UNIVERSITY OF CRETE

Performance Comparison of Preconditioners & Solvers for Big, Sparse, Complex Symmetric Linear Systems

Stavros Avgerinos

Master of Science

School of Sciences and Engineering
Department of Applied Mathematics

July 2015

"This is your life, and it's ending one minute at a time"

Tyler Durden

Dedicated to my parents Ntina and Giorgos

Αφιερώνεται στους γονείς μου Ντίνα και Γιώργο

Abstract

The aim of this thesis is to investigate and compare two solvers for big, sparse, complex, symmetric, linear system of equations. Large sparse linear systems of equations appear in most applications of scientific computing. In particular, discretization of PDEs with the finite element method (FEM) or with the finite difference method (FDM) leads to such problems. First we investigate the PARDISO package, a high-performance, memory efficient direct solver. Then we analyze the QMRPACK, an iterative solver implementing several of the QMR algorithms. More specifically, we use the QMR algorithm based on the look-ahead coupled two-term recurrence Lanczos process and the simplified no-look-ahead version of the same algorithm, both equipped with ILUT and SSOR preconditioners. In conclusion, we compare the runtime between the two packages with test matrices formed by numerical methods.

Acknowledgements

I would like to express my gratitude to my supervisor Michael Plexousakis for the useful comments, remarks and engagement through the learning process of this master thesis. Working with him has been a very rewarding experience on several levels.

Many thanks to my colleague Dimitris Gourzoulidis who saved me a lot of time by providing me useful advices and some of the test matrices used for the numerical experiments.

Crete was not only a place to do graduate work, but also to meet new people and make new friends. All of them, however, made life here much more pleasant and fun. In particular, I would like to thank Apostolis K., Konstantinos Z., Nikos S., Alexandros P., Anastasios S., Michael M. and George M.

Also I would like to thank my hometown friends, George K., George M., Alexandros K. and Stavros B. for their encouragement and belief in me during my years in Crete.

Many thanks to Maria T. and Irene M., staff members, for their valuable help. They would always allocate some time to discuss with me and provide helpful advises.

Most importantly, I would like to thank my parents , Ntina and Giorgos, and my brother Ioannis who guided me in their own way throughout these past few years. They were always encouraging me to continue and finish up what I had started. I thank them for their love and constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The finite element and the finite differences methods are used widely in the numerical solution of partial differential equations. A common aspect of these methods is the requirement of a fast and efficient linear system solver. In this thesis our goal is to compare two linear system solvers, namely the PARDISO direct solver and the iterative QMR algorithm contained in the QMRPACK suite. PARDISO is a direct, parallel and memory efficient solver while QMRPACK is an implementation of iterative QMR algorithms.

In Chapter (2) we present the matrices used in testing solvers and some efficient storage schemes. In Chapter (3) we present the solvers. For each one separately we analyze extensively the methods and the algorithms used by the solvers in order to compute the solution of the linear systems. In Chapter (4) we discuss the preconditioners implemented in QMRPACK and their characteristics. Finally in Chapter (5) we present the numerical experiments and the comparison results.

Chapter 2

Matrix Analysis

2.1 Identities

We consider a general complex symmetric system of linear equations

$$Ax = b, \qquad x, b \in \mathbb{C}^n,\ A \in \mathbb{C}^{n \times n},$$

where A is an $n \times n$ complex symmetric but non-Hermitian matrix ($A \neq \bar{A}^T,\ A = A^T$). The matrix A can be written as $A = B + iC$. The sparsity structure of the matrix A may be as in Figure 2.1.

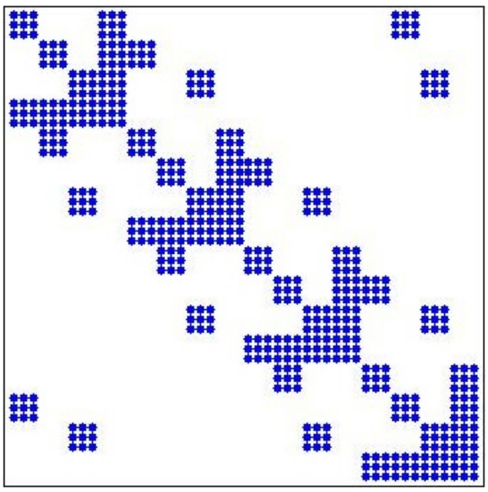


FIGURE 2.1: Sparsity pattern of a matrix formed by Galerkin's method.

In Figure 2.1 we can clearly see the blocks which form the matrix. In order to understand better the structure of the matrix we define the second order elliptic model problem

$$\begin{cases} -\Delta u = f(x,y), & (x,y) \in \Omega \\ u = 0, & (x,y) \in \partial\Omega \end{cases}$$

Let $0 < h \leq 1$ be a spatial discretization parameter and denote by $\mathcal{T}_h = \{K_i^h,\ i = 1, \ldots, d_h$ a partition of Ω consisting of triangles. A peripheral triangle K_i^{*h}, that one intersecting the boundary $\partial\Omega$, may have curved side. Denote the boundary of K_i^h by ∂K_i^h and set $\partial K_{ij}^h = \partial K_i^h \cap \partial K_j^h,\ i,j = 1, \ldots, d_h,\ \partial K_i^{*h} = \partial K_i^h \cap \partial K$ and let $\mathcal{N}_i = \{j : \partial K_{ij}^h$ is a line segment$\}$. Then $|\mathcal{N}_i|$ is the number of neighbours of K_i^h.

We consider the bilinear form

$$\begin{aligned} a_h^\gamma = \sum_{m=1}^{d_h} \Big[& (\nabla u^{(m)}, v^{(m)})_{K_m} \\ & + [-(\frac{\partial u^{(m)}}{\partial n}, v^{(m)})_{\partial K_m^*} - (\frac{\partial v^{(m)}}{\partial n}, u^{(m)})_{\partial K_m^*} + \gamma |\partial K_m^*|^{-1} (u^{(m)}, v^{(m)})_{\partial K_m^*}] \\ & + \sum_{p \in \mathcal{N}_m} \tau_{mp} [-(\frac{\partial u^{(m)}}{\partial n}, v^{(m)} - v^{(p)})_{\partial K_{mp}} - (\frac{\partial v^{(m)}}{\partial n}, u^{(m)} - u^{(p)})_{\partial K_{mp}} \\ & + \gamma |\partial K_{mp}|^{-1} (u^{(m)} - u^{(p)}, v^{(m)} - v^{(p)})_{\partial K_{mp}}] \Big] \end{aligned}$$

and define the matrices $S^{(i,j)}$ by

$$S_{l,k}^{(i,j)} = a_h^\gamma(\phi_k^{K_j}, \phi_l^{K_i}), \qquad 1 \leq l,k \leq N_r, \qquad 1 \leq i,j \leq d_h$$

