizations of Poisson's equation in the three-dimensional case.

Table 3.8. ILUTP(10^{-2}) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	1.6	150	$4.202 \cdot 10^{-8}$	$5.340 \cdot 10^{-7}$
CAVITY26	0.802	25	$1.000 \cdot 10^{-8}$	$4.039 \cdot 10^{-7}$
EX19	15.9	(350)	$3.799 \cdot 10^{+3}$	$5.850 \cdot 10^{+3}$
EX35	22.3	(250)	$9.834 \cdot 10^{+4}$	$3.043 \cdot 10^{+5}$
GARON1	0.209	22	$5.243 \cdot 10^{-9}$	$2.270 \cdot 10^{-7}$
GARON2	4.25	194	$3.082 \cdot 10^{-9}$	$1.689 \cdot 10^{-7}$
RAEFSKY2	9.47	(1000)	$1.095 \cdot 10^{+0}$	na
RAEFSKY3	76.0	(300)	$2.551 \cdot 10^{+2}$	na
SHYY41	–	†	–	–
SHYY161	–	†	–	–
IGBT3	–	†	–	–
NMOS3	18.5	(800)	$8.584 \cdot 10^{+5}$	na
BARRIER2-1	143.9	(850)	$1.488 \cdot 10^{+13}$	na
PARA-4	297.6	(350)	$7.076 \cdot 10^{+7}$	na
PARA-8	–	†	–	–
OHNE2	–	‡	–	–
2D_54019_HIGHK	50.0	(650)	$1.000 \cdot 10^{+0}$	na
3D_51448_3D	49.7	(100)	$1.001 \cdot 10^{+0}$	na
IBM_MATRIX_2	50.5	(100)	$1.009 \cdot 10^{+0}$	na
MATRIX_9	153.3	(650)	$2.547 \cdot 10^{+4}$	na
MATRIX-NEW_3	–	†	–	–
LSQ_2D_1000	0.024	10	$3.300 \cdot 10^{-6}$	$1.068 \cdot 10^{-9}$
LSQ_2D_2000	0.051	12	$7.394 \cdot 10^{-6}$	$9.326 \cdot 10^{-10}$
LSQ_2D_5000	0.148	15	$3.925 \cdot 10^{-5}$	$1.867 \cdot 10^{-9}$
LSQ_2D_10000	0.359	18	$1.229 \cdot 10^{-4}$	$2.338 \cdot 10^{-9}$
LSQ_2D_50000	2.87	32	$3.942 \cdot 10^{-4}$	$1.020 \cdot 10^{-9}$
LSQ_2D_100000	8.32	41	$6.889 \cdot 10^{-4}$	$9.232 \cdot 10^{-10}$
LSQ_2D_200000	24.1	56	$1.461 \cdot 10^{-3}$	$1.277 \cdot 10^{-9}$
LSQ_2D_400000	71.2	88	$2.330 \cdot 10^{-3}$	$1.323 \cdot 10^{-9}$
MPS_2D_10000	0.181	21	$1.384 \cdot 10^{-4}$	$1.224 \cdot 10^{-9}$
MPS_2D_50000	1.95	39	$5.252 \cdot 10^{-4}$	$1.017 \cdot 10^{-9}$
MPS_2D_100000	6.63	49	$1.038 \cdot 10^{-3}$	$1.311 \cdot 10^{-9}$
MPS_2D_200000	32.7	116	$6.359 \cdot 10^{-4}$	$2.952 \cdot 10^{-9}$
MPS_2D_400000	593.2	(1000)	$2.712 \cdot 10^{+0}$	$5.886 \cdot 10^{-5}$
LSQ_3D_10000	0.939	12	$3.407 \cdot 10^{-6}$	$2.472 \cdot 10^{-9}$
LSQ_3D_50000	13.5	19	$2.110 \cdot 10^{-5}$	$4.589 \cdot 10^{-9}$
LSQ_3D_100000	44.0	22	$3.142 \cdot 10^{-5}$	$3.721 \cdot 10^{-9}$
LSQ_3D_200000	136.1	60	$5.565 \cdot 10^{-5}$	$1.108 \cdot 10^{-8}$
MPS_3D_10000	0.242	12	$1.827 \cdot 10^{-5}$	$6.503 \cdot 10^{-9}$
MPS_3D_50000	2.96	20	$4.415 \cdot 10^{-5}$	$3.917 \cdot 10^{-9}$
MPS_3D_100000	9.33	30	$5.733 \cdot 10^{-5}$	$2.845 \cdot 10^{-9}$
MPS_3D_200000	55.0	132	$1.086 \cdot 10^{-4}$	$7.972 \cdot 10^{-9}$

Table 3.9. ILUTP(10^{-3}) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	0.958	9	$1.732 \cdot 10^{-8}$	$2.267 \cdot 10^{-7}$
CAVITY26	0.989	7	$1.017 \cdot 10^{-8}$	$4.671 \cdot 10^{-7}$
EX19	16.3	(800)	$1.803 \cdot 10^{+9}$	$9.515 \cdot 10^{+10}$
EX35	23.1	(550)	$3.341 \cdot 10^{+7}$	$3.237 \cdot 10^{+9}$
GARON1	0.286	9	$4.079 \cdot 10^{-9}$	$2.127 \cdot 10^{-7}$
GARON2	1.93	19	$1.575 \cdot 10^{-9}$	$2.541 \cdot 10^{-7}$
RAEFSKY2	2.17	20	$3.772 \cdot 10^{-9}$	na
RAEFSKY3	71.9	(1000)	$5.590 \cdot 10^{-3}$	na
SHYY41	–	†	–	–
SHYY161	–	†	–	–
IGBT3	0.468	35	$1.652 \cdot 10^{-6}$	na
NMOS3	20.8	(1000)	$4.681 \cdot 10^{+2}$	na
BARRIER2-1	179.3	(500)	$8.186 \cdot 10^{+1}$	na
PARA-4	265.5	(950)	$4.050 \cdot 10^{+7}$	na
PARA-8	–	†	–	–
OHNE2	602.0	(550)	$1.505 \cdot 10^{+1}$	na
2D_54019_HIGHK	63.7	(1000)	$8.610 \cdot 10^{-9}$	na
3D_51448_3D	15.6	44	$4.313 \cdot 10^{-11}$	na
IBM_MATRIX_2	16.1	44	$2.127 \cdot 10^{-12}$	na
MATRIX_9	129.0	297	$6.717 \cdot 10^{-9}$	na
MATRIX-NEW_3	223.9	(1000)	$2.610 \cdot 10^{-7}$	na
LSQ_2D_1000	0.026	6	$1.664 \cdot 10^{-6}$	$7.534 \cdot 10^{-10}$
LSQ_2D_2000	0.063	7	$5.227 \cdot 10^{-6}$	$1.107 \cdot 10^{-9}$
LSQ_2D_5000	0.22	9	$1.366 \cdot 10^{-5}$	$9.187 \cdot 10^{-10}$
LSQ_2D_10000	0.591	11	$2.812 \cdot 10^{-5}$	$7.768 \cdot 10^{-10}$
LSQ_2D_50000	4.34	17	$3.835 \cdot 10^{-4}$	$1.388 \cdot 10^{-9}$
LSQ_2D_100000	10.8	21	$6.911 \cdot 10^{-4}$	$1.010 \cdot 10^{-9}$
LSQ_2D_200000	26.0	26	$1.617 \cdot 10^{-3}$	$1.014 \cdot 10^{-9}$
LSQ_2D_400000	61.9	33	$1.491 \cdot 10^{-3}$	$6.337 \cdot 10^{-10}$
MPS_2D_10000	0.189	11	$1.854 \cdot 10^{-4}$	$2.789 \cdot 10^{-9}$
MPS_2D_50000	1.56	18	$3.036 \cdot 10^{-4}$	$9.645 \cdot 10^{-10}$
MPS_2D_100000	4.02	22	$3.941 \cdot 10^{-4}$	$6.888 \cdot 10^{-10}$
MPS_2D_200000	10.0	26	$5.201 \cdot 10^{-4}$	$9.104 \cdot 10^{-10}$
MPS_2D_400000	25.8	32	$5.917 \cdot 10^{-4}$	$9.799 \cdot 10^{-10}$
LSQ_3D_10000	2.0	8	$7.102 \cdot 10^{-7}$	$8.226 \cdot 10^{-10}$
LSQ_3D_50000	45.5	11	$3.074 \cdot 10^{-6}$	$1.053 \cdot 10^{-9}$
LSQ_3D_100000	176.0	12	$1.991 \cdot 10^{-5}$	$4.117 \cdot 10^{-9}$
LSQ_3D_200000	606.8	14	$3.216 \cdot 10^{-5}$	$3.588 \cdot 10^{-9}$
MPS_3D_10000	0.565	7	$6.883 \cdot 10^{-6}$	$3.735 \cdot 10^{-9}$
MPS_3D_50000	10.9	11	$9.483 \cdot 10^{-6}$	$1.273 \cdot 10^{-9}$
MPS_3D_100000	36.4	13	$3.308 \cdot 10^{-5}$	$2.379 \cdot 10^{-9}$
MPS_3D_200000	101.7	16	$5.606 \cdot 10^{-5}$	$2.110 \cdot 10^{-9}$

Table 3.10. ILUTP(10^{-4}) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16_PHD	1.19	5	$1.347 \cdot 10^{-9}$	$4.110 \cdot 10^{-8}$
CAVITY26_PHD	1.24	4	$2.904 \cdot 10^{-9}$	$1.613 \cdot 10^{-7}$
EX19	23.8	(650)	$1.800 \cdot 10^{+1}$	$1.436 \cdot 10^{+2}$
EX35	23.1	(500)	$1.797 \cdot 10^{+12}$	$3.018 \cdot 10^{+12}$
GARON1	0.369	5	$1.813 \cdot 10^{-10}$	$6.128 \cdot 10^{-9}$
GARON2	3.49	8	$3.275 \cdot 10^{-10}$	$2.335 \cdot 10^{-8}$
RAEFSKY2	3.62	7	$7.545 \cdot 10^{-10}$	na
RAEFSKY3	28.9	20	$3.730 \cdot 10^{-10}$	na
SHYY41	1.23	970	$2.515 \cdot 10^{-6}$	$3.469 \cdot 10^{+0}$
SHYY161	–	†	–	–
IGBT3	0.332	13	$1.584 \cdot 10^{-7}$	na
NMOS3	1.19	15	$1.638 \cdot 10^{-7}$	na
BARRIER2-1	300.9	(1000)	$1.644 \cdot 10^{+0}$	na
PARA-4	461.1	(1000)	$1.245 \cdot 10^{+0}$	na
PARA-8	383.5	(1000)	$5.763 \cdot 10^{-3}$	na
OHNE2	802.7	(1000)	$5.945 \cdot 10^{+0}$	na
2D_54019_HIGHK	14.6	29	$6.438 \cdot 10^{-11}$	na
3D_51448_3D	47.0	16	$2.134 \cdot 10^{-11}$	na
IBM_MATRIX_2	48.2	16	$1.342 \cdot 10^{-10}$	na
MATRIX_9	282.2	25	$1.491 \cdot 10^{-7}$	na
MATRIX-NEW_3	199.1	27	$1.970 \cdot 10^{-10}$	na
LSQ_2D_1000	0.032	4	$2.607 \cdot 10^{-7}$	$1.088 \cdot 10^{-10}$
LSQ_2D_2000	0.085	4	$1.084 \cdot 10^{-5}$	$2.748 \cdot 10^{-9}$
LSQ_2D_5000	0.352	5	$2.423 \cdot 10^{-5}$	$1.868 \cdot 10^{-9}$
LSQ_2D_10000	0.995	6	$9.720 \cdot 10^{-5}$	$3.222 \cdot 10^{-9}$
LSQ_2D_50000	9.76	10	$2.202 \cdot 10^{-4}$	$1.110 \cdot 10^{-9}$
LSQ_2D_100000	24.6	13	$3.037 \cdot 10^{-4}$	$4.884 \cdot 10^{-10}$
LSQ_2D_200000	66.8	16	$4.762 \cdot 10^{-4}$	$3.775 \cdot 10^{-10}$
LSQ_2D_400000	134.4	19	$1.339 \cdot 10^{-3}$	$4.766 \cdot 10^{-10}$
MPS_2D_10000	0.32	7	$1.813 \cdot 10^{-5}$	$4.185 \cdot 10^{-10}$
MPS_2D_50000	2.65	11	$1.453 \cdot 10^{-4}$	$6.785 \cdot 10^{-10}$
MPS_2D_100000	6.65	13	$4.121 \cdot 10^{-4}$	$8.168 \cdot 10^{-10}$
MPS_2D_200000	16.2	16	$1.383 \cdot 10^{-4}$	$3.875 \cdot 10^{-10}$
MPS_2D_400000	37.6	19	$2.823 \cdot 10^{-4}$	$7.434 \cdot 10^{-10}$
LSQ_3D_10000	4.17	5	$5.906 \cdot 10^{-7}$	$9.249 \cdot 10^{-10}$
LSQ_3D_50000	130.6	7	$2.326 \cdot 10^{-6}$	$1.173 \cdot 10^{-9}$
LSQ_3D_100000	510.9	8	$5.681 \cdot 10^{-6}$	$1.861 \cdot 10^{-9}$
LSQ_3D_200000	1743.7	9	$1.672 \cdot 10^{-5}$	$2.919 \cdot 10^{-9}$
MPS_3D_10000	1.13	5	$7.602 \cdot 10^{-7}$	$5.230 \cdot 10^{-10}$
MPS_3D_50000	28.8	7	$5.605 \cdot 10^{-6}$	$1.011 \cdot 10^{-9}$
MPS_3D_100000	119.3	8	$3.242 \cdot 10^{-5}$	$2.991 \cdot 10^{-9}$
MPS_3D_200000	399.7	10	$1.597 \cdot 10^{-5}$	$7.703 \cdot 10^{-10}$

CHAPTER 4

OVERLAPPING PARTITIONING

Additive and multiplicative Schwarz preconditioners are known to be efficient for discretizations of (elliptic) partial differential equation problems involving a geometric domain. They are based on decomposing the domain into a set of overlapping subdomains in such a way that the restrictions of the original problem to each subdomain are solvable; see, e.g., [44], [47], [48]. In our problem $Ax = b$, cf. equation (1.1), we do not assume knowledge about the geometric domain, in fact the problem may not involve a geometric domain at all. Recall that we assume $A \in \mathbb{R}^{n \times n}$ to be a general nonsingular square matrix, i.e., we do not assume any symmetry or definiteness. Algebraic Schwarz methods are a generalization of the (geometric) Schwarz methods that work without an underlying geometric domain by restricting the linear operator A to overlapping subsets of the variables.

An important issue in applying algebraic Schwarz methods is the question how to determine such overlapping subsets. In terms of the graph $\mathcal{G} = \mathcal{G}(A)$ of the matrix A we have to find overlapping subgraphs ("blocks") of $\mathcal{G}$. The OBGP (Overlapping Blocks by Growing a Partition) algorithm is a new approach to compute such overlapping subgraphs. As implied by its name, it works by taking an existing non-overlapping partition and grows each block to include some vertices from other blocks.

In section 4.1 we briefly introduce Schwarz methods. In section 4.2 we introduce the OBGP algorithm. In section 4.2.2 we discuss implementation challenges and how OBGP can be implemented to work in an efficient way. In section 4.3 we give a detailed analysis of the time complexity of OBGP. We finally give results of numerical experiments in section 4.4.

4.1 Schwarz Methods and Preconditioners

As a starting point of our introduction to Schwarz methods we will first reformulate the block Gauss–Seidel method in section 4.1.1. As we will see in section 4.1.2, the block Gauss–Seidel method and the multiplicative Schwarz method are related. To make it easier to observe this relation we will use notation typically known from domain decomposition to introduce the block Gauss–Seidel method; for comparison, see, e.g., [6], [41, pp. 465ff].

Let $A \in \mathbb{R}^{n\times n}$ be the matrix of the linear system $Ax = b$ and $V = \{1, \ldots, n\}$ the set of vertices in the directed graph $\mathcal{G}(A)$ associated with A; see section 2.4 for more details on the graph of a matrix.

We let the nodes $1, \ldots, n$ in V correspond to the unit vectors $e_1, \ldots, e_n$ by associating node k with unit vector e_k. Furthermore, we associate a node set S with the linear subspaces spanned by the unit vectors associated with the nodes in S. Then V is associated with $\mathbb{R}^n$ and a subset $S = \{s_1, \ldots, s_m\} \subset V$ is associated with the linear subspace $\text{span}\{e_{s_1}, \ldots, e_{s_m}\} \subset \mathbb{R}^n$.

Definition 4.1: Let $S = \{s_1, \ldots, s_m\} \subset V$ be an ordered subset of the node set $V = \{1, \ldots, n\}$. The *restriction operator* $R_S \in \mathbb{R}^{m\times n}$ corresponding to S onto the

subspace associated with S is defined by

$$(R_S)_{ij} := \begin{cases} 1 & \text{if } s_i = j, \\ 0 & \text{otherwise.} \end{cases}$$

The transpose R_S^T of a restriction operator R_S is the corresponding *prolongation operator*. $\diamond$

Note that row k, $k = 1, \ldots, m$ of R_S is just the s_kth row of I_n.

Example 4.2: Let $V = \{1, \ldots, 5\}$ and let $S = \{1, 2, 4\}$ and $T = \{4, 1, 2\}$. Then

$$R_S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \text{and} \qquad R_T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Note that $S = T$ but $R_S \neq R_T$.

Using MATLAB notation Definition 4.1 is equivalent to

$$R_S : \mathbb{R}^n \to \mathbb{R}^m,$$

$$y \mapsto y(S).$$

In most cases we consider restrictions to an element of a cover or partition. In order to simplify our notation we introduce the following abbreviation:

Definition 4.3: Let $\{V_i\}_{i=1,\ldots,q}$ be a cover of V. Define $R_i := R_{V_i} \in \mathbb{R}^{n_i \times n}$. $\diamond$

Remark 4.4: Every restriction operator can also be written as $R_i = E_i P_i$, where $E_i = [I_{n_i}\ 0] \in \mathbb{R}^{n_i \times n}$ and $P_i \in \mathbb{R}^{n \times n}$ is a permutation matrix depending on V_i. Then

$$R_i^T R_i = (E_i P_i)^T (E_i P_i) = P_i^T E_i^T E_i P_i = P_i^T \begin{bmatrix} I_{n_i} & 0 \\ 0 & 0 \end{bmatrix} P_i$$

$$= \operatorname{diag}\big(\chi_{V_i}(1), \ldots, \chi_{V_i}(n)\big),$$

where χ_{V_i}, $i = 1, \ldots, q$, is the characteristic function of the set V_i.

Thus, $R_i^T R_i$ is a diagonal matrix with 1 on the diagonal in row k if $k \in V_i$ and 0 otherwise. Hence, $R_i^T R_i$ is the orthogonal projector onto the subspace associated with V_i.

We use the subscript notation also to indicate submatrices and subvectors:

Definition 4.5: Let $\{V_i\}_{i=1,\ldots,q}$ be a cover of the node set $V = \{1, \ldots, n\}$. The *submatrix* A_{ij} of a matrix $A \in \mathbb{R}^{n\times n}$ is defined by

$$A_{ij} := R_i A R_j^T \in \mathbb{R}^{n_i \times n_j}.$$

Let $S = \{s_1, \ldots, s_m\} \subset V$ be an ordered subset of V. The submatrix A_S is defined by

$$A_S := R_S A R_S^T \in \mathbb{R}^{m\times m}.$$

For simplicity, we will write A_i for $A_{ii} = A_{V_i}$ from now on. The *subvector* z_i of a vector $z \in \mathbb{R}^n$ is defined by

$$z_i := R_i z \in \mathbb{R}^{n_i}. \qquad \diamond$$

Remark 4.6: The diagonal entries of A_i, $i = 1, \ldots, q$ are diagonal entries of A. As before, we can write R_i as $R_i = E_i P_i$, where $E_I = [I_{n_i}\ 0]$ and P_i is a permutation matrix. Then

$$A_i = R_i A R_i^T = E_i P_i A P_i^T E_i^T = E_i (P_i A P_i^T) E_i^T.$$

The matrix $P_i A P_i^T$ has the same diagonal entries as A as it is a symmetric permutation of A. The matrix A_i is the $n_i \times n_i$ upper left part of $P_i A P_i^T$ and hence its diagonal entries are diagonal entries of A.

4.1.1 The Block Gauss–Seidel Method

Let $\{V_i\}_{i=1,\dots,q}$ be a partition of V, i.e., the subsets V_i are pairwise disjoint. Let P be the permutation matrix which permutes $(1,\dots,n)^T$ into $(v_1^{(1)},\dots,v_{n_1}^{(1)},v_1^{(2)},\dots,v_{n_q}^{(q)})^T$. Note that

$$P = \begin{bmatrix} R_1 \\ \vdots \\ R_q \end{bmatrix}. \tag{4.1}$$

Then the permuted problem $(PAP^T)Px = Pb$ has a $q \times q$ block structure as follows

$$\begin{bmatrix} A_1 & A_{12} & \cdots & A_{1q} \\ A_{21} & A_2 & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{q1} & \cdots & \cdots & A_q \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{bmatrix}. \tag{4.2}$$

The "blocks" A_{ij} in (4.2) are the same as the submatrices A_{ij} in Definition 4.5. The same holds for the subvectors x_i and b_i,

Example 4.7: Let $A \in \mathbb{R}^{5\times 5}$, $q = 2$, $V_1 = \{1,2,4\}$ and $V_2 = \{5,3\}$. Then

$$R_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

$$R_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

and the permuted matrix PAP^T has the form

$$PAP^T = \begin{bmatrix} A_1 & A_{12} \\ A_{21} & A_2 \end{bmatrix} = \left[\begin{array}{ccc:cc} a_{11} & a_{12} & a_{14} & a_{15} & a_{13} \\ a_{21} & a_{22} & a_{24} & a_{25} & a_{23} \\ a_{41} & a_{42} & a_{44} & a_{45} & a_{43} \\ \hdashline a_{51} & a_{52} & a_{54} & a_{55} & a_{53} \\ a_{31} & a_{32} & a_{34} & a_{35} & a_{33} \end{array}\right].$$

Let $\hat{A} := PAP^T$, $\hat{x} = Px$ and $\hat{b} = Pb$, i.e.,

$$\hat{A} = \begin{bmatrix} A_1 & A_{12} & \cdots & A_{1q} \\ A_{21} & A_2 & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{q1} & \cdots & \cdots & A_q \end{bmatrix}, \quad \hat{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{bmatrix}.$$

Then $\hat{A}\hat{x} = \hat{b}$ is just the permuted problem (4.2). Using the block structure shown in (4.2), we split $\hat{A}$ into $\hat{A} = D - L - U$, where D is the block diagonal part $D = \operatorname{diag}(A_1, \ldots, A_q)$, L the block lower triangular part with $L_{ij} = -A_{ij}$ for $i > j$, and U the block upper triangular part $U_{ij} = -A_{ij}$ for $i < j$. Both L and U have zero block diagonal.

Using this notation, the block Gauss–Seidel iteration is defined by

$$(D - L)\hat{x}^{(k+1)} = U\hat{x}^{(k)} + \hat{b}. \tag{4.3}$$

The new iterate $\hat{x}^{(k+1)}$ from (4.3) can be computed blockwise, resulting in the iteration

$$x_i^{(k+1)} = A_i^{-1}\Big(b_i - \sum_{j<i} A_{ij}x_j^{(k+1)} - \sum_{j>i} A_{ij}x_j^{(k)}\Big), \quad i = 1, \ldots, q, \tag{4.4}$$

where $x_i^{(k)}$ denotes the ith "block" of the kth iterate $\hat{x}^{(k)}$. With $x_i^{(k)}$ we also denote the restriction $R_i x^{(k)}$ of $x^{(k)} = P^{-1}\hat{x}^{(k)}$. Note that the ith block of $\hat{x}^{(k)}$ is exactly the same as the restriction $R_i x^{(k)}$ of vector $x^{(k)}$. Therefore, we do not distinguish between these two interpretations of $x_i^{(k)}$.

Now we want to rewrite the block Gauss–Seidel iteration in a way that is useful for defining the multiplicate Schwarz method. Let

$$\xi^{(k,i)} = \big(x_1^{(k+1)}, \ldots, x_i^{(k+1)}, x_{i+1}^{(k)}, \ldots, x_q^{(k)}\big)^T$$

be the ith intermediate iterate between $x^{(k)}$ and $x^{(k+1)}$, i.e., $\xi^{(k,0)} = x^{(k)}$ and $\xi^{(k,q)} = x^{(k+1)}$. We use our usual restriction notation $\xi_j^{(k,i)} = R_j\xi^{(k,i)}$ to denote the restricted

intermediate iterates. The vectors $\xi^{(k,i)}$ and $\xi^{(k,i-1)}$ differ only in the ith block $\xi_i^{(k,i)} = x_i^{(k+1)} \neq x_i^{(k)} = \xi_i^{(k,i-1)}$ and hence we can express $\xi_j^{(k,i)}$ as

$$\xi_j^{(k,i)} = \begin{cases} x_i^{(k+1)} & \text{if } i = j, \\ \xi_j^{(k,i-1)} & \text{otherwise.} \end{cases} \tag{4.5}$$

Then

$$\begin{aligned} \xi_i^{(k,i)} = x_i^{(k+1)} &\overset{(4.4)}{=} A_i^{-1}\Big(b_i - \sum_{j<i} A_{ij}\xi_j^{(k,i-1)} - \sum_{j>i} A_{ij}\xi_j^{(k,i-1)}\Big) \\ &= A_i^{-1}\Big(b_i + A_i\xi_i^{(k,i-1)} - \sum_{j=1}^{q} A_{ij}\xi_j^{(k,i-1)}\Big) \\ &= \xi_i^{(k,i-1)} + A_i^{-1}\Big(b_i - \sum_{j=1}^{q} A_{ij}\xi_j^{(k,i-1)}\Big). \end{aligned}$$

We can rewrite (4.5) as

$$\xi_j^{(k,i)} = \begin{cases} \xi_i^{(k,i-1)} + A_i^{-1}\Big(b_i - \sum_{j=1}^{q} A_{ij}\xi_j^{(k,i-1)}\Big) & \text{if } i = j, \\ \xi_j^{(k,i-1)} & \text{otherwise.} \end{cases}$$

It follows that

$$\xi^{(k,i)} = \xi^{(k,i-1)} + R_i^T A_i^{-1}\Big(b_i - \sum_{j=1}^{q} A_{ij}\xi_j^{(k,i-1)}\Big) \tag{4.6}$$

$$= \xi^{(k,i-1)} + R_i^T A_i^{-1} R_i\big(b - A\xi^{(k,i-1)}\big). \tag{4.7}$$

Definition 4.8: Let $S \subset V$ be any ordered subset of V. We define Π_S by $\Pi_S := R_S^T A_S^{-1} R_S A$. For a cover $\{V_i\}_{i=1,\dots,q}$ of V we define $\Pi_i := \Pi_{V_i}$. $\diamond$

To derive the stationary iteration form of the block Gauss–Seidel method, we look at the errors. Let $d^{(k,i)} = A^{-1}b - \xi^{(k,i)}$ be the error of the intermediate iterates.

We can compute $d^{(k,i)}$ by

$$\begin{aligned} d^{(k,i)} = A^{-1}b - \xi^{(k,i)} &= A^{-1}b - \xi^{(k,i-1)} - R_i^T A_i^{-1} R_i \big(b - A\xi^{(k,i-1)}\big) \\ &= (A^{-1}b - \xi^{(k,i-1)}) - R_i^T A_i^{-1} R_i A \big(A^{-1}b - \xi^{(k,i-1)}\big) \\ &= (I - R_i^T A_i^{-1} R_i A)(A^{-1}b - \xi^{(k,i-1)}) \\ &= (I - R_i^T A_i^{-1} R_i A) d^{(k,i-1)} \\ &= (I - \Pi_i) d^{(k,i-1)}. \end{aligned}$$

Let

$$Q_i = (I - \Pi_i) \cdots (I - \Pi_1) \quad \text{and} \quad Q = Q_q. \tag{4.8}$$

Then the block Gauss–Seidel errors

$$e^{(k)} = A^{-1}b - x^{(k)} \tag{4.9}$$

satisfy the relation

$$e^{(k+1)} = Q e^{(k)}. \tag{4.10}$$

Combining (4.9) and (4.10) we get

$$x^{(k+1)} = A^{-1}b - e^{(k+1)} = A^{-1}b - Q e^{(k)} = A^{-1}b - Q(A^{-1}b - x^{(k)}),$$

from which we immediately get a formulation of the block Gauss–Seidel method as a stationary iteration

$$x_{\mathrm{GS}}^{(k+1)} = Q x^{(k)} + (I - Q) A^{-1} b. \tag{4.11}$$

Note that this iteration formula can not be used directly since $A^{-1}b$ is not known. However, there is a recursive procedure to compute $z = (I - Q)A^{-1}v$ for some vector v in which A^{-1} cancels; see also [41]. For $i \geq 2$ we have

$$I - Q_i = I - (I - \Pi_i) Q_{i-1} = (I - Q_{i-1}) + \Pi_i - \Pi_i (I - Q_{i-1}) \tag{4.12}$$

and

$$\begin{aligned}(I - Q_i)A^{-1} &= (I - Q_{i-1})A^{-1} + \Pi_i A^{-1} - \Pi_i (I - Q_{i-1})A^{-1} \\ &= (I - Q_{i-1})A^{-1} + R_i^T A_i^{-1} R_i - R_i^T A_i^{-1} R_i A (I - Q_{i-1})A^{-1}.\end{aligned} \tag{4.13}$$

Let $z_i = (I - Q_i)A^{-1}v$, $i = 1, \ldots, q$. Then

$$\begin{aligned}z_1 &= (I - Q_1)A^{-1}v = \Pi_1 A^{-1} v = R_1^T A_1^{-1} R_1 v \\ \text{and} \quad z_i &= z_{i-1} + R_i^T A_i^{-1} R_i (v - A z_{i-1}) \quad \text{for } i \geq 2.\end{aligned} \tag{4.14}$$

Formula (4.14) yields a recursive procedure to compute $z = z_q$.

The stationary iteration form of the block Gauss–Seidel method is usually derived in a different way; see, e.g., [41, pp. 110ff]: Assume we split A as $A = D - L - U$ into the block diagonal part D, the block lower triangular part $-L$, and the block upper triangular part $-U$. Both L and U have a zero block diagonal, cf. Theorem 3.15. Moreover, given a matrix splitting $A = M - N$, a stationary iteration can be defined by

$$x^{(k+1)} = M^{-1} N x^{(k)} + M^{-1} b.$$

For the block Gauss–Seidel iteration $M = D - L$ and $N = M - A = U$, i.e., M is the block lower triangular part (including diagonal) of A. This yields the block Gauss–Seidel iteration

$$x^{(k+1)} = (D - L)^{-1} U x^{(k)} + (D - L)^{-1} b,$$

which is the iteration given by (4.3); cf. (4.11).

4.1.2 Algebraic Multiplicative Schwarz Method

The Schwarz alternating method solves a partial differential equation on a domain Ω which is the union of two (or more) overlapping subdomains. It works by solving

the equation alternatingly on the subdomains. In each local solve, the Dirichlet boundary conditions on the artificial interface inside Ω are given by the latest available approximation of the global solution. This new local solution is then used to update the approximation for the global solution; see, e.g., [44], [47], [48].

The multiplicative Schwarz method is an abstract version of the original alternating method applied to the solution of linear systems. The basic ideas are still the same. We alternatingly use local solves to update the approximation of the global solution. If we use the multiplicative Schwarz method purely algebraically, we also refer to it as the algebraic multiplicative Schwarz method, which is the focus of this chapter. For the convergence theory of algebraic multiplicative Schwarz methods we refer to the literature; see, e.g., [6].

Let $x^{(k)}$ be the kth iterate of the (algebraic) multiplicative Schwarz method and let $\xi^{(k,i)}$ be the ith intermediate iterate after the update stemming from the local solve on the ith subdomain, i.e., $\xi^{(k,0)} = x^{(k)}$ and $\xi^{(k,q)} = x^{(k+1)}$, where q is the number of subdomains. Here the (overlapping) subdomains are the linear subspaces associated with a cover $\{V_i\}_{i=1,\dots,q}$ of $\{1,\dots,n\}$, where n is the order of our problem $Ax = b$. The multiplicative Schwarz method computes $\xi^{(k,i)}$ by

$$\xi^{(k,i)} := \xi^{(k,i-1)} + R_i^T A_i^{-1} R_i \big(b - A\xi^{(k,i-1)}\big), \quad i = 1,\dots,q, \tag{4.15}$$

where A_i and R_i are defined as before. Note that this computation is the same as the computation of $\xi^{(k,i)}$ in (4.7). Therefore, the multiplicative Schwarz method without overlap is just the block Gauss–Seidel method, i.e., the multiplicative Schwarz method can be understood as an extension of the block Gauss–Seidel method, which allows for overlapping blocks.

We can derive from this the stationary multiplicative Schwarz iteration in exactly the same way we used for the block Gauss–Seidel method, i.e., the iteration

is

$$x_{\mathrm{MS}}^{(k+1)} = Qx^{(k)} + (I - Q)A^{-1}b, \tag{4.16}$$

where Q is defined by $Q = (I - \Pi_q) \cdots (I - \Pi_1)$, like we did in (4.8). As in the block Gauss–Seidel case we can not use the iteration formula (4.16) directly, as we do not know $A^{-1}b$. The recursive procedure in (4.14) works in the same way for the multiplicative Schwarz method.

From the stationary iteration we can easily derive the multiplicative Schwarz preconditioner. The application of the multiplicative Schwarz preconditioner to a vector $v \in \mathbb{R}^n$ consists of computing one iteration step of the multiplicative Schwarz method using $b = v$ as the right hand side and $x^{(0)} = 0$ as the initial approximation. Therefore,

$$M_{\mathrm{MS}}^{-1} = (I - Q)A^{-1}. \tag{4.17}$$

The (left) preconditioned matrix-vector multiplication $M_{\mathrm{MS}}^{-1}Av$, $v \in \mathbb{R}^n$, can then be computed by

$$M_{\mathrm{MS}}^{-1}Av = (I - Q)v. \tag{4.18}$$

As discussed before, we dot not use (4.17) or (4.18) directly to obtain the result of a precondition step or a preconditioned matrix-vector multiplication, respectively. Algorithm 4.1 computes the application of a multiplicative Schwarz preconditioner; the algorithm is based on the recursion (4.14). Algorithm 4.2 computes a preconditioned matrix-vector multiplication. The recursion is based on (4.12) in the same way (4.14) is based on (4.13).

4.1.3 Connectivity in the Multiplicative Schwarz Method

One might think that there might be problems with covers that have non-connected sets. As we show now, this is not a problem, since there is a cover which is equivalent (in a sense made precise later), which only has connected sets. We denote

1: **input:** vector v
2: **output:** $z := M^{-1}v \qquad \{z := (I - Q)A^{-1}v\}$
3: $z := 0$
4: **for** $i = 1, \ldots, q$ **do**
5: $\quad z := z + (R_i^T A_i^{-1} R_i)(v - Az)$
6: **end for**

Algorithm 4.1. Application of a Multiplicative Schwarz Preconditioner

1: **input:** vector v
2: **output:** $z := M^{-1}Av \qquad \{z := (I - Q)v\}$
3: $z := 0$
4: **for** $i = 1, \ldots, q$ **do**
5: $\quad z := z + (R_i^T A_i^{-1} R_i A)(v - z)$
6: **end for**

Algorithm 4.2. Multiplicative Schwarz Preconditioned Matrix-Vector Multiplication

the multiplicative Schwarz method based on the cover $\{V_i\}_{i=1,\ldots,q}$ by $\mathrm{MS}(\{V_i\}) = \mathrm{MS}(V_1, \ldots, V_q)$. If we apply the multiplicative Schwarz method for two different covers, we may end up with two different iterations. Let MS_1 and MS_2 be multiplicative Schwarz iterations based on two different covers. We call MS_1 and MS_2 equivalent if their iteration matrices Q_{MS_1} and Q_{MS_2} are equal; here Q_{MS_1} and Q_{MS_2} refer to Q in (4.16). This definition of equivalence implies that $x_{\mathrm{MS}_1}^{(k+1)}$ and $x_{\mathrm{MS}_2}^{(k+1)}$ computed by (4.16) are equal for all choices of $x^{(k)}$ and b.

In preparation for Theorem 4.10, we first present an auxiliary result.

Lemma 4.9: Let $\mathcal{G}(A) = (V, E)$ be the digraph of a matrix $A \in \mathbb{R}^{n \times n}$. Let $S, U \subset V$ be node sets. Then

$$R_S A R_U^T = 0 \text{ and } R_U A R_S^T = 0 \quad \Leftrightarrow \quad \mathrm{inc}(S, U) = \emptyset.$$

Proof: The submatrix $R_S A R_U^T$ is the submatrix of A computed by selecting rows according to S and columns according to U, or in MATLAB notation we have

$R_S A R_U^T = A(S,U)$. Therefore, $R_S A R_U^T$ and $R_U A R_S^T$ both equal to zero is equivalent to A having no nonzero matrix entry a_{ij} such that $i \in S$ and $j \in U$ or $i \in U$ and $j \in S$. In terms of the digraph of A, this is equivalent to having no edge between a node in S and a node in U (regardless of the direction). Finally, we have no edge between S and U if and only if $\mathrm{inc}(S,U) = \emptyset$. □

Theorem 4.10: Let $\mathcal{G}(A) = (V,E)$ be the digraph of a matrix $A \in \mathbb{R}^{n\times n}$. Let $\{V_i\}_{i=1,\dots,q}$ be a cover of V. Let $\{W_j\}_{j=1,\dots,r}$ be a refinement of $\{V_i\}$ for which there exists a surjective map $\kappa : \{1,\dots,r\} \to \{1,\dots,q\}$ such that (a) κ is monotonically increasing, (b) $W_j \cap W_k = \emptyset$ if $\kappa(j) = \kappa(k)$, (c) the connected components of $\mathcal{G}|_{W_j}$ are connected components of $\mathcal{G}|_{V_{\kappa(j)}}$, and (d) $\bigcup_{j\in\kappa^{-1}(i)} W_j = V_i$ for all $i = 1,\dots,q$. Then the two multiplicative Schwarz methods $\mathrm{MS}(\{V_i\})$ and $\mathrm{MS}(\{W_j\})$ are equivalent.