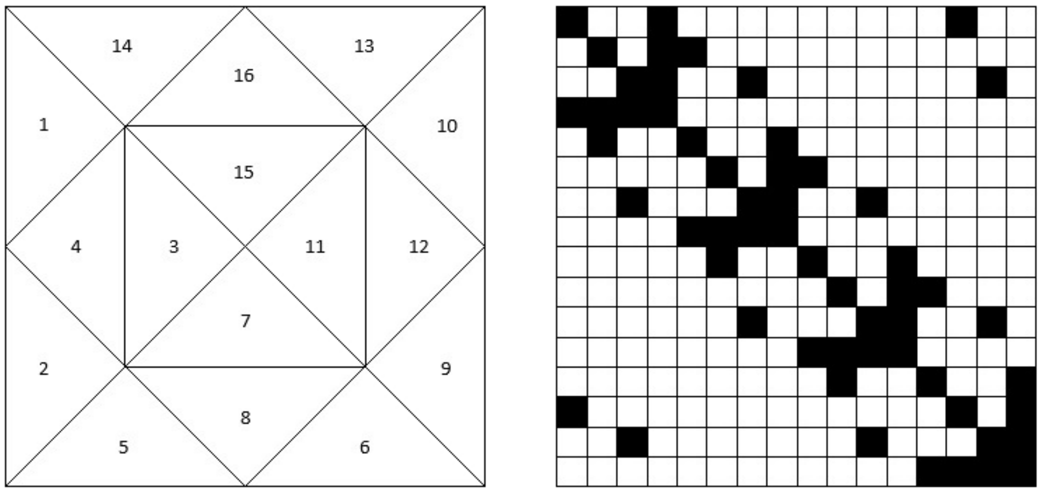
where $\{\hat{\phi}_j\}_{j=1}^{N_r}$ a basis of $\mathbb{P}^r$.

From the definition of a_h^γ it follows that the matrix S, whose (i,j) entry is $S^{(i,j)}$, is block symmetric. Moreover S is sparse, since $S^{(i,j)} \equiv 0$, unless $i = j$ or $j \in \mathcal{N}_i$. Hence S may have at most three nonzero off-diagonal entries per row. By symmetry, it is enough to consider the entries $S^{(i,i)}$ and $S^{(i,j)}$ for $1 \leq j < i \leq d_h$ and $j \in \mathcal{N}_i$. For the diagonal blocks we have

$$
\begin{aligned}
S_{l,k}^{(i,i)} &= (\nabla \phi_k^{K_i}, \nabla \phi_l^{K_i})_{\partial K_i} \\
&+ [-(\frac{\partial \phi_k^{K_i}}{\partial n}, \phi_l^{K_i})_{\partial K_i^*} - (\frac{\partial \phi_l^{K_i}}{\partial n}, \phi_k^{K_i})_{\partial K_i^*} + \gamma |\partial K_i^*|^{-1} (\phi_k^{K_i}, \phi_l^{K_i})_{\partial K_i^*}] \\
&+ \sum_{p \in \mathcal{N}_i} \tau_{ip} [-(\frac{\partial \phi_k^{K_i}}{\partial n}, \phi_l^{K_i})_{\partial K_{ip}} - (\frac{\partial \phi_l^{K_i}}{\partial n}, \phi_k^{K_i})_{\partial K_{ip}}] \\
&+ \sum_{p \in \mathcal{N}_i} [\gamma |\partial K_{ip}|^{-1} (\phi_k^{K_i}, \phi_l^{K_i})_{\partial K_{ip}}]
\end{aligned}
$$

For $j \in \mathcal{N}_i$ and $j < i$ we have

$$
S_{l,k}^{(i,j)} = \big(\frac{\partial \phi_l^{K_i}}{\partial n}, \phi_k^{K_j}\big)_{\partial K_{ij}} - \gamma |\partial K_{i,j}|^{-1} (\phi_l^{K_i}, \phi_k^{K_j})_{\partial K_{ij}}, \qquad 1 \le l, k \le N_r
$$



(a) (b)

FIGURE 2.2: A triangulation of the domain Ω (a) and the corresponding matrix structure (b).

2.2 Storage Format

A sparse data structure represents a matrix in space proportional to the number of non-zero entries. Many storage formats have been proposed to represent sparse matrices. The objective of storage formats for sparse matrices is to best exploit certain matrix properties by reducing memory space, by storing only non-zero elements of a sparse matrix, and by storing these elements in contiguous memory locations for more efficient execution of operations on the matrix data.

From an implementation point of view, there are two categories of storage formats. The notion of the point entry is used to identify storage formats where each entry in the storage format is a single element of the matrix. A block entry refers to storage formats where each entry defines a dense block of elements of any two dimensions. For both cases, programming languages provide static and dynamic data structures. In this thesis we use the CSR point entry matrix storage format. In CSR, the non-zero elements of every row in the matrix and their column indices are stored respectively in two vectors, A and JA. Another vector, IA stores the locations, in JA, of the first element of each row. The storage requirements are one vector, of length equal to the number of rows (NNT), and two additional vectors of length $NZMAX$ (number of non-zero elements).

Assume that A is an $NNT \times NNT$ matrix with $NZMAX$ non-zero elements. In the case where the matrix is symmetric we store only the upper or the lower triangle of the matrix. Thereby the number of the elements stored is reduced from $NZMAX$ (unsymmetric case) to $\frac{1}{2}(NZMAX - NNT) + NNT$ (symmetric case) if the diagonal of the matrix is full.

2									7
	3			-2					
		-1						5	
			4			-3			
2				7					
					2		-4		
		-1				-3			
6					4		-9		
								1	
		-3							-5

(a)

2						5			7
	3			-2					
		-1						5	
			4			-3			
				7				2	
					2		-4		
						-3			1
							-9		
								1	
									-5

(b)

FIGURE 2.3: Sparse non-symmetric linear system (a) and the upper triangular part of a symmetric linear system (b).

In Table 2.1 we can see the differences between storing an unsymmetric and an symmetric matrix.

Many sparse matrix storage formats are in use today. Some of the most popular sparse matrix storage structures are the Coordinate Format (COO), Compressed Sparse Column (CSC), Block Sparse Row/Column (BSR/BSC), Rutherford-Boeing (RB), Modified Sparse Row (MSR), Linked List (LIL) and the Matlab Sparse Matrix scheme. We refer the reader to [18] for an extensive comparison and analysis of storage formats for sparse matrices.

N	Non-symmetric Matrix			Symmetric Matrix		
	IA(N)	JA(N)	A(N)	IA(N)	JA(N)	A(N)
1	1	1	2	1	1	2
2	3	10	7	4	7	5
3	5	2	3	6	10	7
4	7	5	-2	8	2	3
5	9	3	-1	10	5	-2
6	11	9	5	12	3	-1
7	13	4	4	14	9	5
8	15	7	-3	16	4	4
9	18	1	2	17	7	-3
10	19	5	7	18	5	7
11	21	6	2	19	9	2
12		8	-4		6	2
13		3	-1		8	-4
14		7	-3		7	-3
15		1	6		10	1
16		6	4		8	-9
17		8	-9		9	1
18		9	1		10	-5
19		3	-3			
20		10	5			

TABLE 2.1: Illustration of the two *CSR* formats of Figure 2.3 matrices.

Chapter 3

Solvers

3.1 PARDISO

3.1.1 Introduction

The package PARDISO [16] (PARallel DIrect SOlver) is a high-performance, memory efficient and easy to use software for solving large, sparse, symmetric or non-symmetric linear systems of equations. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques. In order to improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update, and pipelining parallelism is exploited with a combination of left- and right-looking supernode techniques. The parallel pivoting methods allow complete supernode pivoting in order to balance numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory and distributed-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed. The approach is based on OpenMP directives and MPI parallelization.