Proof: Let $\Pi_i = R_{V_i}^T A_{V_i}^{-1} R_{V_i} A$, $i = 1,\dots,q$, and $\Pi'_j = R_{W_j}^T A_{W_j}^{-1} R_{W_j} A$, $j = 1,\dots,r$. Furthermore, let $Q = (I - \Pi_q)\cdots(I - \Pi_1)$ and $Q' = (I - \Pi'_r)\cdots(I - \Pi'_1)$. We have to show $Q = Q'$ to prove equivalence of $\mathrm{MS}(\{V_i\})$ and $\mathrm{MS}(\{W_j\})$. Since by (a) the set $\kappa^{-1}(i)$, $i = 1,\dots,q$, is a (nonempty) contiguous subset of $\{1,\dots,r\}$, we can show $Q = Q'$ by showing

$$I - \Pi_i = \prod_{j\in\kappa^{-1}(i)} (I - \Pi'_j) \quad \text{for all } i = 1,\dots,q. \tag{4.19}$$

From now on, we assume $i \in \{1,\dots,q\}$ to be fixed and without loss of generality we assume $\kappa^{-1}(i) = \{1,\dots,m\}$. Let $\mathcal{G}' = \mathcal{G}|_{V_i}$. For any $j,k \in \kappa^{-1}(i)$ with $j \neq k$ we know by assumption (b) that $W_j \cap W_k = \emptyset$. Together with (c) this implies that $\mathrm{inc}(W_j, W_k) = \emptyset$ and hence by Lemma 4.9 we have $R_{W_j} A R_{W_k}^T = 0$. Therefore,

$$\Pi'_j \Pi'_k = R_{W_j}^T A_{W_j}^{-1} \underbrace{R_{W_j} A R_{W_k}^T}_{=0} A_{W_k}^{-1} R_{W_k} A = 0.$$

Thus (4.19) is equivalent to

$$\Pi_i = \sum_{j \in \kappa^{-1}(i)} \Pi'_j = \sum_{j=1}^{m} \Pi'_j \quad \text{for all } i = 1, \ldots, q.$$

From (c) follows that in $\mathcal{G}'$ there is no edge between any two W_j and W_k with $j, k \in \kappa^{-1}(i)$, $j \neq k$. Assumption (d) states that $V_i = \bigcup_{j=1,\ldots,m} W_j$ and assumption (b) says that this union is disjoint, i.e., $W_j \cap W_k = \emptyset$ for $j \neq k$. Putting all of this together, we can find a permutation matrix P such that

$$A_{V_i} = P \begin{bmatrix} A_{W_1} & & 0 \\ & \ddots & \\ 0 & & A_{W_m} \end{bmatrix} P^T.$$

Hence,

$$\begin{aligned} \Pi_i = R_{V_i}^T A_{V_i}^{-1} R_{V_i} &= R_{V_i}^T P \begin{bmatrix} A_{W_1}^{-1} & & 0 \\ & \ddots & \\ 0 & & A_{W_m}^{-1} \end{bmatrix} P^T R_{V_i} \\ &= R_{V_i}^T P \begin{bmatrix} A_{W_1}^{-1} & 0 \\ 0 & 0 \end{bmatrix} P^T R_{V_i} + \cdots + R_{V_i}^T P \begin{bmatrix} 0 & 0 \\ 0 & A_{W_m}^{-1} \end{bmatrix} P^T R_{V_i} \\ &= R_{W_1}^T A_{W_1}^{-1} R_{W_1} + \cdots + R_{W_m}^T A_{W_m}^{-1} R_{W_m} \\ &= \Pi'_1 + \cdots + \Pi'_m. \end{aligned}$$

□

The monotonicity of κ is a necessary condition for Theorem 4.10 as we will show in the following example:

Example 4.11: Consider the 3×3 matrix

$$A = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 2 \end{bmatrix}$$

with the cover $V_1 = \{1, 2\}$, $V_2 = \{2, 3\}$. Then

$$R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad A_1^{-1} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix},$$

$$R_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}, \quad A_2^{-1} = \begin{bmatrix} 0.5 & -0.25 \\ 0 & 0.5 \end{bmatrix}.$$

This yields

$$\Pi_1 = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot A = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & 0.5 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\Pi_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.5 & -0.25 \\ 0 & 0 & 0.5 \end{bmatrix} \cdot A = \begin{bmatrix} 0 & 0 & 0 \\ -0.25 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}$$

and

$$Q = (I - \Pi_2)(I - \Pi_1) = \begin{bmatrix} 0 & 0 & -0.5 \\ 0 & 0 & -0.125 \\ 0 & 0 & 0.25 \end{bmatrix}.$$

Let $W_1 = \{1\}$, $W_2 = \{2\}$ and $W_3 = \{2, 3\} = V_2$, i.e., $\{W_j\}_{j=1,2}$ is a refinement of $\{V_i\}_{i=1,2}$. With $\kappa : \{1, 2, 3\} \to \{1, 2\}$, $\kappa(1) = \kappa(2) = 1$ and $\kappa(3) = 2$, all requirements for Theorem 4.10 are fulfilled, i.e., we have $Q = Q'$. We can easily check that this is indeed the case. Using the notation of the proof we have

$$A_{W_1} = A_{W_2} = \begin{bmatrix} 2 \end{bmatrix} \quad \text{and} \quad A_{W_1}^{-1} = A_{W_2}^{-1} = \begin{bmatrix} 0.5 \end{bmatrix}$$

and hence

$$\Pi_1' = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \Pi_2' = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0.5 \\ 0 & 0 & 0 \end{bmatrix}, \quad \Pi_3' = \begin{bmatrix} 0 & 0 & 0 \\ -0.25 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}.$$

Then

$$Q' = (I - \Pi_3')(I - \Pi_2')(I - \Pi_1') = \begin{bmatrix} 0 & 0 & -0.5 \\ 0 & 0 & -0.125 \\ 0 & 0 & 0.25 \end{bmatrix} = Q.$$

Now we want to consider a case where we reorder $\{W_j\}$ in such a ways that κ is not monotonically increasing but still fulfills all other requirements for Theorem 4.10. Let $\{\tilde{W}_j\}_{j=1,2,3}$ be the reordered cover with $\tilde{W}_1 = \{2,3\} = V_2$, $\tilde{W}_2 = \{1\}$ and $\tilde{W}_3 = \{2\}$. The corresponding index map $\tilde{\kappa}$ from $\{\tilde{W}_j\}$ to $\{V_i\}$ is $\tilde{\kappa}(1) = 2$, $\tilde{\kappa}(2) = 1$ and $\tilde{\kappa}(3) = 1$. Then

$$\tilde{\Pi}_1 = \begin{bmatrix} 0 & 0 & 0 \\ -0.25 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}, \quad \tilde{\Pi}_2 = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \tilde{\Pi}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0.5 \\ 0 & 0 & 0 \end{bmatrix}$$

and hence

$$\tilde{Q} = (I - \tilde{\Pi}_3)(I - \tilde{\Pi}_2)(I - \tilde{\Pi}_1) = \begin{bmatrix} 0.25 & 0 & 0 \\ 0.25 & 0 & 0 \\ -0.5 & 0 & 0 \end{bmatrix} \neq Q.$$

Theorem 4.10 shows that for the application of the multiplicative Schwarz method, we can split the node sets of a cover into their connected components:

Corollary 4.12: Let $\mathcal{G}(A) = (V, E)$ be the digraph of a matrix $A \in \mathbb{R}^{n \times n}$. Let $\{V_i\}_{i=1,\dots,q}$ be a cover of V. Let $j \in \{1, \dots, q\}$ such that $\mathcal{G}|_{V_j}$ is not connected. Let $C_1, \dots, C_m$ be the connected components of $\mathcal{G}|_{V_j}$. Then

$$\mathrm{MS}\big(V_1, \dots, V_q\big) = \mathrm{MS}\big(V_1, \dots, V_{j-1}, C_1, \dots, C_m, V_{j+1}, \dots, V_q\big).$$

4.2 The OBGP Algorithm

The general idea of the OBGP algorithm (Overlapping Blocks by Growing a Partition) is to take an existing non-overlapping partition, e.g., a partition computed by Metis [34] or PABLO [14], [30], [36], and extend the blocks so that they have some overlap. In this way, we can reuse existing high-quality graph partitioners. In addition, in many applications we may have information about the problem which allows us to

easily find a good (non-overlapping) partition, e.g., when working on certain types of discretizations of partial differential equations.

The extension of a block is done separately for each block in the partition and is completely independent of the extensions of the other blocks. Our new graph-based algorithm works very well in many cases where a simple matrix-based overlapping strategy is not useful. An example of this will be presented together with the numerical results in section 4.4.

4.2.1 Growing a Block

Let $V_1, \ldots, V_q$ be an existing (non-overlapping) partition of V, i.e., $\bigcup_{i=1}^{q} V_i = V$ and $V_i \cap V_j = \emptyset$, $i \neq j$. The task of OBGP is to grow this partition into an overlapping cover $W_i, \ldots, W_q$ with $W_i \supsetneq V_i$. Algorithm 4.3 shows an outline of OBGP, which will be explained in this section.

1: **input:** a digraph $\mathcal{G}(A) = (V, E)$
   and a (non-overlapping) partition $\{V_1, \ldots, V_q\}$
2: **output:** an (overlapping) cover $\{W_1, \ldots, W_q\}$ of V
3: **for** $i = 1, \ldots, q$ **do** {go through each block}
4: compute $L := \text{adj}(V_i)$ {set of candidate nodes}
5: select nodes $N \subset L$ for inclusion
6: $W_i := V_i \cup N$
7: **end for**

Algorithm 4.3. Outline of the OBGP Algorithm

In OBGP the growth of one block is independent of the growth of the other blocks. Therefore, we only show how to grow one block. Let $B = V_i$ for some $i = 1, \ldots, q$, be the ordered node set of a block we want to grow. For the remainder of this section, we assume that the index i is fixed.

In order to contain the computational effort, we want to restrict the amount of nodes considered for inclusion into B. We also want to grow existing blocks and not add new ones. It is therefore desirable not to add a new connected component to the

block. Corollary 4.12 shows that adding a new connected component is equivalent to adding a new block. Therefore, in OBGP only nodes directly adjacent to the current block, i.e., nodes in adj(B), will be considered. We call these nodes *candidate nodes*. Thus, in the basic version of OBGP we have $W_i \subset V_i \cup \text{adj}(V_i)$. This restriction reduces the computational effort and is sufficient to guarantee that we do not add a new connected component.

Note that we could end up with reducing the number of connected components if B is not connected, i.e., it can happen that OBGP "connects" two or more of the connected components of $\mathcal{G}|_B$ by growing the block. If we use, e.g., XPABLO to find the non-overlapping partition, it is not uncommon for a block to have more than one connected component since XPABLO often merges small blocks, see section 3.5.2. Having this background in mind, there is, as far as we know, no reason to assume that connecting such components is "bad". Therefore, we do not take measures to prevent this from happening or even to just check for it.

To allow more overlap and more distant nodes to become candidate nodes, a block can be extended several times in OBGP. Doing this ℓ times is called doing ℓ *rounds*. The OBGP variant where each block is grown by ℓ rounds is called OBGP(ℓ). Algorithm 4.4 shows the outline of OBGP(ℓ). We denote the original block B by $B^{(0)}$ and the grown block after k rounds by $B^{(k)}$, i.e., $B^{(\ell)}$ is the grown block computed by OBGP(ℓ). In round k, $1 \leq k \leq \ell$, only nodes in adj($B^{(k-1)}$) are candidate nodes. Therefore, as before, in the first round ($k = 1$) only nodes in adj(B) are candidate nodes. However, the candidate nodes in round $k > 1$ need not to be in adj(B).

Adding all candidate nodes in round k, $1 \leq k \leq \ell$, is equivalent to adding all nodes in adj($B^{(k-1)}$) to the block. This can grow the blocks and hence the computational cost excessively. An example for such a behavior is the `RAEFSKY2` matrix. Although the iteration count reduces substantially by adding all candidate nodes as

```
1:  input: a digraph G(A) = (V, E)
        and a (non-overlapping) partition {V_1, ..., V_q}
2:  output: an (overlapping) cover {W_1, ..., W_q} of V
3:  for i = 1, ..., q do        {go through each block}
4:      B := V_i
5:      for k = 1, ..., ℓ do        {ℓ rounds}
6:          compute L := adj(B)        {set of candidate nodes}
7:          select nodes N ⊂ L for inclusion
8:          B := B ∪ N
9:      end for
10:     W_i := B
11: end for
```

Algorithm 4.4. Outline of the OBGP(ℓ) Algorithm

overlap in each round, the total solving time actually increases, i.e., the reduced iteration count does not compensate the additional time needed in each iteration and in the setup of the preconditioner. In Table 4.1 we show the total solve times and iteration counts for solving `RAEFSKY2` using the different amounts of overlap. Note that we do not add overlap if we do zero rounds. See Figure 4.1 for the corresponding convergence curves.

Table 4.1. Results for Adding All Candidate Nodes as Overlap

Rounds of Adding Overlap	Time	Iter
zero rounds (no overlap)	0.665	39
one round	0.846	16
two rounds	1.388	10
three rounds	2.230	2

In any round k, $k = 1, \ldots, \ell$, at most $\mu\big(|B^{(k-1)}|\big)$ of the candidate nodes can be selected for the extension, where $\mu : \mathbb{N} \to \mathbb{N}$ is a function chosen a priori.

There is also a total growth bound $\nu(|B|) \geq 0$, which limits the size of $B^{(k)}$ for all $k = 1, \ldots, \ell$, i.e., $|B^{(k)}| \leq |B^{(0)}| + \nu(|B^{(0)}|)$, $k = 1, \ldots, \ell$. With both μ and ν,

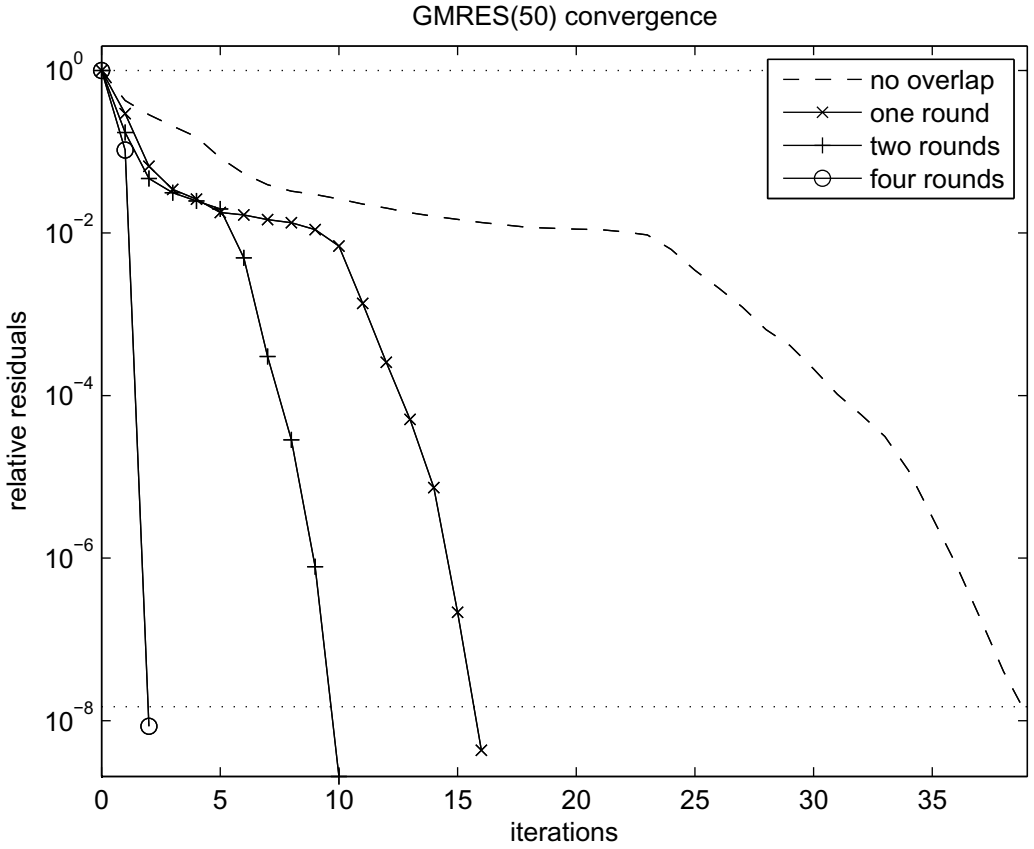


Figure 4.1. Too Much Overlap in Multiplicative Schwarz Preconditioning

The curves show the GMRES convergence for solving a linear system (matrix `RAEFSKY2`) using the block Gauss–Seidel preconditioner (dashed curve) and several multiplicative Schwarz preconditioners (solid curves). Note that the relative residuals plotted in the graph are the relative residuals of the preconditioned systems. They should not be compared with the norms in the "Rel. Res." column of any results table.

the user of OBGP can decide not to enforce any bounds, e.g., by setting $\mu, \nu \equiv \infty$. In most applications we will have $\mu(|B^{(k)}|) < \left|\text{adj}(B^{(k)})\right|$, i.e., in general we expect to have more candidate nodes than the number of nodes we are allowed to add. Therefore, OBGP needs to decide which candidate nodes are actually added to the block. In order to do this, each candidate node is given a node weight and only nodes with largest node weight are added to the block. In round k up to $\mu(|B^{(k-1)}|)$ nodes with largest weights from the candidate node set $\text{adj}(B^{(k-1)})$ are used to grow the block. We add all candidate nodes if the number of candidate nodes $\text{adj}(B^{(k-1)})$ is less than the limit $\mu(|B^{(k-1)}|)$ and less than the number of nodes we can still add according to the total growth bound ν, i.e., we add all candidate nodes if

$$\left|\text{adj}(B^{(k-1)})\right| \leq \min\Big\{ \underbrace{\mu(|B^{(k-1)}|)}_{\text{number of nodes we can add in round } k}, \overbrace{\nu(|B^{(0)}|)}^{\text{total growth bound}} - \underbrace{\big(|B^{(k-1)}| - |B^{(0)}|\big)}_{\text{number of nodes added in first } k-1 \text{ rounds}} \Big\}.$$

Definition 4.13: Let $\mathcal{G} = (V, E)$ be a directed graph with edge weights $w_E(e)$, $e \in E$. Let $B \subset V$ be a set of nodes and $j \in V$. The weight $w(j, B)$ of j with respect to B is defined as

$$w(j, B) := \sum_{e \in \text{inc}(\{j\}, B)} |w_E(e)| \qquad \diamond$$

As discussed before, the number of nodes to be added in round k is bounded by $\mu = \mu(|B^{(k-1)}|)$. Typically we will set this bound to

$$\mu\big(|B^{(k-1)}|\big) = \alpha \cdot \sqrt{\left|B^{(k-1)}\right|},$$

where α is a user-supplied positive constant. A typical value for α is $\alpha = 1$, which was found in our experiments to give good results. This definition of μ is motivated by the size of the boundary in a graph stemming from a discretization of a two-dimensional partial differential equation over a square domain. We will later see that this bound μ also has nice consequences for the time complexity of OBGP, see Corollary 4.19.

The ingredients presented so far are put together in Algorithm 4.5, which shows the basic version of OBGP(ℓ). The set L in the algorithm is the set of candidate nodes and N is the set of candidate nodes selected for extending the block. Note that the loop over all blocks and the loop over all rounds could be exchanged without any further modifications to the algorithm because the extension of a block is independent of the other blocks. The complexity analysis presented later in this chapter will show the advantages of running the ℓ rounds successively for a fixed block.