PARDISO supports a wide range of sparse matrix types and computes the solution of real or complex, symmetric, structurally symmetric or un-symmetric, positive definite, indefinite or Hermitian sparse linear systems of equations on shared or distributed memory architectures. In this thesis we use the PARDISO solver to calculate the solution of a sparse, complex symmetric linear system. For further information about the supported matrix types we refer the reader to [17].

3.1.2 Symmetric Positive Definite Matrices

3.1.2.1 The Permutation Matrix P

Consider the $n \times n$ real, symmetric system of equations

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

where n is large and the matrix A is sparse. When A is factored using Cholesky's method, it normally suffers some fill-in. Since PAP^T is also symmetric and positive definite, for any permutation matrix P, we can instead solve the reordered system

$$(PAP^T)(Px) = Pb. \tag{3.2}$$

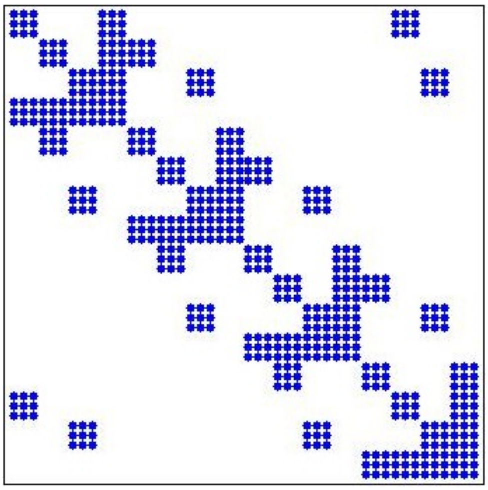
We rewrite (3.2) as

$$\widetilde{A}\widetilde{x} = \widetilde{b} \tag{3.3}$$

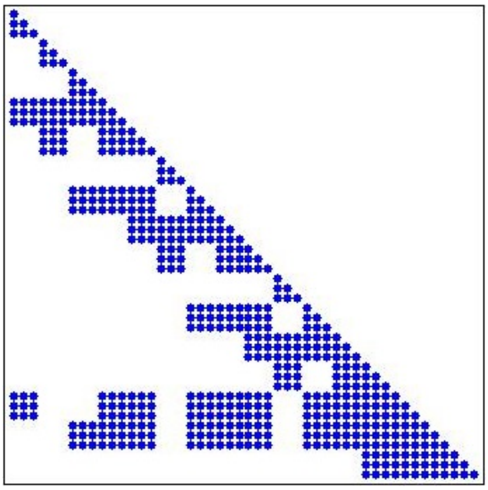
where $\widetilde{A} = PAP^T$, $\widetilde{x} = Px$ and $\widetilde{b} = Pb$.

The solver first computes a symmetric fill-in reducing permutation matrix P. The choice of P can have a dramatic effect on the amount of fill-in that occurs during the factorization. Thus, it is standard practice to reorder the rows and columns of the matrix before performing the factorization. This results in reduced storage requirements and means that the Cholesky factor, or sometimes an incomplete Cholesky factor used as a preconditioner can be applied with fewer arithmetic operations. The problem of finding the best ordering for A in the sense of minimizing the fill is computationally intractable (NP-complete problem). One of the most effective heuristic algorithm is the minimum degree algorithm. Another heuristic algorithm that can be used for the same procedure is the nested dissection algorithm from the METIS package.

The minimum degree algorithm is an algorithm used to permute the rows and columns of a symmetric sparse matrix, to reduce the number of non-zeros in the Cholesky factor. The minimum degree algorithm can be described as follows. Generally, it works only with the structure of A and simulates in some manner the n steps of symmetric Gaussian



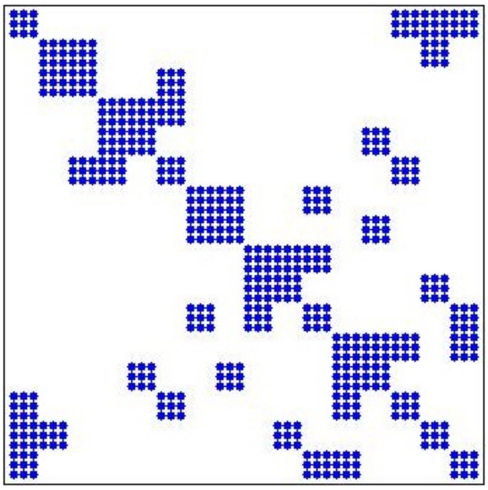
(a) Matrix A (504 non-zero elements)



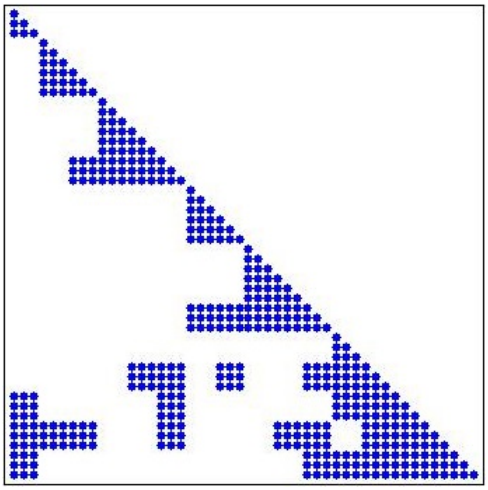
(b) Cholesky factor L (483 non-zero elements)

FIGURE 3.1: Sparsity pattern of a matrix A formed by Galerkin's method without any permutation of the rows and columns (a) and the corresponding Cholesky factor L (b).

elimination. At each step, a row and corresponding column interchange is applied to the part of the matrix remaining to be factored so that the number of non-zeros in the pivot row and column is minimized. Note that since the structure of the matrix is symmetric, the number of non-zeros in the pivot row and pivot column is the same. After n steps, the entire factorization has been simulated, and the order in which the pivot rows and columns were chosen is the ordering. For an extended analysis of the minimum degree algorithm we refer the reader to [7].



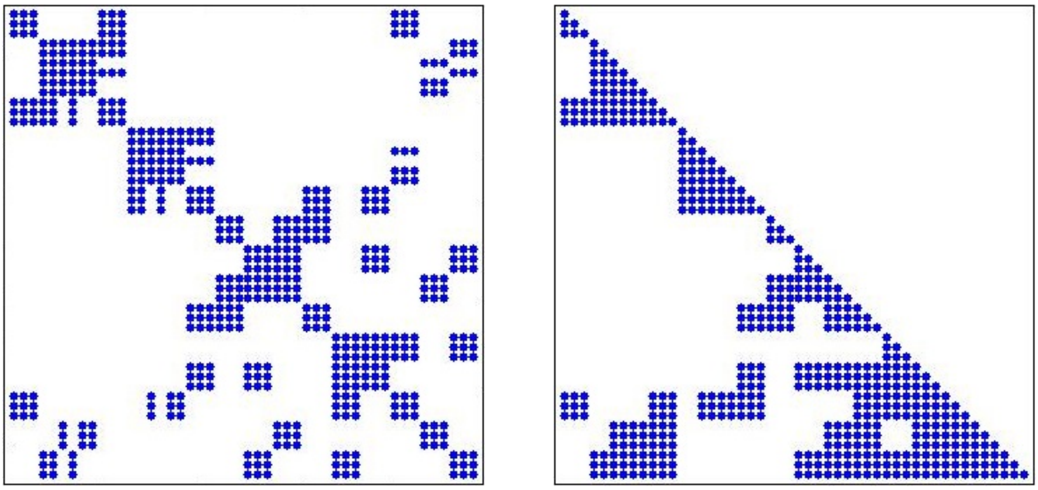
(a) Matrix PAP^T (504 non-zero elements)



(b) Cholesky factor L (402 non-zero elements)

FIGURE 3.2: Sparsity pattern of the matrix PAP^T (A formed by Galerkin's method) with permutation matrix P based on the minimum degree algorithm (a) and the corresponding Cholesky factor L (b).