---

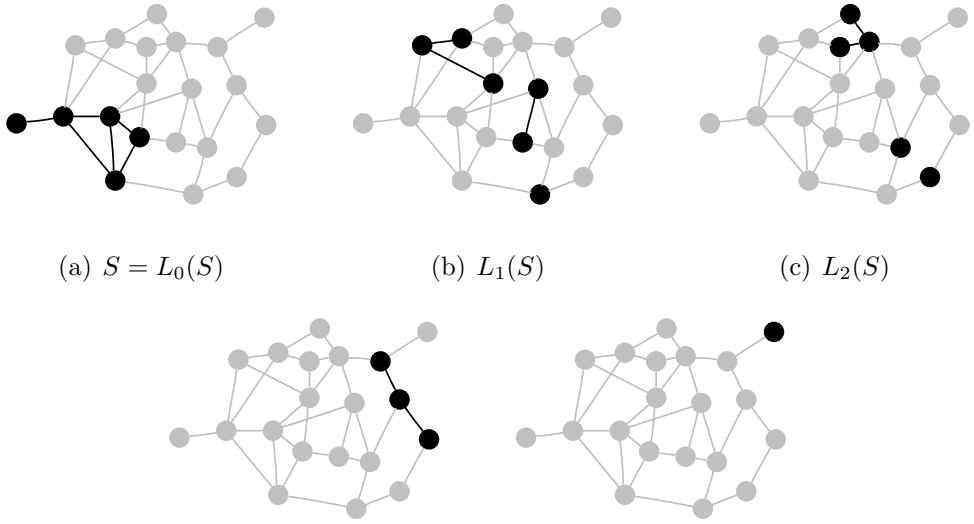
input: a digraph $\mathcal{G}(A) = (V, E)$ with edge-weights w_E
    and a (non-overlapping) partition $\{V_1, \ldots, V_q\}$ of V
output: an (overlapping) cover $\{W_1, \ldots, W_q\}$ of V
for $i = 1, \ldots, q$ **do** {go through each block}
    $V_i^{(0)} = V_i$
    $\nu_0 := \nu(|V_i^{(0)}|)$
    for $k = 1, \ldots, \ell$ **do** {ℓ rounds}
        $L := \mathrm{adj}(V_i^{(k-1)})$
        $\mu := \mu(V_i^{(k-1)})$
        compute weights $w(v, V_i^{(k-1)})$ for all $v \in L$
        select nodes $N \subset L$ with largest weights $\{|N| = \min\{|L|, \mu, \nu_{k-1}\}\}$
        $V_i^{(k)} := V_i^{(k-1)} \cup N$
        $\nu_k := \nu_{k-1} - |N|$. $\{\nu_k \geq 0$ since $|N| \leq \nu_{k-1}\}$
    end for
    $W_i := V_i^{(\ell)}$
end for

Algorithm 4.5. Basic Version of the OBGP(ℓ) Algorithm

For the following observation we need to generalize the concept of the adjacency set adj($\cdot$) to level sets:

Definition 4.14: Let $\mathcal{G} = (V, E)$ be a graph and $S \subset V$. The kth *level set* $L_k(S)$ with respect to S is defined as

$$L_k(S) := \begin{cases} S & \text{if } k = 0, \\ \mathrm{adj}(S) & \text{if } k = 1, \\ \mathrm{adj}\big(L_{k-1}(S)\big) \setminus L_{k-2}(S) & \text{if } k > 1. \end{cases} \qquad \diamond$$



(a) $S = L_0(S)$ (b) $L_1(S)$ (c) $L_2(S)$

(d) $L_3(S)$ (e) $L_4(S)$

Figure 4.2. Level Sets $L_0(S)$ through $L_4(S)$ with Respect to the Node Set S

Figure 4.2 shows for some graph $G = (V, E)$ and some node set S the level sets $L_0(S)$ through $L_4(S)$.

Let $B = B^{(0)} = V_i$, $i \in \{1, \ldots, q\}$, be any block and let $N^{(k)}$ and $L^{(k)}$ be the sets N and L computed in the kth round of adding nodes to B in Algorithm 4.5, i.e., $N^{(k)} \subset L^{(k)} = L_1(B^{(k-1)})$ and $B^{(k)} = B^{(k-1)} \cup N^{(k)}$. The sets $N^{(k)}$ and $L^{(k)}$ can be described in terms of level sets by

$$N^{(1)} \subset L^{(1)} = L_1(B),$$

$$N^{(2)} \subset L^{(2)} = L_1(B \cup N^{(1)}),$$

$$\vdots$$

$$N^{(k)} \subset L^{(k)} = L_1\Big(B \cup \bigcup_{j=1}^{k-1} N^{(j)}\Big).$$

In terms of levels set with respect to B, the relations

$$N^{(1)} \subset L^{(1)} = L_1(B),$$

$$N^{(2)} \subset L^{(2)} \subset L_1(B) \cup L_2(B),$$

$$\vdots$$

$$\text{and} \quad N^{(k)} \subset L^{(k)} \subset \bigcup_{j=1}^{k} L_j(B)$$

hold.

This shows the following result.

Proposition 4.15: A node added to a block in the kth round comes from one of the first k level sets with respect to the original block from the non-overlapping partition.

Figure 4.3 illustrates the growth process of OBGP.

4.2.2 Implementation

In this section we discuss in detail all ingredients needed to implement Algorithm 4.5 in an efficient way. Let $V_i^{(0)} = V_i$ be the ith non-overlapping block of a (non-overlapping) partition $\{V_1, \ldots, V_q\}$ of V. Let $V_i^{(k)}$ be the block after the kth round and W_i be the block after all ℓ rounds, i.e., $W_i = V_i^{(\ell)}$. Similar to the notation in the previous section, let $N_i^{(k)}$ and $L_i^{(k)}$ be the sets N and L computed in the kth round for block i. Since each block is considered separately, the index i will be omitted in many cases where the statements hold for all $i = 1, \ldots, q$.

In the case of multiple rounds, i.e., if $\ell > 1$, we can update the set L of candidate nodes going from one round to the next instead of computing it from scratch in each round. We will see in section 4.3 that it is in fact much more efficient to update L. In the kth round the new set of candidate nodes $L^{(k)}$ consists of the previous candidate nodes from $L^{(k-1)}$ not added to the block in round $k-1$, i.e., not

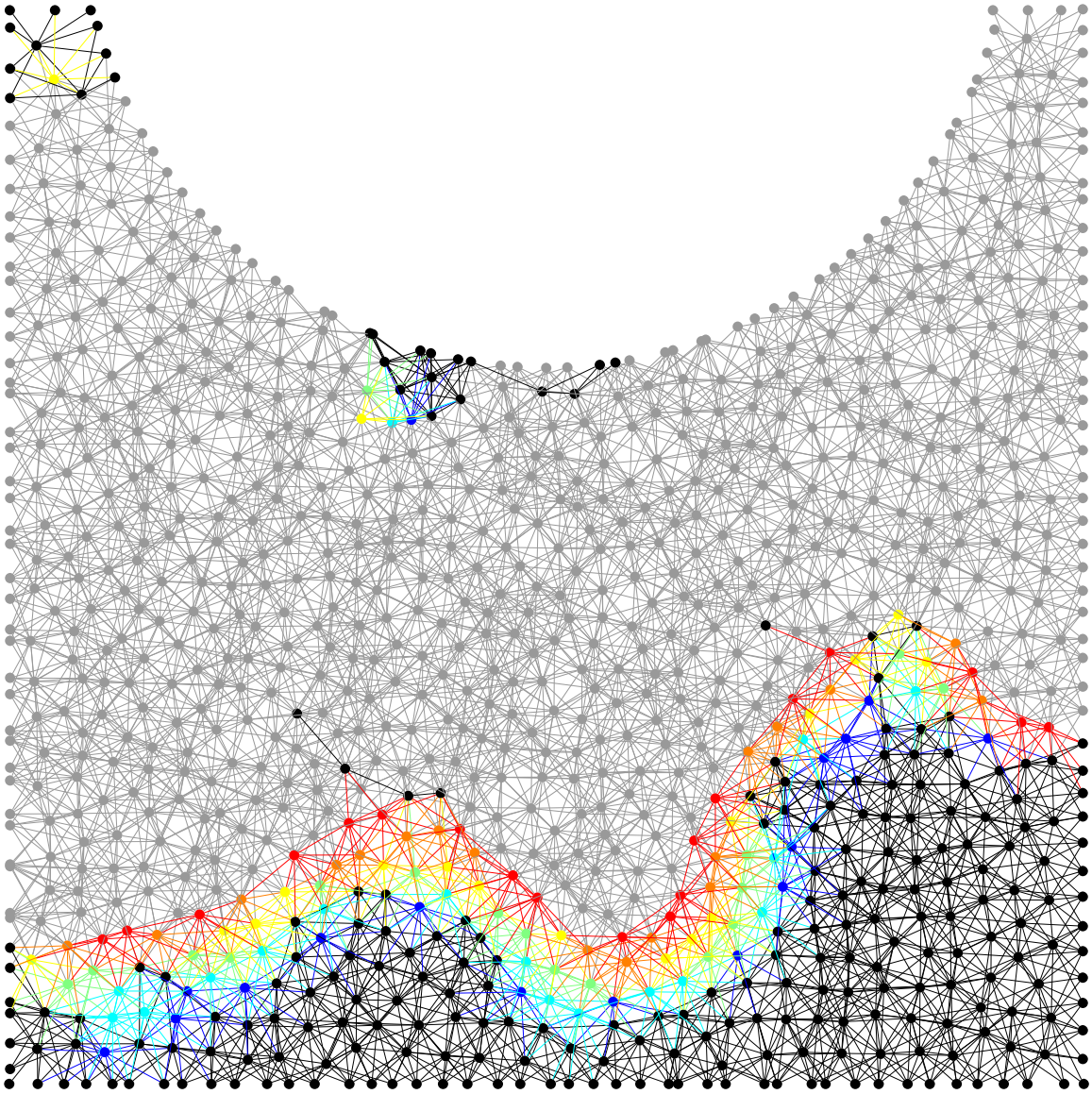


Figure 4.3. OBGP(6) Block Growth of `LSQ_2D_1000`

The figure shows how the fourth block (the green block) in Figure 3.2 is grown by OBGP(6). The nodes in the original block are in black. The nodes added in round one to six are shown in blue, cyan, green, yellow, orange and red, respectively. Edges are shown in the color of the round in which they became edges inside the (grown) block. Nodes and edges not inside the block after the sixth round are shown in grey.

in $N^{(k-1)}$, and nodes adjacent to $N^{(k-1)}$, which are not in the previous block $V^{(k-1)}$, i.e.,

$$L^{(k)} = \big(L^{(k-1)} \setminus N^{(k-1)}\big) \cup \big(L_1(N^{(k-1)}) \setminus V^{(k-1)}\big).$$

As a more detailed version of Algorithm 4.5, Algorithm 4.6 describes OBGP(ℓ) using updates of the candidate set. This algorithm also shows how the node weights $w(v) = w(v, V_i^{(k-1)})$, $v \in L$, are computed and updated. Figure 4.4 shows how OBGP grows a given block.

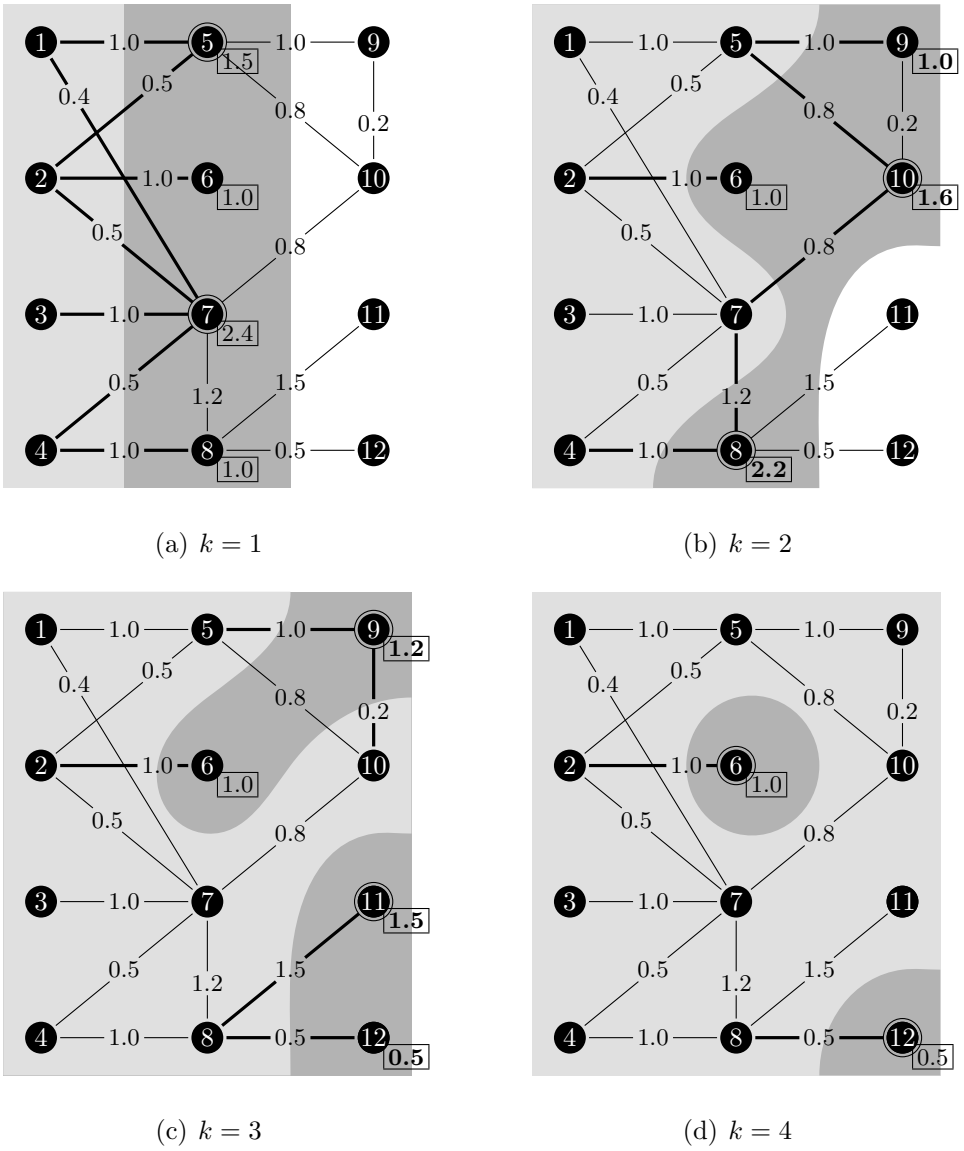
For a time and space efficient implementation of OBGP(ℓ), the sets $V_i^{(k)}$, $k = 0, \ldots, \ell$, and the set N are not stored separately. For a fixed block number i, the invariants

$$V^{(k-1)} \subset V^{(k)}, \qquad V^{(k-1)} \cap N^{(k)} = \emptyset \qquad \text{and} \qquad V^{(k-1)} \cup N^{(k)} = V^{(k)}$$

hold for $k = 1, \ldots, \ell$. We can therefore store the node numbers of the nodes in these sets as shown in Figure 4.5 in an integer vector of size n. Furthermore, this integer vector can be reused for the different blocks as each block is grown independently of the other blocks.

An important choice is the data structure for storing the set L. Several operations are done with L:

- Adding a node to L.
- Selecting and removing some nodes with largest weights.
- Iterating over the nodes in L. Note that it is not necessary to traverse the nodes in a specific order.
- Changing the weight associated with a node in L.



(a) $k = 1$ (b) $k = 2$

(c) $k = 3$ (d) $k = 4$

Figure 4.4. OBGP Block Growth Showing Edge Weights

In each picture the current block $B^{(k-1)}$ is shown with a light grey background and the set L of candidate nodes with a slightly darker grey background. The index k is the counter of the current round, cf. Algorithms 4.5 and 4.6. The edges used to compute the node weights are printed with thick lines. The computed node weights are printed in a box right next to the nodes. Node weights printed in bold have been changed from the previous round. The nodes selected for inclusion into the block are marked by an extra circle around the node.

input: a digraph $\mathcal{G}(A) = (V, E)$ with edge-weights w_E
and a (non-overlapping) partition $\{V_1, \ldots, V_q\}$ of V
output: an (overlapping) cover $\{W_1, \ldots, W_q\}$ of V
set $w(j) := 0$, $j = 1, \ldots, n$
for $i = 1, \ldots, q$ **do** {go through each block}
  $V_i^{(0)} := V_i$
  $\nu_0 := \nu(|V_i^{(0)}|)$
  $N^{(0)} := V_i$ {newly added nodes}
  $L := \emptyset$ {level set}
  for $k = 1, \ldots, \ell$ **do**
    for all $v \in N^{(k-1)}$ **do**
      for all $e \in E$ incident to v **do**
        if e connects v to some node $v' \notin V_i^{(k-1)}$ **then**
          $w(v') := w(v') + |w_E(e)|$
          if $v' \notin L$ **then**
            $L := L \cup \{v'\}$
          end if
        end if
      end for
    end for
    select nodes $N^{(k)} \subset L$ with largest weights
      $\{|N^{(k)}| = \min\{|L|, \mu(V_i^{(k-1)}), \nu_{k-1}\}\}$
    $L := L \setminus N^{(k)}$
    $V_i^{(k)} := V_i^{(k-1)} \cup N^{(k)}$
    $w(j) := 0$, $j \in N^{(k)}$
    $\nu_k := \nu_{k-1} - |N^{(k)}|$. $\{\nu_k \geq 0$ since $|N^{(k)}| \leq \nu_{k-1}\}$
  end for
  $w(j) := 0$, $j \in L$ {now we have $w \equiv 0$}
  $W_i := V_i^{(\ell)}$
end for

Algorithm 4.6. OBGP(ℓ)

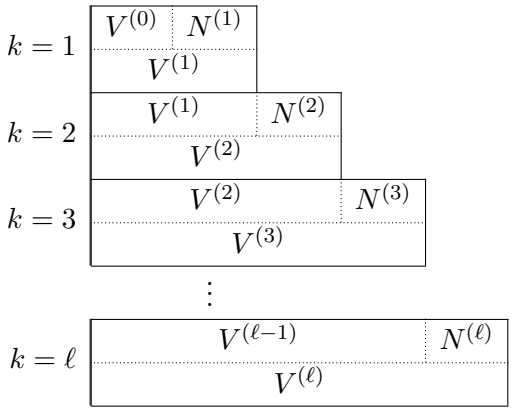


Figure 4.5. Storage of Node Sets $V^{(k)}$ and $N^{(k)}$
The figure shows how to store $V^{(k)}$ and $N^{(k)}$ in OBGP(ℓ) for a fixed block i. Only one integer vector of size n is needed.

There is no single "best" data structure allowing all listed operations with optimal time complexity. If we use, e.g., a linked list with an external index to store L, adding nodes, iterating over the nodes and changing node weight all have time complexity $\mathcal{O}(1)$, but selecting the node with largest weight is $\mathcal{O}(|L|)$. Overall, we mainly want all operations to have a reasonable time complexity. The best compromise we know of is to use a heap data structure. In section 4.3 and in our numerical results in section 4.4, we use a binary heap. We refer to [16] for a detailed description and analysis of binary heaps. If a binary heap is used, adding a node to L has time complexity $\mathcal{O}(\log|L|)$ and removing the largest node has time complexity $\mathcal{O}(\log|L|)$. In the following, "heap" will always mean "binary heap", although the results hold also for other kinds of heaps like binomial heaps or Fibonacci heaps. We again refer to [16] for detailed information about binomial heaps and Fibonacci heaps.

4.3 Complexity Analysis of OBGP

In this section we assume that a heap data structure is used for storing L.

Theorem 4.16: Let L be stored in a heap. Then algorithm OBGP(ℓ) can be implemented in such a way that the time complexity is $\mathcal{O}(q \cdot (\text{nnz}(A) + n \log n))$.

Proof: We will show that the time complexity for the computation of one particular set W_i is $\mathcal{O}(\text{nnz}(A) + n \log n)$; see lines 5–27 in Algorithm 4.6 for the necessary steps.

Updating w is done for edges incident to nodes in $N^{(k)}$, $k = 0, \ldots, \ell - 1$. Note that $N^{(k)} \cap N^{(k')} = \emptyset$ for $k \neq k'$, i.e., for each node in V, the incident edges are considered at most once and hence any edge in E is considered at most twice. Therefore, the time complexity for updating w (line 13) is $\mathcal{O}(\text{nnz}(A))$.

A node selected and removed from L (lines 20 and 21) is added to the block and can therefore not be added to L again, see the condition in line 12. Therefore, each node of V is added to L at most once and thus also selected and removed from L at most once. An upper bound for $|L|$ is n. This gives a time complexity of $\mathcal{O}(n \log n)$ for adding nodes to L and for selecting and removing nodes from L.

It is easy to see that the work outside the k-loop has time complexity $\mathcal{O}(n)$. Adding this together, the time complexity for the computation of one W_i block is $\mathcal{O}(\text{nnz}(A) + n \log n)$. For q blocks the total time complexity results as $\mathcal{O}(q \cdot (\text{nnz}(A) + n \log n))$. □

For many practical problem cases a tighter time complexity can be shown than that of Theorem 4.16. For a more detailed analysis we have to use properties of the bound function ν.

The following Lemma shows some basic results needed for Theorem 4.20.

Lemma 4.17: Let $\mathcal{G} = (V, E)$ be a directed graph and let d be the maximum degree, i.e., $d = \max_{v \in V} \deg(v)$. If the total growth of block B is bounded by $|B| + \nu(|B|) \leq c_\nu |B|$, where c_ν is a constant independent of $|B|$, then

1. $|W_i| = \mathcal{O}(|V_i|)$ and $\sum_{i=1}^{q} |W_i| = \mathcal{O}(n)$.