Nested dissection is an algorithm for preserving sparsity in Gaussian elimination on symmetric positive definite matrices. Nested dissection can be viewed as a recursive divide-and-conquer algorithm on an undirected graph. It uses separators in the graph, which are small sets of vertices whose removal divides the graph approximately in half. The basic idea behind the multilevel graph partition algorithms implemented in METIS is very simple. The graph is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, and then this partition is projected back towards the original graph (finer graph), by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut. For further information about the METIS package and the nested dissection algorithm we refer the reader to [6] [9].



(a) Matrix PAP^T (504 non-zero elements) (b) Cholesky factor L (462 non-zero elements)

FIGURE 3.3: Sparsity pattern of the matrix PAP^T (A formed by Galerkin's method) with permutation matrix P based on the nested dissection algorithm (a) and the corresponding Cholesky factor L (b).

3.1.2.2 Cholesky Factorization

The standard approach for the inversion of a positive definite matrix is to perform first the LL^T factorization of the permuted matrix $\widetilde{A} = PAP^T$. We write out the equation $\widetilde{A} = LL^T$, for example in the 3×3 case

$$\begin{pmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \begin{pmatrix} L_{1,1} & L_{2,1} & L_{3,1} \\ & L_{2,2} & L_{3,2} \\ & & L_{3,3} \end{pmatrix}$$

$$= \begin{pmatrix} L_{1,1}^2 & L_{2,1}L_{1,1} & L_{3,1}L_{1,1} \\ L_{2,1}L_{1,1} & L_{2,1}^2 + L_{2,2}^2 & L_{3,1}L_{2,1} + L_{3,2}L_{2,2} \\ L_{3,1}L_{1,1} & L_{3,1}L_{2,1} + L_{3,2}L_{2,2} & L_{3,1}^2 + L_{3,2}^2 + L_{3,3}^2 \end{pmatrix}$$

and obtain the following formulate for the entries of L

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}, \qquad j = 1, 2, \ldots, n$$
$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right), \qquad i = j+1, j+2, \ldots, n$$

We rewrite (3.3)

$$LL^T \widetilde{x} = \widetilde{b} \tag{3.4}$$

3.1.2.3 Computing the Solution Vector

The PARDISO algorithm computes all the elements $x_i, \ i = 1, 2, \ldots n$, following the steps below:

1. Solve $Ly = \widetilde{b}$, by forward substitution
2. Solve $L^T \widetilde{x} = y$, by back substitution
3. Compute $x = P^T \widetilde{x}$

3.1.3 Symmetric Indefinite Matrices

3.1.3.1 Block LDL^T Factorization

A symmetric, possibly non-Hermitian matrix $A \in \mathbb{C}^{n \times n}$ can be factored into LDL^T, where L is unit lower triangular and D is block diagonal with each block of order 1 or 2. This is a generalization of the Cholesky factorization, which requires positive semidefiniteness. The process is described in [8] as follows. Assuming A is non-zero, there exists a permutation matrix P such that

$$PAP^T = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{11} is nonsingular, and $s = 1$ or 2 denoting that A_{11} is 1×1 or 2×2, A_{21} is a $(n-s) \times s$ matrix and A_{22} is a $(n-s) \times (n-s)$ matrix. If A_{11} is $s \times s$, we say that an $s \times s$ pivot has been used. The decomposition in outer product form is

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I_s & 0 \\ A_{21}A_{11}^{-1} & I_{n-s} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I_s & A_{11}^{-1}A_{21}^T \\ 0 & I_{n-s} \end{bmatrix}$$

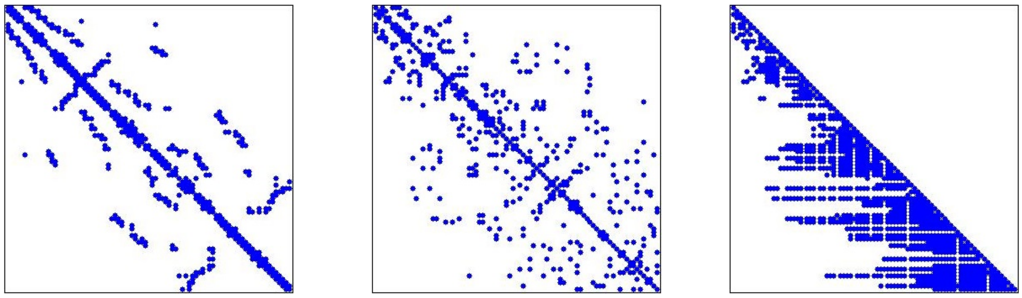
where $S = A_{22} - A_{21}A_{11}^{-1}A_{21}^T$ is the Schur complement. Iteratively applying the reduction to the Schur complement, we obtain the factorization in the form $PAP^T = LDL^T$, where P is a permutation matrix, L is unit lower triangular, and D is block diagonal with each block of order 1 or 2. In LDL^T factorization, choosing the permutation matrix P and pivot size s at each step is called diagonal pivoting.

3.1.3.2 Bunch-Kaufman Pivoting

$a = (1 + \sqrt{17})/8 \approx 0.64039$
$\lambda = \|A(2:n,1)\|_\infty$
if $\lambda = 0$ **then**
| nothing to do on this stage of the elimination
end
$r = \min\{i \geq 2 : |a_{i1}| = \lambda\}$
if $|a_{11}| \geq a\lambda$ **then**
| use a_{11} as a 1×1 pivot $(s = 1,\ P = I)$
else
| $\sigma = \left\| \begin{bmatrix} A(1:r-1,r) \\ A(r+1:n,r) \end{bmatrix} \right\|_\infty$
| **if** $|a_{11}|\sigma \geq a\lambda^2$ **then**
| | use a_{11} as a 1×1 pivot $(s = 1,\ P = I)$
| **else if** $|a_{rr}| \geq a\sigma$ **then**
| | use a_{rr} as a 1×1 pivot $(s = 1,\ P$ swaps rows and columns 1 and $r)$
| **else**
| | use $\begin{bmatrix} a_{11} & a_{r1} \\ a_{r1} & a_{rr} \end{bmatrix}$ as a 2×2 pivot $(s = 2,\ P$ swaps rows and columns 2 and $r)$
| **end**
end

Algorithm 3.1: Bunch-Kaufman Pivoting Algorithm

To describe the pivoting strategy of Bunch and Kaufman [2] it suffices to describe the choice of P and s on the first stage of the factorization. This algorithm 3.1 determines the pivot for the first stage of block LDL^T factorization with the Bunch-Kaufman pivoting strategy applied to a complex symmetric matrix $A \in \mathbb{C}^{n \times n}$. Another possibility to improve the pivoting accuracy is to use symmetric weighted matchings algorithms. These methods identify large entries in A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.



(a) Matrix A (523 non-zero elements)

(b) Matrix PAP^T (523 non-zero elements)

(c) Cholesky factor L (1136 non-zero elements)

FIGURE 3.4: Sparsity pattern of the matrix A resulted from a least squares problem (a). Also the pattern of the permuted matrix PAP^T (b) and the corresponding Cholesky factor L (c) based on the Bunch-Kaufman pivoting algorith.

3.1.4 Structurally Symmetric Matrices

An alternative to working in complex arithmetic is to solve an equivalent real system. Let $A = B + iC$ with both $B,\ C$ real, positive definite $n \times n$ matrices. We consider the symmetric modification

$$\begin{bmatrix} B & -C \\ C & B \end{bmatrix} \begin{bmatrix} x \\ -y \end{bmatrix} = \begin{bmatrix} d \\ e \end{bmatrix} \tag{3.5}$$

The coefficient matrix is a structurally symmetric matrix ($a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0,\ \forall i, j$). For such a matrix an LDL^T factorization exists. The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = LDL^T$. Although LDL^T without pivoting is not always stable for structurally symmetric matrices, the solver uses partial pivoting in the supernodes. Solving (3.5) requires $8n^3/3$ real operations as opposed to $n^3/3$ complex operations to solve the original complex system directly, and it requires the same amount of storage. A complex operation requires between 2 and 8 real operations, so solving the complex system should, in principal, be the more efficient option, but the actual relative costs of real and complex arithmetic will depend on the computing environment.

3.1.5 Iterative Refinement

For all the matrix cases above (3.1.2 - 3.1.4), when the solution vector x is computed, an iterative refinement method [19] is applied in order to improve the accuracy. Due to the presence of rounding errors, the computed solution x may sometimes deviate from the exact solution x^*. Starting with $x_1 = x$, iterative refinement computes a sequence $\{x_1, x_2, x_3, \ldots\}$ which converges to x^* when certain assumptions are met. To describe the iterative refinement algorithm it suffices to describe the m-th iteration:

---

for $m = 1, 2, \ldots$
  Compute the residual
  $r_m = b - Ax_m$
  Solve the system
  $Ad_m = r_m$
  Add the correction
  $x_{m+1} = x_m + d_m$
end

Algorithm 3.2: Iterative Refinement Algorithm

3.2 QMRPACK

QMRPACK is a collection of FORTRAN-77 library routines implementing several of the QMR algorithms and an eigenvalue solver, available from NETLIB. The QMRPACK package currently includes three main QMR algorithms for the solution of square linear systems:

1. The original QMR algorithm, based on the look-ahead three-term Lanczos process.
2. The QMR algorithm based on the look-ahead coupled two-term recurrence Lanczos process.
3. The TFQMR algorithm (Transpose Free QMR)

Even though the main emphasis of the package is on the linear systems solvers, also included is a routine for obtaining approximate eigenvalues from the look-ahead three-term Lanczos algorithm. In addition, the package includes simplified no-look-ahead versions of both the three-term and the coupled two-term QMR algorithms. The codes are available for general non-Hermitian matrices, in single and double precision, real and complex data types. The library support routines include the BLAS routines and routines from LINPACK.

Some of the issues that come up in the design and use of iterative methods solvers are the choice of a convergence criterion, the integration of the preconditioner in the algorithm, and the implementation of the matrix-vector operations. In QMRPACK, the convergence criterion used is the relative residual norm $\|r_n\|_2/\|r_0\|_2$. This choice is hard-coded in the algorithms, and there is no facility for the user to change it without additional coding. The preconditioner is not explicitly incorporated into the linear system solvers, though of course this could be done for all the algorithms in the package. The application of the preconditioner rests with the user, inside the matrix-vector routines, and the codes solve the already preconditioned system. This means that all the quantities generated by the codes belong to the preconditioned system, and not the original system. In particular, the convergence criterion will use the preconditioned residual to determine convergence. For the matrix-vector routines, QMRPACK uses reverse communication. This mechanism consists of having the solver routines return to the caller, with a flag set, when a matrix-vector multiplication is needed, then having the caller perform the operation and call back the solver. The advantage of reverse communication is that it allows complete flexibility over the data structure for the matrix and the implementation of the matrix-vector operations, since all the matrix information is external to the solvers.

The routines in QMRPACK have several control parameters, allowing the user complete control over such aspects of the algorithm as the generation of the auxiliary starting vectors, the choice between the computation of the true residual norm at every step or the use of the residual norm upper bounds that are available in the QMR algorithms, output history, and so forth. All the vector operations in QMRPACK are implemented via calls to BLAS routines, which means that the package will benefit from optimized versions of these routines where available.

In this thesis we use QMRPACK to calculate the solution of a sparse, complex symmetric linear system of equations. More specifically, we use the QMR algorithm based on the look-ahead coupled two-term recurrence Lanczos process and the simplified no-look-ahead version of the same algorithm, both equipped with ILUT and SSOR preconditioners.

3.2.1 Krylov Subspace Methods

Consider the $N \times N$ complex symmetric system of equations

$$Ax = b \tag{3.6}$$

where N is large and A is sparse.

The n-th Krylov subspace of $\mathbb{C}^N$ generated by $c \in \mathbb{C}^N$ and the $N \times N$ matrix A is defined by

$$K_n(c, A) = span\{c, Ac, \ldots, A^{n-1}c\}$$

Let $x_0 \in \mathbb{C}^N$ be an arbitrary initial guess for the linear system (3.6). Let $r_0 = b - Ax_0$ be the corresponding residual vector. An iterative scheme for solving (3.6) is called Krylov subspace method. For any choice of x_0 it produces approximate solutions of the form

$$x_n \in x_0 + K_n(r_0, A), \qquad n = 1, 2, \ldots \tag{3.7}$$

Clearly, the design of a Krylov subspace algorithm consists of two main parts:

1. The construction of suitable basis vectors for the Krylov subspaces $K_n(r_0, A)$ in (3.7)
2. The choice of the actual iterates x_n

The QMR method is an example of a Krylov subspace iteration, where the basis vectors are generated by means of the nonsymmetric Lanczos process, and the iterates are characterized by a quasi-minimal residual property.