2. $|\operatorname{inc}(W_i)| \leq d|W_i| \leq dc_\nu|V_i|$ and hence $\sum_{i=1}^{q} |\operatorname{inc}(W_i)| = \mathcal{O}(dn)$.

Note that with a maximum degree of d the number of edges is $\mathcal{O}(dn)$.

Proof: 1. follows immediately from $\nu(|B|) \leq c_\nu|B|$ and $\sum_{i=1}^{q} \mathcal{O}(|V_i|) = \mathcal{O}(n)$.

2. The set $\operatorname{inc}(W_i)$ contains the edges considered by OBGP while computing W_i. Since d is the maximum number of edges incident to a node, the number of edges in $\operatorname{inc}(W_i)$ is bounded by $|\operatorname{inc}(W_i)| \leq d|W_i|$. Thus,

$$\sum_{i=1}^{q} |\operatorname{inc}(W_i)| \leq d \sum_{i=1}^{q} |W_i| \leq dc_\nu \sum_{i=1}^{q} |V_i| = dc_\nu n = \mathcal{O}(dn). \qquad \square$$

We will show that with our suggested μ and a mild restriction on ℓ, we automatically fulfill the assumptions of Lemma 4.17. For this it is necessary to know by how much a single block can grow in ℓ rounds:

Theorem 4.18: Let $V_1, \ldots, V_q$ be a non-overlapping partition and let $W_1, \ldots, W_q$ be the cover computed by OBGP(ℓ). Let the number of nodes to be added to $V_i^{(k)}$, $i = 1, \ldots, q$, $k = 0, \ldots, \ell$, be bounded by

$$\mu\big(|V_i^{(k)}|\big) = \alpha \cdot \sqrt{\big|V_i^{(k)}\big|}. \tag{4.20}$$

Then the size of W_i, $i = 1, \ldots, q$, is bounded by

$$|W_i| \leq |V_i| + \ell \cdot \mu\big(|V_i|\big) + \frac{\ell\,(\ell-1)\,\alpha^2}{4}. \tag{4.21}$$

Proof: Use induction over ℓ. The inequality (4.21) clearly holds for $\ell = 0$ and $\ell = 1$. Then for $\ell + 1$

$$\begin{aligned}
|V_i^{(\ell+1)}| &\leq |V_i^{(\ell)}| + \mu\big(|V_i^{(\ell)}|\big) \\
&\leq |V_i| + \ell\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4} + \alpha\sqrt{\big|V_i^{(\ell)}\big|} \\
&\leq |V_i| + \ell\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4} + \alpha\sqrt{|V_i| + \ell\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4}}.
\end{aligned}$$

The identity (4.20) implies $\mu\big(|V_i|\big)^2 = \alpha^2|V_i|$ and hence

$$\begin{aligned}
|V_i^{(\ell+1)}| &\leq |V_i| + \ell\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4} + \sqrt{\mu\big(|V_i|\big)^2 + \ell\alpha^2\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^4}{4}} \\
&\leq |V_i| + \ell\mu\big(|V_i|\big) + \frac{\ell(\ell-1)\alpha^2}{4} + \mu\big(|V_i|\big) + \frac{\ell\alpha^2}{2} \\
&\leq |V_i| + (\ell+1)\mu\big(|V_i|\big) + \frac{(\ell+1)\ell\alpha^2}{4}.
\end{aligned}$$

□

Corollary 4.19: Let V_i be a node set of a partition $\{V_i\}_{i=1,\dots,q}$ of V. If $\mu(|V_i|) = \alpha\sqrt{|V_i|}$ and $\ell \leq \sqrt{|V_i|}$, then the total growth of block B is bounded by $c_\nu|B|$, i.e., with this choice of μ and ℓ the assumptions for Lemma 4.17 are satisfied.

Proof:

$$\begin{aligned}
|W_i| &\leq |V_i| + \ell \cdot \mu\big(|V_i|\big) + \frac{\ell\,(\ell-1)\,\alpha^2}{4} \\
&\leq |V_i| + \alpha|V_i| + \frac{\ell^2\alpha^2}{4} \\
&\leq |V_i| + \alpha|V_i| + \frac{\alpha^2}{4}|V_i| \\
&= \underbrace{(1 + \alpha + \alpha^2/4)}_{=:\,c_\nu}|V_i|.
\end{aligned}$$

□

Theorem 4.20: Let L be stored in a heap. Let $d = \max_{v\in V}\deg(v)$ be the maximum degree and let $s = \max_{i=1}^{q}|V_i|$ be the maximum block size of the non-overlapping partition. Let the total block growth be bounded by $|W_i| \leq c_\nu|V_i|$ for all $i = 1,\dots,q$. Then the algorithm OBGP(ℓ) can be implemented in such a way that the time complexity is $\mathcal{O}(dn + n\log s)$.

Proof: From Lemma 4.17 follows that we consider $\mathcal{O}(dn)$ edges while growing all blocks. The time for updating the node weights w is linear in the number of edges considered, i.e., the total time for updating w is $\mathcal{O}(dn)$.

In total, at most $c_\nu n$ nodes are added and removed from L, since $\sum_{i=1}^{q}|W_i| \leq c_\nu n$. The maximum size of L is at most $\max_{i=1}^{q}|\operatorname{inc}(W_i)| \leq \max_{i=1}^{q} d|W_i| \leq dc_\nu s$.

Thus, the total time for adding nodes to L and removing the largest nodes from L is bound by

$$c_\nu n\mathcal{O}\big(\log(dc_\nu s)\big) = c_\nu n\mathcal{O}(\log d + \log c_\nu + \log s) = \mathcal{O}\big(n(\log d + \log s)\big).$$

So far, we have ignored the phenomenon that nodes in the heap may change their weight and hence may have to be moved to a different position inside the heap. In Algorithm 4.6 we can see that the weight of a node can only grow, i.e., nodes only move upwards inside the heap because of a changed weight. Over the time a particular node stays in the heap, it could move from the bottom of the heap up to the top. This is also the worst case scenario as the node can not move downwards again. Nodes newly added to a heap are first put at the bottom and then moved up until they reach their correct position. In the worst case, a newly added nodes has to be moved up through the whole heap because the node belongs at the top. Therefore, the time needed for moving a node up to the top due to weight changes is already included in our worst case bound for the time for adding the node to the heap.

The total time complexity is then

$$\mathcal{O}(dn) + \mathcal{O}\big(n(\log d + \log s)\big) = \mathcal{O}(dn + n\log s) \qquad \square$$

Note that Corollary 4.19 shows that with our default choice of μ we attain the total block growth bound for Theorem 4.20 as long as $\ell \leq \sqrt{|V_i|}$ for all $i = 1, \ldots, q$, i.e., with the default μ and ℓ not too large we do not have to bound the total block growth explicitly by setting ν.

4.3.1 Dealing With Nodes With High Degree

Theorem 4.20 shows that the execution time of OBGP can be sensitive to the maximum degree d. If the original system has, e.g., dense or nearly dense rows or columns,

then d is $\mathcal{O}(n)$ and the worst case time complexity becomes $\mathcal{O}(n^2)$. If this ever becomes a problem in practice, the situation can be resolved by "removing" the nodes with high degree from the set of nodes considered for overlap, i.e., we let OBGP(ℓ) work on $\mathcal{G}|_{V \setminus V_H}$, where V_H is the set of nodes with high degree. As the nodes in V_H are especially well connected to nodes outside their block, it may be helpful to add them as another block, i.e., to use them as some kind of coarse-grid correction.

Since we did not observe severe problems with nodes of high degree in our test cases, we let OBGP(ℓ) always work on the whole graph.

4.4 Numerical Results

Table 4.2 shows a comparison of XPABLO-based block Gauss–Seidel preconditioning and multiplicative Schwarz preconditioning based on XPABLO+OBGP(ℓ). The notation is the same as in previous comparison tables, see section 3.7 for a detailed description. A star ($\star$) in the "Iter" column denotes that GMRES stagnated, i.e., two consecutive iterates were the same.

Detailed results including relative residual and relative error norms are shown in the following tables: In Table 4.4 for OBGP(5), Table 4.5 for OBGP(10), and Table 4.6 for OBGP(20). Figure 4.6 shows GMRES(50) convergence curves for the `MPS_2D_50000` problem. Note that the relative residual norms plotted in the graph are the relative residual norms of the preconditioned systems. They should not be compared with the norms in the "Rel. Res." column of any results table.

4.5 Discussion

The numerical results show impressively that adding overlap can improve the performance of block-based iterative solvers, especially for the computational fluid dynamics (CFD) problems and the semiconductor device simulation problems. For these problems the iteration count improves tremendously and, what is important for practical

Table 4.2. Comparison of XPABLO and XPABLO+OBGP(ℓ) Results

Matrix	XPABLO		$\ell = 5$		$\ell = 10$		$\ell = 20$	
	Time	Iter	Time	Iter	Time	Iter	Time	Iter
CAVITY16	0.601	49	0.481	28	0.441	22	0.411	**13**
CAVITY26	0.61	45	0.578	33	**0.519**	24	0.533	**19**
EX19	6.66	252	0.969	101	0.827	**51**	**0.801**	**51**
EX35	2.48	28	2.85	28	2.22	20	**1.64**	**10**
GARON1	0.333	37	0.293	23	0.241	14	**0.261**	**10**
GARON2	5.91	163	6.01	135	3.96	79	**2.68**	**40**
RAEFSKY2	0.665	39	**0.651**	26	0.702	21	0.859	**13**
RAEFSKY3	31.8	390	20.2	198	15.6	138	**9.15**	**62**
SHYY41	**0.112**	**6**	–	⋆	–	⋆	–	⋆
SHYY161	**9.14**	**19**	395.0	(400)	399.9	(350)	415.3	(350)
IGBT3	3.29	120	1.31	35	0.901	21	**0.842**	**14**
NMOS3	4.99	76	2.3	29	2.27	26	**1.89**	**16**
BARRIER2-1	**854.5**	**680**	1213.6	(1000)	1242.8	(1000)	1309.7	(1000)
PARA-4	519.0	210	139.7	55	102.2	39	**87.3**	**31**
PARA-8	196.8	72	90.8	33	79.9	28	**79.7**	**27**
OHNE2	1339.2	517	598.3	240	335.4	128	**161.3**	**53**
2D_54019_HIGHK	49.4	177	22.8	81	20.0	68	**15.6**	**47**
3D_51448_3D	16.2	44	11.7	31	**10.4**	26	10.5	**23**
IBM_MATRIX_2	15.2	42	10.5	28	9.63	24	**9.27**	**20**
MATRIX_9	139.5	142	72.2	76	53.5	52	**46.6**	**42**
MATRIX-NEW_3	80.9	93	69.7	86	52.6	62	**38.4**	**42**
LSQ_2D_1000	0.024	1	0.024	1	0.024	1	0.070	1
LSQ_2D_2000	0.084	15	**0.075**	6	0.079	4	0.092	**3**
LSQ_2D_5000	0.369	24	0.297	12	**0.294**	9	0.325	**6**
LSQ_2D_10000	1.04	34	0.819	19	**0.758**	14	0.802	**10**
LSQ_2D_50000	18.4	75	11.9	41	9.89	31	**9.1**	**23**
LSQ_2D_100000	88.4	111	48.4	57	39.1	43	**33.2**	**32**
LSQ_2D_200000	525.0	188	259.0	96	**177.9**	61	178.1	**46**
LSQ_2D_400000	2467.8	240	1476.5	125	6025.3	89	**oom**	
MPS_2D_10000	1.56	73	1.08	42	0.831	30	**0.616**	**17**
MPS_2D_50000	36.6	185	24.2	116	17.9	82	**10.7**	**42**
MPS_2D_100000	172.8	257	115.1	177	71	103	**45.5**	**61**
MPS_2D_200000	1019.8	418	509.1	225	506.5	180	**222.9**	**92**
MPS_2D_400000	6889.9	759	3087.2	387	2023.0	250	**1354.0**	**146**
LSQ_3D_10000	**0.779**	18	0.814	13	0.872	9	1.22	**7**
LSQ_3D_50000	9.49	26	**9.41**	21	9.85	18	11.4	**13**
LSQ_3D_100000	33.2	31	**30.2**	25	30.7	22	33.4	**17**
LSQ_3D_200000	127.9	38	144.8	31	**110.6**	28	1246.7	**22**
MPS_3D_10000	0.599	27	0.63	24	**0.582**	19	0.604	**13**
MPS_3D_50000	9.18	44	8.28	38	7.95	34	**7.27**	**26**
MPS_3D_100000	39.4	55	34.2	49	31.7	44	**27.4**	**34**
MPS_3D_200000	176.1	70	142.6	62	127.7	53	**106.3**	**42**

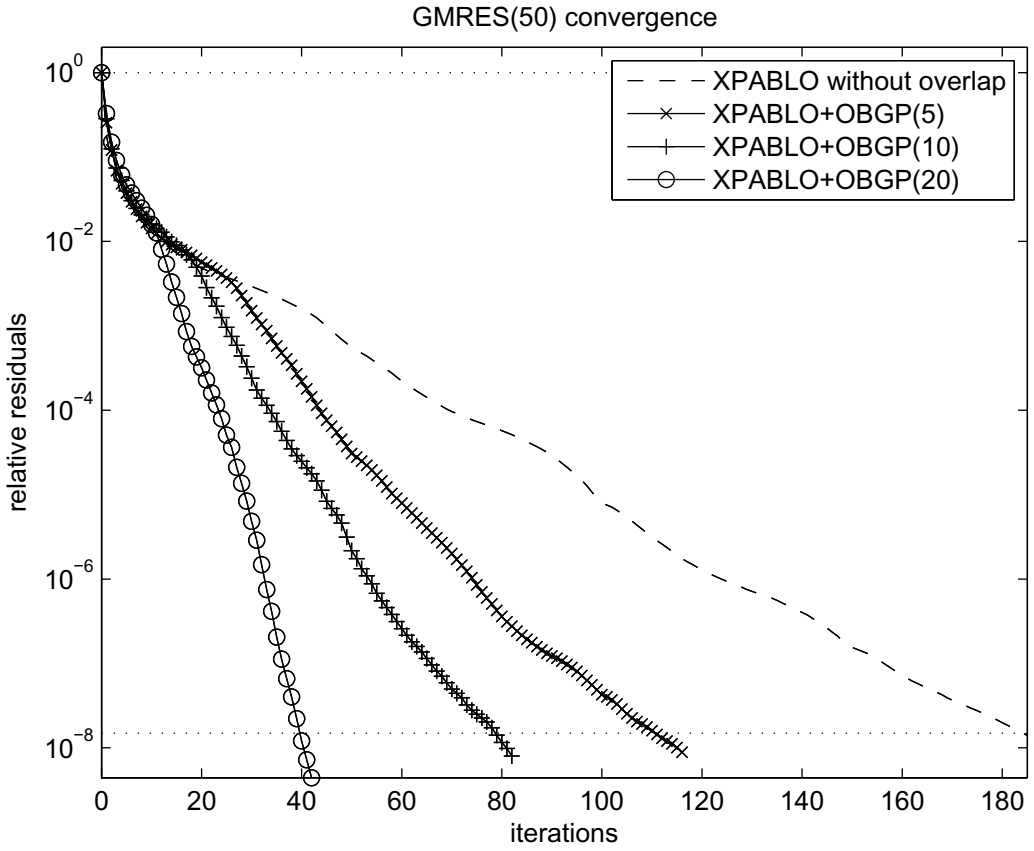


Figure 4.6. Convergence Curves for `MPS_2D_50000`

Note that the relative residuals plotted in the graph are the relative residuals of the preconditioned systems. They should not be compared with the norms in the "Rel. Res." column of any results table.

applications, also the total time needed to solve the linear system improves by adding overlap. It is not uncommon for our test problems to see a speedup by a factor of two or more. For the `OHNE2` matrix the speedup factor is more than eight (161.3 seconds for XPABLO+OBGP(20) compared to 1339.2 seconds for XPABLO). Recall that this is an example, where both a direct method and ILU-based preconditioning failed to solve the linear system at all.

Since we assume the multiplicative Schwarz preconditioner to be a better approximation of A if we increase the overlap or add overlap in the first place, it is not surprising that we can in general see a decrease in the iteration count if we increase the amount of overlap. This assumption is for general problems really only a heuristic and we can observe some counterexamples from our test problems: With `CAVITY26`, `SHYY41`, `SHYY161`, and `BARRIER2-1` the iteration count does not always decrease when the overlap is increased.

For M-matrices the assumption of getting a lower iteration count by adding (more) overlap is based on the result that for M-matrices the multiplicative Schwarz method converges faster if the amount of overlap is increased, see [6]. Note that the `MPS_2D_`$*$ and the `MPS_3D_`$*$ matrices are M-matrices, see [45].

The lower iteration count achieved by adding overlap is paid by a higher computation cost to setup the preconditioner and to apply it (or compute a preconditioned matrix-vector multiplication) in each iteration step. The increased setup cost and increased cost per iteration can be observed for several test problems, e.g., for `RAEFSKY2` and for `LSQ_2D_200000`. For `RAEFSKY2` the XPABLO-based preconditioner without overlap is the fastest, although the iteration count is reduced from 39 to 13 if we compare XPABLO with XPABLO+OBGP(20). For `LSQ_2D_200000` the iteration count goes down from 61 to 46 in the step from $\ell = 10$ to $\ell = 20$, but the total solving time does not improve.

Moreover, more memory is needed if we add more overlap. Problems from the increased memory consumption can be seen for the `LSQ_2D_400000` and the `LSQ_3D_200000` problems. For the first problem our solver ran out of memory for $\ell = 20$. For the `LSQ_3D_200000` problem the solver could finish for $\ell = 20$ only with severe swapping such that the total time for $\ell = 20$ is more than ten times the total time for $\ell = 10$.

In general we can conclude that the amount of overlap needed to minimize the total solving time is problem specific.

The failure to "solve" the `SHYY41` and `SHYY161` problems does not contradict the general robustness of the XPABLO+OBGP(ℓ) approach, since these two problems are severely ill-conditioned: The 1-norm condition estimates are $3.51 \cdot 10^{48}$ for `SHYY41` and $8.23 \cdot 10^{277}$ for `SHYY161` and the relative error norms for the "solution" computed by UMFPACK are $3.54 \cdot 10^{26}$ and $1.71 \cdot 10^{255}$, respectively.

For the meshfree discretizations we also see a general improvement of the iteration count by adding overlap. For most problems the total solving times can be improved by adding overlap, but not always the best total time is achieved for the maximal tested overlap ($\ell = 20$). We can observe that $\ell = 20$ gave the best results for the MPS discretizations, both in 2-D and in 3-D. For the LSQ discretizations a smaller amount of overlap may be the best. More experiments would be needed to come to a clear conclusion. Compared to the direct solver and to ILU-based preconditioners, we can see the following patterns: For the two-dimensional problems there is no big change in the overall picture, i.e., for (`LSQ_2D_`$*$ and `MPS_2D_`$*$) the direct solver is still the best, followed closely by ILUTP(10^{-3}). For the LSQ discretizations of the three-dimensional problem XPABLO-based preconditioners were already superior to the direct solver and the best ILU-based preconditioners. Additionally, we can seen further improvements of the XPABLO results by adding overlap. For

the MPS discretizations of the two-dimensional problem ILUTP(10^{-2}) is the fastest ILU-based preconditioner and XPABLO+OBGP(20) is the fastest XPABLO-based preconditioner, with both being faster than the direct solver. In a direct comparison ILUTP(10^{-2}) is still two to three times faster than XPABLO+OBGP(20) for the 2-D MPS discretizations. However, so far we have not optimized the XPABLO parameters or the overlap-parameter ℓ to these specific problem. The results in Table 4.3 show the effect of a simple increase of the XPABLO block sizes to $minbs = 800$ and $maxbs = 4000$. For the largest three systems the total time is much lower than in the experiments using the default blocks sizes $minbs = 200$ and $maxbs = 1000$. Moreover, the measured total time of 39.5 seconds is 28 % lower than the best ILU-based total time of 55.0 seconds (using ILUTP(10^{-2})).