3.2.2 The Coupled Two-term Look-ahead Lanczos Process

The QMRPACK uses a unique approach in constructing the Lanczos vectors. The basic idea is to break up the three-term recurrences in the Lanczos process into coupled two-term recurrences, by using (in addition to the Lanczos vectors) a suitable second set of basis vectors for the underlying Krylov subspaces. The QMR based algorithm on this coupled two-term procedure has better numerical properties than the original implementation of QMR based on three-term recurrences. The algorithm generates, in addition to the Lanczos vectors $\{v_j\}_{j=1}^n$ and $\{w_j\}_{j=1}^n$, a second set of basis vectors, $\{p_j\}_{j=1}^n$ and $\{q_j\}_{j=1}^n$, such that, for $n = 1, 2, \ldots$

$$span\{p_1, p_2, \ldots, p_n\} = K_n(v_1, A) \quad \text{and} \quad span\{q_1, q_2, \ldots, q_n\} = K_n(w_1, A^T)$$

The four sets of basis vectors are generated using coupled two-term recurrences of the form

$$V_n = P_n U_n, \quad AP_n = V_{n+1} L_n, \quad W_n = Q_n \Gamma_n^{-1} U_n \Gamma_n, \quad A^T Q_n = W_{n+1} \Gamma_{n+1}^{-1} L_n \Gamma_n$$

Here $P_n = [p_1\ p_2\ \dots\ p_n]$ and $Q_n = [q_1\ q_2\ \dots\ q_n]$, while U_n is an upper triangular matrix and L_n is an upper Hessenberg matrix, given by

$$U_n = \begin{bmatrix} 1 & u_{1,2} & \cdots & u_{1,n} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & u_{n-1,n} \\ 0 & \cdots & 0 & 1 \end{bmatrix} \qquad L_n = \begin{bmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,n} \\ \rho_2 & l_{2,2} & & \vdots \\ 0 & \rho_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & l_{n,n} \\ 0 & \cdots & 0 & \rho_{n+1} \end{bmatrix}$$

and Γ_n is the diagonal matrix defined by

$$\Gamma_n = diag(\gamma_1, \gamma_2, \dots, \gamma_n) \quad \text{where} \quad \gamma_j = \begin{cases} 1, & j = 1 \\ \gamma_{j-1}\rho_j/\xi_j, & j > 1 \end{cases}$$

and $\rho_j,\ \xi_j$ are scale factors used to ensure that v_j and w_j, respectively, obey the scaling $\|v_n\| = \|w_n\| = 1,\ n = 1, 2, \dots$. The matrices L_n and U_n define a factorization of the block tridiagonal Hessenberg matrix H_n generated by the three-term look-ahead Lanczos algorithm

$$H_n = L_n U_n$$

In addition, it is possible to reduce L_n and U_n to block bidiagonal, by constructing the basis vectors p_n and q_n so as to be block A-biorthogonal. Here, all the vectors $v_j,\ w_j,\ p_j,\ q_j$ are constructed using look-ahead techniques. For example we have blocks

$$V^{(j)} = [v_{n_j} \quad v_{n_j+1} \quad \cdots \quad] \quad \text{and} \quad W^{(j)} = [w_{n_j} \quad w_{n_j+1} \quad \cdots \quad]$$
$$P^{(j)} = [p_{m_j} \quad p_{m_{j+1}} \quad \cdots \quad] \quad \text{and} \quad Q^{(j)} = [q_{m_j} \quad q_{m_j+1} \quad \cdots \quad]$$

where $v_{n_j},\ w_{n_j},\ p_{m_j},\ q_{m_j}$ are called regular, the other vectors in the block are called inner and the indices $n_j,\ m_j$ satisfy

$$1 = n_1 < n_2 < \cdots < n_l \leq n < n_{l+1}, \quad l = l(n)$$
$$1 = m_1 < m_2 < \cdots < m_k \leq n < m_{k+1}, \quad k = k(n)$$

and $l,\ k$ are the numbers of look-ahead steps that have been performed during the first n steps of the Lanczos process. The second set of regular vectors satisfy the A-biorthogonality condition

$$q_i^T A p_{m_j} = 0, \qquad \forall i < m_j$$

while the inner vectors satisfy only a relaxed version of this condition. The structure of W_n parallels that of V_n and the structure of Q_n parallels that of P_n. The A-biorthogonality of the $p_j,\ q_j$ sets of basis vectors can be written as

$$Q_n^T A P_n = E_n = diag(E^{(1)}, E^{(2)}, \ldots, E^{(k)}), \quad E^{(j)} = (Q^{(j)})^T A P^{(j)}$$

and the biorthogonality of the $v_j,\ w_j$ can be written as

$$W_n^T V_n = D_n = diag(D^{(1)}, D^{(2)}, \ldots, D^{(l)}), \quad D^{(j)} = (W^{(j)})^T V^{(j)}$$

3.2.3 QMR Algorithms

In the QMR method, the vectors generated by the Lanczos algorithm are used as a basis for the Krylov subspace $K_n(r_0, A)$ in (3.7). The n-th QMR iterate x_n is then defined by

$$x_n = x_0 + V_n z_n$$

where $z_n \in \mathbb{C}^n$ is the unique solution of the least squares problem

$$\|f_{n+1} - \Omega_{n+1} H_n z_n\| = \min_{z \in \mathbb{C}^n} \|f_{n+1} - \Omega_{n+1} H_n z\|$$

Here

$$f_{n+1} = \omega_1 \rho_1 \cdot [1 \quad 0 \quad \cdots \quad 0]^T \in \mathbb{R}^{n+1}$$

with $\rho_1 = \|r_0\|$ and

$$\Omega_{n+1} = diag(\omega_1, \omega_2, \ldots, \omega_{n+1}), \quad \omega_j > 0, \quad j = 1, 2, \ldots, n+1$$

is an arbitrary diagonal weighting matrix. The standard choice for the weights above is $\omega_j = 1, \ \forall j$. Then the residual vector $r_n = b - Ax_n$ satisfies

$$r_n = V_{n+1} \Omega_{n+1}^{-1} (f_{n+1} - \Omega_{n+1} H_n z_n) \tag{3.8}$$

The n-th QMR iterate x_n is characterized by a minimization of the second factor in (3.8). This is just the quasi-minimal residual property. We remark that the QMR iterates x_n can be easily updated from step to step. Due to the block tridiagonal structure of H_n, this update can be implemented with only short recurrences. Also, we note that the quasi-minimal residual property can be used to derive convergence results for the QMR method.

3.2.3.1 QMR Based on Coupled Two-term Lanczos with Look-ahead

We consider the quasi-minimal residual approach and briefly outline how it can be combined with the coupled two-term look-ahead Lanczos algorithm of 3.2.2 to obtain the modified QMR method. Let $x_0 \in \mathbb{C}^N$ be an initial guess for the solution of (3.6), and $r_0 = b - Ax_0$ the corresponding initial residual, where $\rho_1 = \|r_0\|$. Choosing $v_1 = r_0/\rho_1$ as the starting right Lanczos vector and w_1 with $\|w_1\| = 1$ as an arbitrary starting left Lanczos vector, one obtains the four basis sets $V_n, \ W_n, \ P_n, \ Q_n$ of which the ones of interest are V_n and P_n, related by

$$V_n = P_n U_n, \quad AP_n = V_{n+1} L_n$$

Once the basis vectors are constructed, the n-th QMR iterate is selected from the shifted Krylov subspace $x_0 + K_n(r_0, A)$ as

$$x_n = x_0 + P_n y_n \tag{3.9}$$

where $y_n \in \mathbb{C}^n$ is defined by the quasi-minimal residual condition

$$\|f_{n+1} - L_n y_n\| = \min_{z \in \mathbb{C}^n} \|f_{n+1} - L_n y\| \tag{3.10}$$

This is an $(n+1) \times n$ least-squares problem, where

$$f_{n+1} = \rho_1 \cdot [1 \quad 0 \quad \cdots \quad 0]^T \ \in \mathbb{R}^{n+1}$$

Note that, by setting

$$z_n = (U_n)^{-1} y_n$$

and inserting in (3.10), one obtains the equivalent least-squares problem

$$\|f_{n+1} - L_n U_n z_n\| = \min_{z \in \mathbb{C}^n} \|f_{n+1} - L_n U_n z\|$$

which is exactly the least-squares problem solved by the QMR algorithm based on the three-term Lanczos process. Thus, the QMR iterates (3.9) are, in exact arithmetic, identical to the iterates of the original QMR algorithm. However in finite precision arithmetic, the coupled QMR algorithm is more robust than the three-term recurrence version. For a full discuss of the QMR method based on coupled two-term recurrences we refer the reader to [4].

Chapter 4

Preconditioners

The rate of convergence of a Krylov subspace method for a linear system $Ax = b$, depends on the condition number of the matrix A. Therefore, if we have a matrix M which is a crude approximation to A, $M^{-1}A$ is closer to the identity than A is and should have a smaller condition number. It would be expected that a Krylov subspace method would converge faster for the *preconditioned* system

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

For example, choosing M to be the diagonal part of A can be a possible choice. Such a matrix M is called a *preconditioner* or, more precisely, a *left preconditioner*. In the case of a *right preconditioner*, one solves

$$AM^{-1}u = b \quad \text{where} \quad u = Mx$$

Preconditioning is often applied from both sides

$$M_1^{-1}AM_2^{-1}u = M_1^{-1}b \quad \text{where} \quad u = M_2x$$

where M_1 and M_2 are the preconditioning matrices. Note that we now solve a linear system in u, not in x. As soon as u is found, x can be computed as $x = M_2^{-1}u$. This *two-sided preconditioning* is necessary when the matrix A is symmetric. Left preconditioning destroys the symmetry whereas applying *two-sided preconditioning* we can get a symmetric *preconditioned* matrix.

4.1 ILUT

Incomplete LU factorizations, combined with a good Krylov subspace projection process, are often regarded as the best *general purpose* iterative solvers. In general, the reliability of such methods for solving problems from various origins depends much more on the quality of the preconditioner than on the iterative method. For a full discussion on preconditioning techniques we refer the reader to [3].

A common way to define a preconditioner is through an ILU factorization obtained from an approximate Gaussian elimination process. When Gaussian elimination is applied to a sparse matrix A, a large number of non-zero elements in the factors, L and U, may appear in locations occupied by zero elements in A. These fill-ins often have small values and, therefore, they can be dropped to obtain a sparse approximate LU factorization, referred to as an incomplete LU (ILU) factorization. The simplest of these procedures, $ILU(0)$, is obtained by performing the standard LU factorization of A and dropping all fill-in elements generated during the process. Thus, the factors, L and U, have the same pattern as the lower and upper triangular parts of A respectively.

In the early work on ILU preconditioners, it was understood that ILU(0) could be ineffective and that more accurate factorizations would be needed. This row-wise algorithm is based on the so-called i, k, j Gaussian elimination process, whereby the i-th step computes the i-th rows of L and U, Algorithm 4.1.

```
for i = 1 : n
    w = A_{i,1:n}
    for k = 1 : i − 1
        w_k = w_k / u_{k,k}
        w_{k+1:n} = w_{k+1:n} − w_k · U_{k,k+1:n}
    end
    for j = 1 : i − 1
        l_{i,j} = w_j  (l_{i,i} = 1)
    end
    for j = 1 : n
        u_{i,j} = w_j
    end
end
```

Algorithm 4.1: i, k, j-ordered Gaussian Elimination

Here $a_{i,k}$, $l_{i,k}$ and $u_{i,k}$ represent the scalar entries at the i-th row and k-th column of the matrices A, L and U, respectively. $A_{i,1:n}$ denotes the complete i-th row of A (transposed as a column vector), while $A_{1:n,j}$ denotes the j-th column of A, $\boldsymbol{w}_{k+1:n}$ denotes the last $n-k$ entries in the vector $\boldsymbol{w}$, $U_{k,k+1:n}$ denotes the last $n-k$ entries in the k-th row of U (transposed as a column vector), $L_{i,1:i-1}$ denotes the first $i-1$ entries in the i-th row of L (transposed as a column vector), and so forth. Of note in Algorithm 4.1 is that at the i-th step, the i-th row of A is modified by previously computed rows of U, while the later rows of A and U are not accessed. The incomplete version of this algorithm is based on exploiting sparsity in the elimination and dropping small values according to a certain *dropping rule.*

---

for $i = 1 : n$
    $\boldsymbol{w} = A_{i,1:n}$
    for $k = 1 : i - 1$
        if $w_k \neq 0$ **then**
            $w_k = w_k / u_{k,k}$
            Apply first dropping rule to w_k
        end
        if w_k *in not dropped* **then**
            $\boldsymbol{w}_{k+1:n} = \boldsymbol{w}_{k+1:n} - w_k \cdot U_{k,k+1:n}$
        end
    end
    for $j = 1 : i - 1$
        $l_{i,j} = w_j$ $(l_{i,i} = 1)$
    end
    Apply second dropping rule to $L_{i,1:i-1}$
    for $j = 1 : n$
        $u_{i,j} = w_j$
    end
    Apply second dropping rule to $U_{i,i+1:n}$
end

Algorithm 4.2: *ILUT*

The dropping strategy uses two parameters. The first parameter is a drop tolerance τ, which is used mainly to avoid doing an elimination if the pivot w_k is too small. The second parameter is an integer p, which controls the number of entries that are kept in the i-th rows of L and U. Details can be found in [14] and [10]. The general structure of the algorithm is given as Algorithm 4.2.

The Algorithm 4.2 is row-based for column-oriented programming paradigms (when using *CSC* formatting), however, a column-based approach which is used in QMRPACK is more efficient. Furthermore, the triangular solves involving the L and U factors can be efficiently computed using a column-oriented data structure. For the column version of *ILUT*, at a given step j, the initial j-th column of A, a_j , is transformed by zeroing out entries above the diagonal element. As in the row version, operations of the form $\boldsymbol{w} = \boldsymbol{w} - w_k l_k$ are performed to eliminate entries of $\boldsymbol{w}$ from top to bottom, until all entries strictly above the diagonal are zeroed out. In the *ILU* case, only a few of these eliminations are performed.

4.2 SSOR

The $SSOR$ preconditioner can be derived from the coefficient matrix without any work. If the original, symmetric, matrix is decomposed as

$$A = D + L + L^T$$

where D is the diagonal, L is the lower part and L^T is the upper triangular part of A, the $SSOR$ matrix is defined as

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

or, parameterized by ω

$$M(\omega) = \frac{1}{2-\omega}(\frac{1}{\omega}D + L)(\frac{1}{\omega}D)^{-1}(\frac{1}{\omega}D + L)^T$$

The optimal value of the parameter ω will reduce the number of iterations. In practice, however, the spectral information needed to calculate the optimal ω is prohibitively expensive to compute. The $SSOR$ matrix is given in factored form, so this preconditioner shares many properties of other factorization-based methods. For instance, its suitability for vector processors or parallel architectures depends strongly on the ordering of the variables. For more details about the $SSOR$ preconditioner we refer the reader to [15] and [20].

Chapter 5

Numerical Experiments

The experiments have been performed on a quad-core Intel® Core™ i7-3630QM (6M Cache, up to 3.40 GHz) processor. The tolerance at the QMRPACK is defined 1.0e-12. The purpose of this section is to compare the runtime between the two packages. Bellow we briefly describe the test matrices.