Table 4.3. XPABLO+OBGP(20) Results Using $minbs = 800$ and $maxbs = 4000$

Matrix	Time	Iter	Rel. Res.	Rel. Err.
MPS_3D_10000	1.16	10	$2.192 \cdot 10^{-5}$	$6.734 \cdot 10^{-9}$
MPS_3D_50000	4.15	31	$4.347 \cdot 10^{-4}$	$2.927 \cdot 10^{-8}$
MPS_3D_100000	12.3	40	$2.403 \cdot 10^{-3}$	$8.763 \cdot 10^{-8}$
MPS_3D_200000	39.5	51	$4.570 \cdot 10^{-3}$	$8.904 \cdot 10^{-8}$

Table 4.4. XPABLO+OBGP(5) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	0.481	28	$8.763 \cdot 10^{+0}$	$7.491 \cdot 10^{+1}$
CAVITY26	0.578	33	$4.452 \cdot 10^{+0}$	$8.458 \cdot 10^{+1}$
EX19	0.969	101	$3.904 \cdot 10^{+0}$	$1.307 \cdot 10^{+0}$
EX35	2.85	28	$2.432 \cdot 10^{+4}$	$2.742 \cdot 10^{+4}$
GARON1	0.293	23	$1.273 \cdot 10^{+0}$	$1.938 \cdot 10^{+2}$
GARON2	6.01	135	$1.162 \cdot 10^{+0}$	$4.029 \cdot 10^{+2}$
RAEFSKY2	0.651	26	$3.468 \cdot 10^{+0}$	na
RAEFSKY3	20.2	198	$3.925 \cdot 10^{+2}$	na
SHYY41	–	⋆	–	–
SHYY161	395.0	(400)	$6.614 \cdot 10^{-1}$	$6.486 \cdot 10^{-1}$
IGBT3	1.31	35	$7.036 \cdot 10^{+4}$	na
NMOS3	2.3	29	$5.603 \cdot 10^{+3}$	na
BARRIER2-1	1213.6	(1000)	$1.814 \cdot 10^{+4}$	na
PARA-4	139.7	55	$2.031 \cdot 10^{+4}$	na
PARA-8	90.8	33	$1.013 \cdot 10^{+0}$	na
OHNE2	598.3	240	$5.614 \cdot 10^{+5}$	na
2D_54019_HIGHK	22.8	81	$2.412 \cdot 10^{+1}$	na
3D_51448_3D	11.7	31	$9.733 \cdot 10^{-1}$	na
IBM_MATRIX_2	10.5	28	$9.746 \cdot 10^{-1}$	na
MATRIX_9	72.2	76	$7.785 \cdot 10^{+4}$	na
MATRIX-NEW_3	69.7	86	$2.640 \cdot 10^{+1}$	na
LSQ_2D_1000	0.024	1	$2.782 \cdot 10^{-10}$	$6.631 \cdot 10^{-14}$
LSQ_2D_2000	0.075	6	$1.863 \cdot 10^{-5}$	$1.572 \cdot 10^{-9}$
LSQ_2D_5000	0.297	12	$5.000 \cdot 10^{-5}$	$1.316 \cdot 10^{-9}$
LSQ_2D_10000	0.819	19	$1.426 \cdot 10^{-4}$	$1.693 \cdot 10^{-9}$
LSQ_2D_50000	11.9	41	$6.628 \cdot 10^{-3}$	$9.723 \cdot 10^{-9}$
LSQ_2D_100000	48.4	57	$4.161 \cdot 10^{-2}$	$3.286 \cdot 10^{-8}$
LSQ_2D_200000	259.0	96	$2.183 \cdot 10^{-1}$	$1.095 \cdot 10^{-7}$
LSQ_2D_400000	1476.5	125	$1.905 \cdot 10^{+0}$	$3.865 \cdot 10^{-7}$
MPS_2D_10000	1.08	42	$2.115 \cdot 10^{-3}$	$9.862 \cdot 10^{-9}$
MPS_2D_50000	24.2	116	$4.751 \cdot 10^{-1}$	$4.806 \cdot 10^{-7}$
MPS_2D_100000	115.1	177	$1.964 \cdot 10^{+0}$	$8.619 \cdot 10^{-7}$
MPS_2D_200000	509.1	225	$2.845 \cdot 10^{+0}$	$8.258 \cdot 10^{-7}$
MPS_2D_400000	3087.2	387	$9.414 \cdot 10^{+0}$	$1.665 \cdot 10^{-6}$
LSQ_3D_10000	0.814	13	$8.407 \cdot 10^{-6}$	$5.757 \cdot 10^{-9}$
LSQ_3D_50000	9.41	21	$8.498 \cdot 10^{-5}$	$1.364 \cdot 10^{-8}$
LSQ_3D_100000	30.2	25	$4.178 \cdot 10^{-4}$	$3.628 \cdot 10^{-8}$
LSQ_3D_200000	144.8	31	$5.735 \cdot 10^{-4}$	$2.704 \cdot 10^{-8}$
MPS_3D_10000	0.63	24	$3.395 \cdot 10^{-5}$	$1.016 \cdot 10^{-8}$
MPS_3D_50000	8.28	38	$9.213 \cdot 10^{-4}$	$6.259 \cdot 10^{-8}$
MPS_3D_100000	34.2	49	$3.764 \cdot 10^{-3}$	$1.377 \cdot 10^{-7}$
MPS_3D_200000	142.6	62	$8.193 \cdot 10^{-3}$	$1.596 \cdot 10^{-7}$

Table 4.5. XPABLO+OBGP(10) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	0.441	22	$8.763 \cdot 10^{+0}$	$7.491 \cdot 10^{+1}$
CAVITY26	0.519	24	$4.452 \cdot 10^{+0}$	$8.458 \cdot 10^{+1}$
EX19	0.827	51	$3.904 \cdot 10^{+0}$	$1.307 \cdot 10^{+0}$
EX35	2.22	20	$2.432 \cdot 10^{+4}$	$2.742 \cdot 10^{+4}$
GARON1	0.241	14	$1.273 \cdot 10^{+0}$	$1.938 \cdot 10^{+2}$
GARON2	3.96	79	$1.162 \cdot 10^{+0}$	$4.029 \cdot 10^{+2}$
RAEFSKY2	0.702	21	$3.468 \cdot 10^{+0}$	na
RAEFSKY3	15.6	138	$3.925 \cdot 10^{+2}$	na
SHYY41	–	⋆	–	–
SHYY161	399.9	(350)	$7.604 \cdot 10^{-1}$	$7.086 \cdot 10^{-1}$
IGBT3	0.901	21	$7.036 \cdot 10^{+4}$	na
NMOS3	2.27	26	$5.603 \cdot 10^{+3}$	na
BARRIER2-1	1242.8	(1000)	$1.808 \cdot 10^{+4}$	na
PARA-4	102.2	39	$2.031 \cdot 10^{+4}$	na
PARA-8	79.9	28	$1.013 \cdot 10^{+0}$	na
OHNE2	335.4	128	$5.615 \cdot 10^{+5}$	na
2D_54019_HIGHK	20.0	68	$2.412 \cdot 10^{+1}$	na
3D_51448_3D	10.4	26	$9.733 \cdot 10^{-1}$	na
IBM_MATRIX_2	9.63	24	$9.746 \cdot 10^{-1}$	na
MATRIX_9	53.5	52	$7.785 \cdot 10^{+4}$	na
MATRIX-NEW_3	52.6	62	$2.640 \cdot 10^{+1}$	na
LSQ_2D_1000	0.024	1	$2.782 \cdot 10^{-10}$	$6.631 \cdot 10^{-14}$
LSQ_2D_2000	0.079	4	$2.346 \cdot 10^{-5}$	$1.991 \cdot 10^{-9}$
LSQ_2D_5000	0.294	9	$1.313 \cdot 10^{-5}$	$3.488 \cdot 10^{-10}$
LSQ_2D_10000	0.758	14	$1.876 \cdot 10^{-4}$	$2.143 \cdot 10^{-9}$
LSQ_2D_50000	9.89	31	$5.228 \cdot 10^{-3}$	$7.563 \cdot 10^{-9}$
LSQ_2D_100000	39.1	43	$2.149 \cdot 10^{-2}$	$1.315 \cdot 10^{-8}$
LSQ_2D_200000	177.9	61	$1.826 \cdot 10^{-1}$	$6.349 \cdot 10^{-8}$
LSQ_2D_400000	6025.3	89	$6.521 \cdot 10^{-1}$	$1.059 \cdot 10^{-7}$
MPS_2D_10000	0.831	30	$3.413 \cdot 10^{-3}$	$1.567 \cdot 10^{-8}$
MPS_2D_50000	17.9	82	$1.864 \cdot 10^{-1}$	$1.498 \cdot 10^{-7}$
MPS_2D_100000	71	103	$3.189 \cdot 10^{-1}$	$1.381 \cdot 10^{-7}$
MPS_2D_200000	506.5	180	$1.847 \cdot 10^{+0}$	$6.190 \cdot 10^{-7}$
MPS_2D_400000	2023.0	250	$7.218 \cdot 10^{+0}$	$1.487 \cdot 10^{-6}$
LSQ_3D_10000	0.872	9	$4.100 \cdot 10^{-6}$	$2.898 \cdot 10^{-9}$
LSQ_3D_50000	9.85	18	$1.893 \cdot 10^{-5}$	$3.044 \cdot 10^{-9}$
LSQ_3D_100000	30.7	22	$7.986 \cdot 10^{-5}$	$6.963 \cdot 10^{-9}$
LSQ_3D_200000	110.6	28	$3.625 \cdot 10^{-4}$	$1.713 \cdot 10^{-8}$
MPS_3D_10000	0.582	19	$6.261 \cdot 10^{-5}$	$1.909 \cdot 10^{-8}$
MPS_3D_50000	7.95	34	$4.915 \cdot 10^{-4}$	$3.303 \cdot 10^{-8}$
MPS_3D_100000	31.7	44	$2.570 \cdot 10^{-3}$	$9.394 \cdot 10^{-8}$
MPS_3D_200000	127.7	53	$4.993 \cdot 10^{-3}$	$9.790 \cdot 10^{-8}$

Table 4.6. XPABLO+OBGP(20) Solve Results for the Test Matrices

Matrix	Time	Iter	Rel. Res.	Rel. Err.
CAVITY16	0.411	13	$8.763 \cdot 10^{+0}$	$7.491 \cdot 10^{+1}$
CAVITY26	0.533	19	$4.452 \cdot 10^{+0}$	$8.458 \cdot 10^{+1}$
EX19	0.801	51	$3.904 \cdot 10^{+0}$	$1.307 \cdot 10^{+0}$
EX35	1.64	10	$2.432 \cdot 10^{+4}$	$2.742 \cdot 10^{+4}$
GARON1	0.261	10	$1.273 \cdot 10^{+0}$	$1.938 \cdot 10^{+2}$
GARON2	2.68	40	$1.162 \cdot 10^{+0}$	$4.029 \cdot 10^{+2}$
RAEFSKY2	0.859	13	$3.468 \cdot 10^{+0}$	na
RAEFSKY3	9.15	62	$3.925 \cdot 10^{+2}$	na
SHYY41	–	⋆	–	–
SHYY161	415.3	(350)	$8.853 \cdot 10^{-1}$	$8.309 \cdot 10^{-1}$
IGBT3	0.842	14	$7.036 \cdot 10^{+4}$	na
NMOS3	1.89	16	$5.603 \cdot 10^{+3}$	na
BARRIER2-1	1309.7	(1000)	$6.552 \cdot 10^{+5}$	na
PARA-4	87.3	31	$2.031 \cdot 10^{+4}$	na
PARA-8	79.7	27	$1.013 \cdot 10^{+0}$	na
OHNE2	161.3	53	$5.615 \cdot 10^{+5}$	na
2D_54019_HIGHK	15.6	47	$2.412 \cdot 10^{+1}$	na
3D_51448_3D	10.5	23	$9.733 \cdot 10^{-1}$	na
IBM_MATRIX_2	9.27	20	$9.746 \cdot 10^{-1}$	na
MATRIX_9	46.6	42	$7.785 \cdot 10^{+4}$	na
MATRIX-NEW_3	38.4	42	$2.640 \cdot 10^{+1}$	na
LSQ_2D_1000	0.070	1	$2.782 \cdot 10^{-10}$	$6.631 \cdot 10^{-14}$
LSQ_2D_2000	0.092	3	$6.778 \cdot 10^{-11}$	$6.546 \cdot 10^{-15}$
LSQ_2D_5000	0.325	6	$1.338 \cdot 10^{-4}$	$3.545 \cdot 10^{-9}$
LSQ_2D_10000	0.802	10	$1.244 \cdot 10^{-4}$	$1.396 \cdot 10^{-9}$
LSQ_2D_50000	9.1	23	$3.150 \cdot 10^{-3}$	$4.534 \cdot 10^{-9}$
LSQ_2D_100000	33.2	32	$1.099 \cdot 10^{-2}$	$6.697 \cdot 10^{-9}$
LSQ_2D_200000	178.1	46	$2.400 \cdot 10^{-2}$	$6.353 \cdot 10^{-9}$
LSQ_2D_400000	**oom**			
MPS_2D_10000	0.616	17	$6.522 \cdot 10^{-4}$	$3.167 \cdot 10^{-9}$
MPS_2D_50000	10.7	42	$2.388 \cdot 10^{-2}$	$1.455 \cdot 10^{-8}$
MPS_2D_100000	45.5	61	$3.319 \cdot 10^{-1}$	$1.146 \cdot 10^{-7}$
MPS_2D_200000	222.9	92	$3.601 \cdot 10^{-1}$	$8.148 \cdot 10^{-8}$
MPS_2D_400000	1354.0	146	$9.835 \cdot 10^{-1}$	$1.965 \cdot 10^{-7}$
LSQ_3D_10000	1.22	7	$6.277 \cdot 10^{-7}$	$4.416 \cdot 10^{-10}$
LSQ_3D_50000	11.4	13	$3.475 \cdot 10^{-5}$	$5.590 \cdot 10^{-9}$
LSQ_3D_100000	33.4	17	$1.699 \cdot 10^{-4}$	$1.480 \cdot 10^{-8}$
LSQ_3D_200000	1246.7	22	$1.533 \cdot 10^{-4}$	$7.249 \cdot 10^{-9}$
MPS_3D_10000	0.604	13	$8.171 \cdot 10^{-6}$	$2.447 \cdot 10^{-9}$
MPS_3D_50000	7.27	26	$4.827 \cdot 10^{-4}$	$3.258 \cdot 10^{-8}$
MPS_3D_100000	27.4	34	$1.279 \cdot 10^{-3}$	$4.679 \cdot 10^{-8}$
MPS_3D_200000	106.3	42	$3.819 \cdot 10^{-3}$	$7.469 \cdot 10^{-8}$

CHAPTER 5

PARTITIONING FOR UNSYMMETRIC PERMUTATIONS

In Chapter 3 we presented a preconditioning framework based on XPABLO in the following manner:

1. Compute a maximum product transversal, i.e., compute scaling matrices $\hat{R}$ and $\hat{C}$ and a permutation matrix Σ such that $\hat{A} = \Sigma\hat{R}A\hat{C}$ is an I-matrix.
2. Compute a permutation matrix P such that the blocks found by PABLO correspond to diagonal blocks in $P\hat{A}P^T$

Even if A is symmetric, $\hat{A}$ will be in general nonsymmetric. Therefore, there is no intrinsic reason to limit ourselves to symmetric permutations in step 2. Recall that the goal of XPABLO is to find blocks such that the diagonal blocks are relatively full and contain most of the large entries of the matrix and that the off-diagonal blocks are very sparse and do not contain many large entries. This fits closely to block Jacobi preconditioning. Our experiments (see section 3.7) have shown the usefulness of block Gauss–Seidel preconditioning based on the block structure determined by XPABLO. For this reason, it seems natural to try to modify XPABLO to find a block structure that fits more closely to block Gauss–Seidel preconditioning, i.e., to have a variant of XPABLO which concentrates most of the off-diagonal entries in the lower triangular part. In this chapter we will present our work toward such a modified

XPABLO algorithm. To have more freedom to move matrix entries to the lower triangular part, we decided to employ unsymmetric permutations, i.e., our modified algorithm tries to find permutation matrices P and Q such that $P\hat{A}Q$ is nearly block triangular and has a (relatively) dense block diagonal. The blocks should correspond to a partition found by our modified XPABLO. We call this approach Unsymmetric PABLO (UPABLO).

The basic idea behind UPABLO is simple: We use a modified version of XPABLO to find rectangular blocks. After we have found a block we split it into a square part, which we will permute onto the diagonal, and the remaining part, which we will permute into the strictly lower block triangular part of the (permuted) matrix $P\hat{A}Q$. The trick is to order the blocks such that tall blocks (more rows than columns) come before wide blocks (more columns than rows). This is illustrated in Figure 5.1.

We have to note though, that preliminary testing showed the new approach to perform poorly in practice. In fact, we did not find a problem for which UPABLO based block Gauss–Seidel preconditioning was clearly superior to XPABLO based block Gauss–Seidel preconditioning. Actually, in many cases the performance was noticeably worse. Nevertheless, we think that the graph theoretical model of the reordering problem presented in this chapter may be useful as a basis to find a PABLO-like algorithm that fits more closely to block Gauss–Seidel preconditioning than the XPABLO algorithm of Chapter 3. For the obvious reasons, we will not show numerical results.

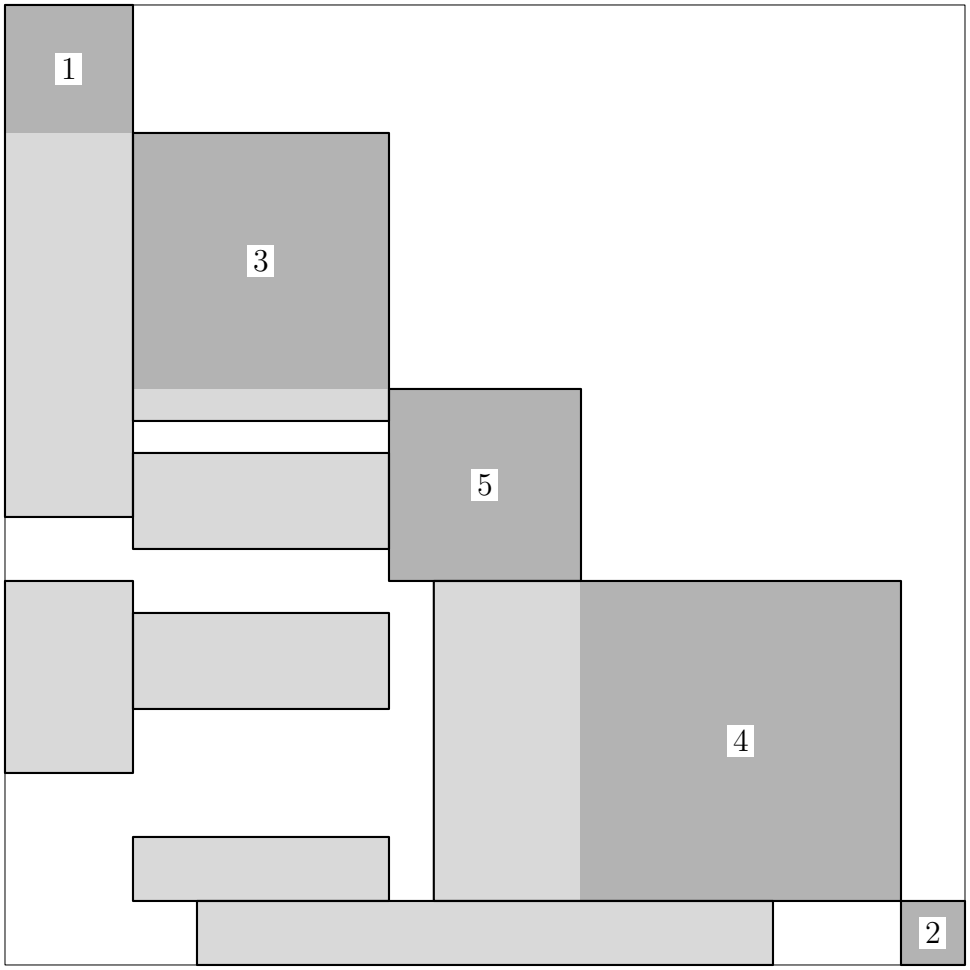


Figure 5.1. Ordering of Rectangular Blocks in UPABLO

The figure shows the matrix after being permuted according to the UPABLO block structure. The rectangular blocks found by the modified XPABLO algorithm are shown with a grey background. The square parts selected to become the diagonal blocks are shown with a darker shade of grey. The numbers indicate the order in which the blocks were found. Notice that the blocks are ordered in such a way that they are all completely inside the lower block triangular part of the matrix. The rectangular blocks do not need to be contiguous and we can not expect them to be contiguous in the permuted matrix.

5.1 More on Bipartite Graphs

The directed graph of a matrix is not well suited for this new problem of finding an unsymmetric permutation. A much more useful model is the bipartite graph of a matrix; see section 2.4.3 for the definition used here.

We will often refer to the subset of row nodes or the subset of column nodes of a given node set. The following notation will be used:

Definition 5.1: Let $\mathcal{B} = (V_r, V_c, E)$ be a bipartite graph and let $S \subseteq V_r \cup V_c$ be a set of nodes. The set S_r of row nodes in S and the set S_c of column nodes in S are defined as

$$S_r := S \cap V_r \qquad \text{and} \qquad S_c := S \cap V_c. \qquad \diamond$$

A subset of the nodes of the bipartite graph of a matrix induces a submatrix:

Definition 5.2: Let $A \in \mathbb{R}^{m \times n}$ be a matrix with bipartite graph $\mathcal{B} = (V_r, V_c, E)$. Let $S \subseteq V_r \cup V_c$ be a set of nodes with $S_r, S_c \neq \emptyset$. We write $A|_S$ to denote the $|S_r| \times |S_c|$ submatrix

$$A|_S := (a_{ij}), \quad i, j \text{ such that } r_i \in S_r,\ c_j \in S_c. \qquad \diamond$$

5.1.1 Partitions and Permutations

Definition 5.3: A partition of a bipartite graph $\mathcal{B} = (V_r, V_c, E)$ is a partition of the node set $V = V_r \cup V_c$. Let $\mathcal{V}_q = \{V_1, \ldots, V_q\}$ be such a partition of the bipartite graph $\mathcal{B} = (V_r, V_c, E)$. The corresponding *row partition* $\mathcal{R}_q = \{R_1, \ldots, R_q\}$ and *column partition* $\mathcal{C}_q = \{C_1, \ldots, C_q\}$ are defined by

$$R_i := V_i \cap V_r \qquad \text{and} \qquad C_i := V_i \cap V_c, \quad i = 1, \ldots, q,$$

respectively. $\diamond$

We define a special class of permutations for bipartite graphs, since we want that row nodes stay row nodes and that column nodes stay column nodes under the permutation.

Definition 5.4: Let $\mathcal{B} = (V_r, V_c, E)$ be a bipartite graph with $V = V_r \cup V_c$. A *b-permutation* on $\mathcal{B}$ is a V_r and V_c invariant permutation $\pi : V \to V$ on V, i.e., π is a permutation on V with $\pi(V_r) = V_r$ and $\pi(V_c) = V_c$. The permuted graph is defined as $\pi(\mathcal{B}) := (V_r, V_c, E')$ with $\{r_i, c_j\} \in E'$ if $\{\pi(r_i), \pi(c_j)\} \in E$.

We define the row permutation $\pi_r : V_r \to V_r$ and the column permutation $\pi_c : V_c \to V_c$ corresponding to the b-permutation π by $\pi_r := \pi|_{V_r}$ and $\pi_c := \pi|_{V_c}$. $\diamond$

To define permutations corresponding to a partition we use the permutations corresponding to a partition of a node set as in Definition 2.23.

Definition 5.5: Let $\mathcal{B} = (V_r, V_c, E)$ be a bipartite graph and let $V = V_r \cup V_c$. Let $\mathcal{V}_q$ be a partition of $\mathcal{B}$ with row and column partitions $\mathcal{R}_q$ and $\mathcal{C}_q$. Let $\bar{\pi}_r$ be the permutation corresponding to the partition $\mathcal{R}_q$ of the set V_r. Similarly, let $\bar{\pi}_c$ be the permutation corresponding to the partition $\mathcal{C}_q$ of the set V_c. The permutation $\pi : V \to V$ defined by

$$\pi(v) := \begin{cases} \bar{\pi}_r(v) & \text{if } v \in V_r, \\ \bar{\pi}_c(v) & \text{if } v \in V_c \end{cases}$$

is called the *b-permutation* $\pi : V \to V$ *corresponding to* $\mathcal{V}_q$. $\diamond$

We notice that π is well-defined, since it is obviously V_r and V_c invariant, i.e., π is a b-permutation in the sense of Definition 5.4. Moreover, since we consider the partition $\mathcal{V}_q$ to consist of ordered sets, the row permutation $\bar{\pi}_r$ and the column permutation $\bar{\pi}_c$ are unique. Hence, the b-permutation π is uniquely defined by $\mathcal{V}_q$.

We finally note that the permutation $\bar{\pi}_r$ in Definition 5.5 is identical to the row permutation $\pi_r = \pi|_{V_r}$ in Definition 5.4. The same goes for $\bar{\pi}_c = \pi|_{V_c} = \pi_c$.

Recall from Remark 2.28 that the undirected graph $\mathcal{G} = (V_r \cup V_c, E)$ associated with the bipartite graph $\mathcal{B}(A) = (V_r, V_c, E)$ of a matrix A is isomorphic to the undirected graph $\tilde{\mathcal{G}} = (\tilde{V}, \tilde{E})$ of the symmetric matrix

$$\tilde{A} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \in \mathbb{R}^{(m+n)\times(m+n)}.$$

Let f be the bijection $f : V_r \cup V_c \to \{1, \ldots, m+n\}$ defined by $f(r_1) = 1, \ldots, f(r_m) = m$ and $f(c_1) = m+1, \ldots, f(c_n) = m+n$; see Remark 2.28. Then $\tilde{V} = \{1, \ldots, m+n\} = f(V)$ and $\tilde{E} = f(E)$ where $f(E) = \{f(e) \,|\, e \in E\}$; cf. Definition 2.21.

Let $\mathcal{V}_q$ be a partition of $V = V_r \cup V_c$. Then $\tilde{\mathcal{V}}_q = f(\mathcal{V}_q) = \{f(V_1), \ldots, f(V_q)\}$ is a partition of $\tilde{V}$. The following Lemma will help us explore the question whether the b-permutation corresponding to $\mathcal{V}_q$ is related to the permutation corresponding to $\tilde{\mathcal{V}}_q$.

Lemma 5.6: Let $\mathcal{B} = (V_r, V_c, E)$ be a bipartite graph with $V = V_r \cup V_c$ and let $f : V \to \tilde{V}$ be the bijection that maps V to $\tilde{V} = \{1, \ldots, m+n\}$, where $m = |V_r|$ and $n = |V_c|$. Let $\mathcal{V}_q = \{V_1, \ldots, V_q\}$ be a partition of V. Then, the permutation $\pi : V \to V$ is the b-permutation corresponding to the partition $\mathcal{V}_q$ if and only if $\tilde{\pi} = f \circ \pi \circ f^{-1}$ is the permutation corresponding to the partition $\mathcal{W}_q = \{f(R_1), \ldots, f(R_q), f(C_1), \ldots, f(C_q)\}$ of the set $\tilde{V}$ in the sense of Definition 2.23.

Proof: "$\Rightarrow$": Let π be the b-permutation corresponding to the partition $\mathcal{V}_q$. Then π_r is the permutation corresponding to the partition $\mathcal{R}_q$ of V_r and hence $\tilde{\pi}_r = f \circ \pi_r \circ f^{-1}$ is the permutation corresponding to the partition $\{f(R_1), \ldots, f(R_q)\}$ of $f(V_r) = \{1, \ldots, m\}$. Similarly, $\tilde{\pi}_c = f \circ \pi_c \circ f^{-1}$ is the permutation corresponding to the partition $\{f(C_1), \ldots, f(C_q)\}$ of $f(V_c) = \{m+1, \ldots, m+n\}$. Together, this implies

the equality

$$\tilde{\pi}(v) = \begin{cases} \tilde{\pi}_r(v) & \text{if } v \leq m, \\ \tilde{\pi}_c(v) & \text{if } v > m. \end{cases}$$

Then $\tilde{\pi}$ is the permutation corresponding to the partition $\mathcal{W}_q$.

"$\Leftarrow$": Let $\tilde{\pi}$ be the permutation corresponding to the partition $\mathcal{W}_q$. Since the R_i, $i = 1, \ldots, q$, are disjoint and f is bijective we can observe that $\bigcup_{i=1}^{q} f(R_i) = f(V_r)$ is a disjoint union, i.e., $\tilde{\pi}$ is $f(V_r)$ invariant. Thus, $\tilde{\pi}_r := \tilde{\pi}|_{f(V_r)}$ maps $\{1, \ldots, m\}$ to the ordered set $f(R_1) \cup \cdots \cup f(R_q)$, i.e., $\tilde{\pi}_r$ is the permutation corresponding to the partition $f(\mathcal{R}_q) = \{f(R_1), \ldots, f(R_q)\}$. Similarly, $\tilde{\pi}_c := \tilde{\pi}|_{f(V_c)}$ is the permutation corresponding to the partition $f(\mathcal{C}_q) = \{f(C_1), \ldots, f(C_q)\}$. Then $\bar{\pi}_r = f^{-1} \circ \tilde{\pi}_r \circ f$ is the permutation corresponding to $\mathcal{R}_q$ and $\bar{\pi}_c$ is the permutation corresponding to $\mathcal{C}_q$, i.e., $\pi = f^{-1} \circ \tilde{\pi} \circ f$ is the b-permutation corresponding to $\mathcal{V}_q$. □

According to the Lemma, the b-permutation π corresponding to $\mathcal{V}_q$ is related to the permutation $\tilde{\pi}$ corresponding to $\mathcal{W}_q = \{f(R_1), \ldots, f(R_q), f(C_1), \ldots, f(C_q)\}$ by the relation

$$\tilde{\pi} \circ f = f \circ \pi.$$

The permutation $\pi' : \tilde{V} \to \tilde{V}$ corresponding to the partition $\tilde{\mathcal{V}}_q$ could be quite different, since $\tilde{\mathcal{V}}_q \neq \mathcal{W}_q$. Note that π' will be in general neither $f(V_r)$ nor $f(V_c)$ invariant.

We now want to discuss how a b-permutation corresponds to a permutation of a matrix.

Definition 5.7: Let $A \in \mathbb{R}^{m \times n}$ be a matrix with bipartite graph $\mathcal{B} = (V_r, V_c, E)$. Let π be a b-permutation on $\mathcal{B}$. Then $\pi(A)$ denotes the permuted matrix PAQ, where P is the permutation matrix corresponding to π_r and Q is the permutation matrix

corresponding to π_c, i.e.,

$$\big(\pi(A)\big)_{ij} = (PAQ)_{ij} = a_{\pi_r(i),\pi_c(j)}, \quad i \in V_r,\ j \in V_c.$$

If C is a submatrix of A with bipartite graph $\mathcal{B}' = (V_r', V_c', E')$ we write $\pi(C)$ to denote the corresponding submatrix of $\pi(A)$, i.e.,

$$\big(\pi(C)\big)_{ij} = a_{\pi_r(i),\pi_c(j)}, \quad i \in V_r' \subset V_r,\ j \in V_c' \subset V_c. \qquad \diamond$$

Lemma 5.8: If a matrix A is permuted in the sense of Definition 5.7, then the bipartite graph of the permuted matrix is equal to the b-permuted bipartite graph of the matrix, i.e., $\mathcal{B}(\pi(A)) = \pi(\mathcal{B}(A))$.

Proof: Let E be the set of edges of $\mathcal{B}(A)$, E_1 the set of edges of $\mathcal{B}(\pi(A))$ and E_2 the set of edges of $\pi(\mathcal{B}(A))$. It is sufficient to show $E_1 = E_2$. Let $e = \{r_i, c_j\} \in E_1$. Then

$$e \in E_1 \ \Leftrightarrow\ (\pi(A))_{i,j} \neq 0 \ \Leftrightarrow\ a_{\pi(i),\pi(j)} \neq 0 \ \Leftrightarrow\ \{r_{\pi(i)}, c_{\pi(j)}\} \in E \ \Leftrightarrow\ e \in E_2,$$

which finishes the proof. $\square$

Now, recall that $\mathcal{B}(A)$ is isomorphic to the undirected graph of

$$\tilde{A} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}.$$

In the same way $\mathcal{B}(\pi(A))$ is isomorphic to the undirected graph of

$$\begin{bmatrix} 0 & \pi(A) \\ \pi(A)^T & 0 \end{bmatrix} = \begin{bmatrix} 0 & PAQ \\ Q^T A^T P^T & 0 \end{bmatrix},$$

where P is the permutation matrix corresponding to π_r and Q is the permutation matrix corresponding to π_c.

Let Π be the permutation matrix

$$\Pi = \begin{bmatrix} P & 0 \\ 0 & Q^T \end{bmatrix}.$$

Lemma 5.9: The undirected graph $\mathcal{G}(\Pi\tilde{A}\Pi^T)$ and the bipartite graph $\mathcal{B}(PAQ)$ are isomorphic, i.e., $\mathcal{G}(\Pi\tilde{A}\Pi^T) \cong \mathcal{B}(PAQ)$.

Proof: The simple calculation

$$\Pi\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}\Pi^T = \begin{bmatrix} P & 0 \\ 0 & Q^T \end{bmatrix}\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}\begin{bmatrix} P^T & 0 \\ 0 & Q \end{bmatrix} = \begin{bmatrix} 0 & PAQ \\ (PAQ)^T & 0 \end{bmatrix}$$

shows that $\mathcal{G}(\Pi\tilde{A}\Pi^T)$ and $\mathcal{B}(PAQ)$ are isomorphic. □

Lemma 5.8 and Lemma 5.9 imply the isomorphisms

$$\mathcal{G}(\Pi\tilde{A}\Pi^T) \cong \mathcal{B}(PAQ) = \mathcal{B}(\pi(A)) = \pi(\mathcal{B}(A)).$$

Let π be the b-permutation corresponding to the partition $\mathcal{V}_q$. Let $\hat{A} = \pi(A)$ be the matrix permuted according to $\mathcal{V}_q$. Then $\hat{A}$ has a $q \times q$ block structure

$$\hat{A} = \begin{bmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qq} \end{bmatrix} \qquad \text{where } A_{ij} = A|_{R_i \cup C_j} = \hat{A}|_{\pi^{-1}(R_i \cup C_j)}.$$

Then $A_{ii} = A|_{V_i} = \hat{A}|_{\pi^{-1}(V_i)}$. Notice that the blocks are in general not square, not even the "diagonal" blocks A_{ii}. Therefore, the A_{ii} may not even be positioned along the diagonal of $\hat{A}$. This block structure is called the block structure induced by the partition $\mathcal{V}_q$.

5.1.2 Balance

Often we need bipartite graphs where the set of row nodes and the set of column nodes have the same size.

Definition 5.10: A bipartite graph $\mathcal{B} = (V_r, V_c, E)$ is called *balanced* if V_r and V_c have the same size, i.e., if $|V_r| = |V_c|$.

A subset $S \subset V$ of the nodes is called balanced if the induced subgraph $\mathcal{B}|_S$ is balanced.

A partition $\mathcal{V}_q = \{V_1, \ldots, V_q\}$ is called balanced if all sets V_i, $i = 1, \ldots, q$, are balanced. $\diamond$

Example 5.11: Let $\mathcal{B} = (V_r, V_c, E)$ with $V_r = \{r_1, \ldots, r_8\}$, $V_c = \{c_1, \ldots, c_8\}$ be the bipartite graph of the 8×8 matrix A. Let $\mathcal{V} = \{V_1, V_2, V_3\}$ with

$$V_1 = \{r_1, r_2, r_3, r_4, c_1, c_2\},$$

$$V_2 = \{r_5, r_6, c_3\},$$

$$\text{and} \quad V_3 = \{r_7, r_8, c_4, c_5, c_6, c_7, c_8\}$$

be a partition of $\mathcal{B}$. The partition $\mathcal{V}$ induces the following 3×3 block structure:

$$A = \begin{bmatrix}
\times & \times & \times & \times & & & \times \\
 & \times & & & \times & & \\
\times & & \times & \times & & & \\
\times & \times & \times & \mathbf{0} & & \times & \times \\
 & & \times & & \times & & \times \\
 & & \times & & & \times & \\
\times & \times & & & \times & \times & \mathbf{0} \\
 & \times & & \times & \times & & \times
\end{bmatrix}.$$

The diagonal entries are printed in bold with a slightly larger font. Note that the partition $\mathcal{V}$ is not balanced and that the blocks $A|_{V_1}$, $A|_{V_2}$, and $A|_{V_3}$ are not square. Moreover, some of the diagonal entries are not inside a block.

Lemma 5.12: Let $A \in \mathbb{R}^{m \times n}$ be a matrix with bipartite graph $\mathcal{B} = (V_r, V_c, E)$. Let $\mathcal{V}_q$ be a partition of $\mathcal{B}$. Then, the partition $\mathcal{V}_q$ is balanced if and only if the matrix A and all submatrices A_{ii}, $i = 1, \ldots, q$, induced by the partition are square.

Proof: If $\mathcal{V}_q$ is balanced then $|(V_i)_r| = |(V_i)_c|$ for all $i = 1, \ldots, q$. Therefore, the induced submatrices are square. Furthermore, $|V_r| = \sum_{i=1}^{q} |(V_i)_r| = \sum_{i=1}^{q} |(V_i)_c| = |V_c|$ and therefore the whole matrix A is square.

If an induced submatrix A_{ii} is square, then the corresponding node set V_i is balanced. If now all A_{ii}, $i = 1, \ldots, q$, are square, then the partition $\mathcal{V}_q$ is balanced.□

Proposition 5.13: Let $A \in \mathbb{R}^{m \times n}$ be a matrix with bipartite graph $\mathcal{B} = (V_r, V_c, E)$. Let $\mathcal{V}_q$ be a partition of $\mathcal{B}$. Let $\hat{A} = PAQ$ be the b-permuted matrix according to the partition $\mathcal{V}_q$. Then, the matrix $\mathrm{diag}(A_{11}, \ldots, A_{qq})$ is block diagonal if and only if $\mathcal{V}_q$ is balanced.

Proof: Let $D = \mathrm{diag}(A_{11}, \ldots, A_{qq})$. The matrix D is block diagonal if and only if all A_{ii}, $i = 1, \ldots, q$, are square. Using this, Lemma 5.12 shows that D is block diagonal if and only if $\mathcal{V}_q$ is balanced. □

Recall that we intend to find a block diagonal structure suitable for block Jacobi or block Gauss–Seidel preconditioning. Proposition 5.13 now tells us, that we need to find a balanced partition.

5.2 The Unsymmetric PABLO algorithm

Now we show how we modify XPABLO to produce a balanced partition of a bipartite graph. There are three modifications.

1. We let XPABLO run on the undirected graph associated with the bipartite graph. In this way we can reuse most of XPABLOs internal structure. There is one modification to XPABLO at this stage: After finishing a block, XPABLO updates the degrees of the nodes adjacent to some node inside the block. It may happen that the degree of a node becomes zero. We modify XPABLO to additionally add these nodes to the block, even if *maxbs* was already reached.

2. After a block is computed by XPABLO we step in and select a maximal balanced subset of the nodes in the block. We will discuss this in more details in section 5.2.1 and section 5.2.2.
3. We reorder the blocks such that the non-square part of the block that we chopped off will end up in the lower triangular part of the permuted matrix, i.e., we not only produce diagonal blocks, but also permute the matrix into a more triangular shape.

The exact working procedure is given in Algorithm 5.1. In the next sections we will give more details about modifications 2 and 3.

Remark 5.14: Modification 1 adds nodes with zero degree to the current block, even if *maxbs* was already reached. This is typically only a minor change to the way *maxbs* works. We assume for a moment that *maxbs* is not used to enforce a maximum block size. Let v be a node added to the block by the modified XPABLO, i.e., v has degree zero after the block B is removed from the graph. Just before B is removed, the node v has the property

$$\deg|_B(v) = \deg(v). \tag{5.1}$$

Since v is adjacent to B it gets tested for inclusion into B. Moreover, v gets tested at some point where (5.1) holds. If the connectivity criterion is used with $\beta \leq 1$, then the node gets added to the block. Therefore, with our typical choices of τ and β and with *maxbs* not in effect there are no nodes with property (5.1) after finishing a block. In this case, the modification only changes the way *maxbs* works as it allows some more nodes to slip through.

Before we discuss several possibilities to select a balanced subset of B we want to show that it will always be possible to do so.

1: **input:** A balanced bipartite graph $\mathcal{B} = (V_r, V_c, E)$
2: **output:** A balanced partition $\mathcal{V}_q = \{V_1, \ldots, V_q\}$ of $\mathcal{B}$
3: set $C := V_r \cup V_c$
4: set $\kappa := 0$ and $\lambda := 0$
5: **while** $C \neq \emptyset$ **do**
6: find block $B \subset C$ in $\mathcal{B}|_C$ using XPABLO {use modified XPABLO}
7: find maximal balanced subset $B' \subset B$
8: **if** $|(B \setminus B')_c| = 0$ **then** {no column nodes in $B \setminus B'$}
9: set $\kappa := \kappa + 1$
10: set $V_\kappa^L := B'$
11: **else** {no row nodes in $B \setminus B'$}
12: set $\lambda = \lambda + 1$
13: set $V_\lambda^R := B'$
14: **end if**
15: set $C := C \setminus B'$
16: **end while**
17: set $V_i := V_i^L$, $i = 1, \ldots, \kappa$, and $V_{\kappa+j} := V_{\lambda-j+1}^R$, $j = 1, \ldots, \lambda$
18: set $q := \kappa + \lambda$

Algorithm 5.1. The UPABLO Algorithm

Lemma 5.15: The node set B found in the UPABLO algorithm is either of size $|B| = 1$ or contains a nonempty balanced subset.

Proof: The set C is always balanced during any step of the UPABLO algorithm. It is initialized as a node set of a balanced bipartite graph, thus it is balanced in the beginning. Only at one place in the algorithm (in line 15) nodes are removed from C. Note that the set B' that is removed from C is always a balanced set. Therefore, C is always balanced. At the point where we select $B \subset C$ using XPABLO (in line 6), C is nonempty and hence $|C| \geq 2$. Now, without loss of generality, we may assume that the first node put into B is a row-node. Then the next nodes we check for inclusion are all column-nodes and further row-nodes can only be added after adding a column-node first. Thus we either add no additional node at all and get $|B| = 1$ or we have at least one row-node and one column-node in B. □

5.2.1 Selecting a Maximal Balanced Subset

One important step in UPABLO is to find a balanced subset of a given node set, see line 7 in Algorithm 5.1. We will first state this problem as a general problem in a bipartite graph of a matrix and present our approach to it. In section 5.2.2 we will consider the case that we additionally have to keep certain "fixed" nodes in the balanced subset.

Let $\mathcal{B} = (V_r, V_c, E)$ be the bipartite graph of the matrix A and let $S \subset V_r \cup V_c$ be a (non-balanced) set of nodes. This task is to select a maximal balanced subset $T \subset S$, i.e., to select a balanced subset having the maximal possible size $|T| = 2\min\{|S_r|, |S_c|\}$. The selection process should be fast and result in a T such that $A|_T$ is well-conditioned and, if possible, sparse relative to $A|_S$. With these two contradictory goals it is not possible to give one best algorithm for the selection problem. To simplify the discussion we assume—without loss of generality—that $|S_r| > |S_c|$, i.e., we assume that S contains more row nodes than column nodes.

To select a maximal balanced subset T of S we use a matching approach. We apply the matching algorithm MC64 described in section 3.2 to find a maximum matching M in $\mathcal{B}|_S$. If we assume $A|_S$ not to be structurally singular we have $|M| = |S_c|$ and the set T of nodes incident to the edges in M is a maximal balanced subset of P. Moreover, if we also apply the scaling computed by MC64 we can transform $A|_T$ into an I-matrix.

Alternatives: We considered some alternatives to the matching approach. We could employ a QR decomposition or an LU decomposition, both with pivoting. The pivot rows together with all columns would form the balanced subset.

A different alternative could be based on condition number estimations. We would select the set of rows, which give us the "best" condition number estimate.

5.2.2 Selecting a Maximal Balanced Subset Containing Fixed Nodes

In the previous section we have shown several approaches to select a maximal balanced subset $B' \subset B$. We now concentrate on the observation that an "unlucky" selection of nodes for the maximal balanced subset may restrict the selection of available nodes for later constructed diagonal blocks such that one or more of them will be structurally singular. Let B contain m row nodes and n column nodes. For the following discussion we assume that $m \geq n$. In the case $n > m$ we can simply work on the transposed problem. Since we have more row nodes than column nodes it follows that all column nodes of B will be in B'. Let

$$Z := \left\{ v \in B_r \;\middle|\; \deg|_{C \setminus B}(v) = 0 \right\}, \tag{5.2}$$

i.e., any node in Z is a row node which is only connected to column nodes in B and therefore has degree zero in $\mathcal{B}|_{C \setminus B}$. If such a node is not added into B' it will induce a zero row into the submatrix corresponding to $\mathcal{B}|_{C \setminus B}$ and therefore to the future block it will end up in. If we do not put such a node into B', we make at least one of the "later" diagonal block structurally singular. If this node is instead added to B' we may avoid this. Depending on the size of Z relative to the size of B_c there are several cases:

- If $|Z| = 0$, we select a balanced subset of B without fixing any nodes as described in section 5.2.1.
- If $|Z| \geq |B_c|$ we basically do the same as in the case $|Z| = 0$ only replacing B_r with Z. If we actually have $|Z| > |B_c|$, it will not be possible to put all nodes in Z into B', i.e., we will definitely get some singular diagonal block(s).
- If $0 < |Z| < |B_c|$ we indeed select a balanced subset of $B' \subset B$ with the constraint that $Z \subset B'$. The following two-step approach will find such a

balanced subset. In each step we use one of the methods from section 5.2.1 to select a balanced subset without having fixed nodes.

1. Select a maximal balanced subset T_1 of $Z \cup B_c$. Since $|Z| < |B_c|$ we have $Z \subset T_1$.
2. Select a maximal balanced subset T_2 of $B \setminus T_1$.

Then $B' = T_1 \cup T_2$ is a maximal balanced subset of B.

5.3 Discussion

An early prototype implementation of UPABLO did not contain any special technique to prevent the submatrix corresponding to the subgraph $\mathcal{B}|_{C \setminus B}$ from becoming structurally singular. But it turned out that this was a very common problem in practical experiments. Therefore we implemented the technique described in section 5.2.2 to protect against singular diagonal blocks. Even then the diagonal blocks were often very ill-conditioned. As an experiment we modified the algorithm to allow for balanced subsets with less than maximal size and implemented a subset selection in the following way: For each rectangular block we computed a pivoted QR factorization and dropped elements for which the diagonal entries of R became to small. Using this technique we were able to attain reasonable well-conditioned diagonal blocks, but the performance of the block Gauss–Seidel preconditioners based on these blocks was still poor in the sense that we did not improve the iteration count compared to an XPABLO-based block Gauss–Seidel preconditioner of similar size.

There are several points, which may contribute to the problems we have encountered with UPABLO:

First of all, using XPABLO on the undirected graph associated with the bipartite graph of the matrix is different from using XPABLO on the directed graph

of the matrix. Therefore, it could be that we would need different criteria and/or different parameter settings to find the kind of rectangular blocks we want to find.

Secondly, finding an unsymmetric permutation of the matrix can also be viewed as a matching problem, cf. section 3.2: We want to match each row in the matrix with a column in the matrix such that the corresponding permuted matrix has a lower triangular part which is heavier than the upper triangular part, i.e., we want the permutation corresponding to the matching to permute as many entries of the matrix as possible into the lower triangular part. We think that the UPABLO approach to this matching problem is in conflict with the maximum product transversal found by MC64. On the other hand, it is probably not a good idea to leave out MC64, since MC64 is so important to the performance of XPABLO.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We have seen in Chapter 3 that XPABLO can be used to construct robust preconditioners for many problems from computational fluid dynamics and semiconductor device simulation. In general they performed better than ILUTP-based preconditioners. The accuracy of solutions obtained using ILUTP-based was found to be quite low for the UF test problems, whereas the XPABLO-based preconditioners did not suffer in the same way. This phenomenon deserves further study. When using XPABLO for preconditioning we compute a sparse factorization of the diagonal blocks. This factorization limits the size of the blocks to be found by XPABLO. Using inexact solves on the diagonal blocks would allow for much larger blocks.

In Chapter 4, we have shown by experiments that the performance of a multiplicative Schwarz preconditioner without overlap, i.e., the performance of a block Gauss–Seidel preconditioner, can be improved by adding overlap to the diagonal blocks. We introduced with OBGP a new and fast algorithm to add overlap to existing blocks. The growing of a block is based purely on the entries of the matrix and can therefore be applied in an algebraic way. The work on OBGP could be continued in several ways: OBGP could be used on top of other graph partitioners than XPABLO, e.g., on top of Metis [34]. The covers found by OBGP could be used for

other Schwarz methods, e.g., for additive Schwarz and restricted additive Schwarz methods. A parallel version of OBGP could be developed.

The graph-theoretical basis for an unsymmetric version of XPABLO was developed in Chapter 5. Several aspects of UPABLO deserve further study, e.g., the question of finding a balanced subset of a bipartite node set and the question of better combining the UPABLO matching problem with the maximum product transversal found by MC64.

Using XPABLO and OBGP together is an useful and robust tool for preconditioning, but so far only in a single-processor setting. Developing a parallel version of XPABLO would face two big obstacles: First, doing the work of XPABLO in parallel. There is no obvious starting point for parallelization and several characteristics of XPABLO are intrinsically serial, e.g., with XPABLO each block depends on the previously found blocks and we do not know a priori which part of the graph is needed to find a specific block. Secondly, the partition found by XPABLO is in most cases not very useful in a parallel setting. In a parallel program we would want to balance the computational effort and reduce the idle times. Therefore, we usually would want XPABLO to find a given number of blocks such that all blocks are in some sense of similar size, both is not done by XPABLO so far. While the graph partitioning phase of XPABLO can not be parallelized without substantial changes to the algorithm, we could parallelize the preconditioner by employing asynchronous methods; see [31]. On the other hand, the role of XPABLO in a parallel application could be very different: XPABLO could be employed for the task of locally solving a (smaller) linear system. Furthermore, there is some potential for parallelization in the setup and application of the preconditioners based on a XPABLO reordering.

BIBLIOGRAPHY

[1] G. Alefeld, *Über die Durchführbarkeit des Gaußschen Algorithmus bei Gleichungen mit Intervallen als Koeffizienten*, Computing Supplementum, 1 (1977), pp. 15–19.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Society of Industrial and Applied Mathematics, Philadelphia, third ed., 1999.

[3] M. Arioli, I. Duff, and D. Ruiz, *Stopping criteria for iterative solvers*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 138–144.

[4] M. Benzi, *Preconditioning techniques for large linear systems: A survey*, Journal of Computational Physics, 182 (2002), pp. 418–477.

[5] M. Benzi, H. Choi, and D. B. Szyld, *Threshold ordering for preconditioning nonsymmetric problems*, in Scientific Computing, Proceedings of the Workshop, 10–12 March 1997, Hong Kong, G. Golub, S.-H. Lui, F. Luk, and R. Plemmons, eds., Singapore, 1997, Springer, pp. 159–165.

[6] M. Benzi, A. Frommer, R. Nabben, and D. B. Szyld, *Algebraic theory of multiplicative Schwarz methods*, Numerische Mathematik, 89 (2001), pp. 605–639.

[7] M. Benzi, J. C. Haws, and M. Tůma, *Preconditioning highly indefinite and nonsymmetric matrices*, SIAM Journal on Scientific Computing, 22 (2000), pp. 1333–1353.

[8] M. Benzi, W. Joubert, and G. Mateescu, *Numerical experiments with parallel orderings for ILU preconditioners*, Electronic Transactions on Numerical Analysis, 8 (1999), pp. 88–114.

[9] M. Benzi, D. B. Szyld, and A. van Duin, *Orderings for incomplete factorization preconditionings of nonsymmetric problems*, SIAM Journal on Scientific Computing, 20 (1999), pp. 1652–1670.

[10] M. Benzi and M. Tůma, *Orderings for factored approximate inverse preconditioning*, SIAM Journal on Scientific Computing, 21 (2000), pp. 1851–1868.

[11] A. Berman and R. J. Plemmons, *Nonnegative Matrices in the Mathematical Sciences*, Classics in Applied Mathematics, SIAM, Philadelphia, PA, USA, 1994. Corrected republication, with supplement, of work first published in 1979 by Academic Press, New York.

[12] M. Bollhöfer and Y. Saad, *ILUPACK - preconditioning software package, release v1.0, May 14, 2004*. Available online at `http://ilupack.tu-bs.de/`.

[13] A. Bunse-Gerstner and R. Stöver, *On a conjugate gradient-type method for solving complex symmetric linear systems*, Linear Algebra and its Applications, 287 (1999), pp. 105–123.

[14] H. Choi and D. B. Szyld, *Application of threshold partitioning of sparse matrices to Markov chains*, in IEEE International Computer Performance and Dependability Symposium IPDS'96, Urbana-Champaign, Illinois, IEEE Computer Society Press, Los Alamitos, California, Sept. 1996, pp. 158–165.

[15] H. Choi and D. B. Szyld, *Threshold ordering for preconditioning nonsymmetric problems with highly varying coefficients*, Tech. Rep. 96-51, Department of Mathematics, Temple University, Philadelphia, May 1996.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 3rd ed., 2009.

[17] T. A. Davis, *University of Florida sparse matrix collection.* Available online at `http://www.cise.ufl.edu/research/sparse/matrices/`.

[18] ——, *Algorithm 8xx: UMFPACK: an Unsymmetric-Pattern Multifrontal Method*, ACM Transactions on Mathematical Software, 30 (2004). Available online at `http://www.cise.ufl.edu/research/sparse/umfpack/`.

[19] ——, *Direct Methods for Sparse Linear Systems*, Fundamentals of Algorithms, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2006.

[20] I. S. Duff, *MA57 - a code for the solution of sparse symmetric definite and indefinite systems*, ACM Transaction on Mathematical Software, 30 (2004), pp. 118–144.

[21] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Monographs on Numerical Analysis, Clarendon Press, Oxford, UK, 1986.

[22] I. S. Duff and J. 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 (1999), pp. 889–901.

[23] ——, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM Journal on Matrix Analysis and Applications, 22 (2001), pp. 973–996.

[24] I. S. Duff and J. A. Scott, *A parallel direct solver for large sparse highly unsymmetric linear systems*, ACM Transaction on Mathematical Software, 30 (2004), pp. 95–117.

[25] I. S. Duff and B. Uçar, *Combinatorial problems in solving linear systems*, Tech. Rep. RAL-TR-2009-016, Rutherford Appleton Laboratory, Aug. 2009.

[26] L. C. Dutto, W. G. Habashi, and M. Fortin, *Parallelizable block diagonal preconditioners for the compressible Navier-Stokes equations*, Computer Methods in Applied Mechanics and Engineering, 117 (1994), pp. 15–47.

[27] S. C. Eisenstat, *Efficient implementation of a class of preconditioned conjugate gradient methods*, SIAM Journal on Scientific and Statistical Computing, 2 (1981), pp. 1–4.

[28] S. Fischer, *Über auf Takagi-Faktorisierungen beruhende Präkonditionierer für CSYM (On Takagi-factorization based preconditioners for CSYM)*, PhD thesis, Department of Mathematics and Science, University of Wuppertal, Germany, March 2006. In German.

[29] D. Fritzsche, *Graph theoretical methods for preconditioners*, Master's thesis, Department of Mathematics and Science, University of Wuppertal, Germany, June 2004.

[30] D. Fritzsche, A. Frommer, and D. B. Szyld, *Extensions of certain graph-based algorithms for preconditioning*, SIAM Journal on Scientific Computing, 29 (2007), pp. 2144–2161.

[31] A. Frommer and D. B. Szyld, *On asynchronous iterations*, Journal of Computational and Applied Mathematics, 123 (2000), pp. 201–216.

[32] A. George and J. W. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[33] A. Greenbaum, *Iterative methods for solving linear systems*, vol. 17 of Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[34] G. Karypis and V. Kumar, *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*, Sept. 1998. User Manual, University of Minnesota, Department of Computer Science. Available online at `http://www.cs.umn.edu/~karypis/metis/metis/files/manual.ps`.

[35] M. Olschowka and A. Neumaier, *A new pivoting strategy for Gaussian elimination*, Linear Algebra and its Applications, 240 (1996), pp. 131–151.

[36] J. O'Neil and D. B. Szyld, *A block ordering method for sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 11 (1990), pp. 811–823.

[37] J. M. Ortega, *Efficient implementations of certain iterative methods*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 882–891.

[38] ——, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.

[39] C. C. Paige and Z. Strakos, *Residual and backward error bounds in minimum residual Krylov subspace methods*, SIAM Jourbal on Scientific Computing, 23 (2002), pp. 1898–1923.

[40] J. K. Reid and J. A. Scott, *Guidelines for the development of HSL software, 2008 version*, Tech. Rep. RAL-TR-2008-027, Rutherford Appleton Laboratory, Aug. 2008.

[41] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, second ed., 2003.

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

[43] O. Schenk, S. Röllin, and A. Gupta, *The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23 (2004), pp. 400–411.

[44] H. A. Schwarz, *Über einen Grenzübergang durch alternierendes Verfahren*, Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, 15 (1870), pp. 272–286.

[45] B. Seibold, *Minimal positive stencils in meshfree finite difference methods for the poisson equation*, Computer Methods in Applied Mechanics and Engineering, 198 (2008), pp. 592–601.

[46] V. Simoncini and D. B. Szyld, *Recent computational developments in Krylov subspace methods for linear systems*, Numerical Linear Algebra with Applications, 14 (2007), pp. 1–59.

[47] B. F. Smith, P. E. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, New York, 1996.

[48] A. Toselli and O. B. Widlund, *Domain Decomposition Methods - Algorithms and Theory*, vol. 34 of Springer Series in Computational Mathematics, Springer, Berlin Heidelberg, 2005.

[49] H. F. Walker, *Implementation of the GMRES method using Householder transformations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 152–163.

APPENDIX A

NOTATION

$\langle A \rangle$	comparison matrix of A; see Definition 2.4, p. 16
ϵ_M	the machine precision (machine epsilon).
adj(V)	set of nodes adjacent to V; see Definition 2.15, p. 19.
$\mathcal{B}(A)$	bipartite graph of matrix A; see Definition 2.27, p. 23.
deg(v)	degree of vertex (node) v. All incoming and outgoing edges are counted; see Definition 2.18, p. 21.
$\mathcal{G}(A)$	directed graph of matrix A; see Definition 2.13, p. 19.
inc(v)	set of edges incident to v; see Definition 2.14, p. 19.
inc(S, T)	set of edges incident to both S and T; see Definition 2.17, p. 20.
$L_k(S)$	kth level set with respect to S; see Definition 4.14, p. 92.
nnz(A)	number of nonzero entries in matrix A; see p. 1.