The qc324$^{5.1}$ and qc2534$^{5.3}$ matrices are formed by a quantum chemistry model of H_2^+ in an electromagnetic field.

The dwg961b$^{5.2}$ matrix arises from an electrical engineering model. The matrix is formed by an edge element method used to solve the waveguide problem of conductors with finite conductivity and cross section in a lossy dielectric medium.

The dielFilterV3clx$^{5.4}$ matrix came from analysis of a microwave combline filter with second order vector finite elements. For further information about the matrices described above we refer the reader to [1].

The femHlmtz$^{5.5}$ matrix is formed by a standard Galerkin finite element method for the Helmholtz equation. For more information about the femHlmtz matrix we refer the reader to [11].

The femSch$^{5.6}$ matrices are formed by an adaptive discontinuous Galerkin finite element method for the non-linear (cubic) Schrödinger equation with linear basis functions, $\gamma = 12$ and time discretization step $dt = 1.0e-05$.

The femSch$(\gamma, d)^{5.7}$ matrices are formed by an adaptive discontinuous Galerkin finite element method for the non-linear (cubic) Schrödinger equation with linear ($d = 1$) and quadratic ($d = 2$) basis functions , variable $\gamma = 18, 30, 60$ and time discretization step $dt = 1.0e-04$. For more information about the femSch matrices we refer the reader to [13].

5.1 Matrix qc324

qc324		
Type		Complex symmetric indefinite
Size		324×324
Non-zero elements		26730
Longest row/column	Index	82
	Non-zero elements	83
Shortest row/column	Index	1
	Non-zero elements	82
Average non-zeros per row/column		82
Diagonal dominance		No

Table 5.1: qc324 Matrix statistics.

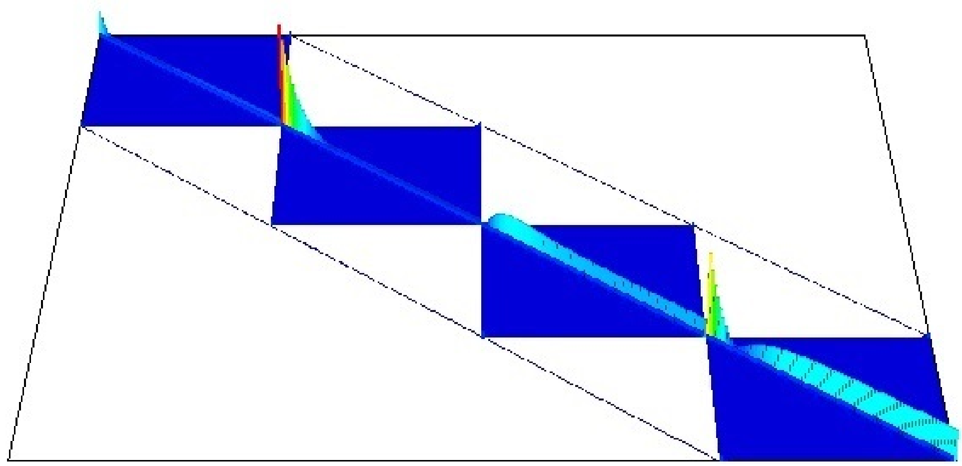


Figure 5.1: 3-D Value-colored sparsity pattern of qc324 matrix.

qc324 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.006981s	0.009740s	0.000835s	*0.017556s*
2	0.006724s	0.005877s	0.000592s	*0.013193s*
4	0.006504s	0.005029s	0.000525s	*0.012058s*

Table 5.2: Pardiso run results of qc324 matrix.

qc324 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		1925	*1.504s*
ILUT	$p = 5,\ \tau = 1.0e-06$	8	*0.116s*
	$p = 5,\ \tau = 1.0e-08$	8	*0.118s*
	$p = 10,\ \tau = 1.0e-06$	8	*0.121s*
	$p = 10,\ \tau = 1.0e-08$	8	*0.123s*
SSOR	$\omega = 1.3$	-	-
	$\omega = 1.6$	-	-
	$\omega = 1.9$	-	-

Table 5.3: QMR(CPL) run results of qc324 matrix.

qc324 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		1866	*1.431s*
ILUT	$p = 5,\ \tau = 1.0e-06$	8	*0.122s*
	$p = 5,\ \tau = 1.0e-08$	8	*0.119s*
	$p = 10,\ \tau = 1.0e-06$	8	*0.115s*
	$p = 10,\ \tau = 1.0e-08$	8	*0.123s*
SSOR	$\omega = 1.3$	-	-
	$\omega = 1.6$	-	-
	$\omega = 1.9$	-	-

Table 5.4: QMR(CPX) run results of qc324 matrix.

qc324 Pardiso vs. QMR		
Pardiso best total time		*0.012s*
QMR best total time	CPL	*0.116s*
	CPX	*0.115s*
Pardiso worst total time		*0.018s*
QMR worst total time	CPL	*1.504s*
	CPX	*1.431s*
QMR/Pardiso		*9.5834*

Table 5.5: Pardiso vs. QMR best/worst run results of qc324 matrix.

5.2 Matrix dwg961b

dwg961b		
Type		Complex symmetric indefinite
Size		961 × 961
Non-zero elements		10591
Longest row/column	Index	723
	Non-zero elements	19
Shortest row/column	Index	1
	Non-zero elements	6
Average non-zeros per row/column		11
Diagonal dominance		No

Table 5.6: dwg961b Matrix statistics.

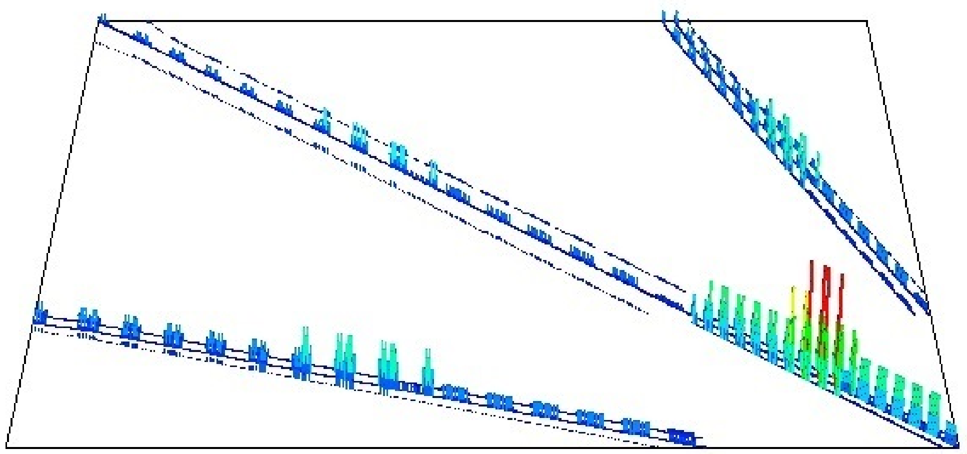


Figure 5.2: 3-D Value-colored sparsity pattern of dwg961b matrix.

dwg961b Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.010363s	0.003130s	0.000851s	*0.014344s*
2	0.010175s	0.001832s	0.000582s	*0.012589s*
4	0.010042s	0.001343s	0.000529s	*0.011914s*

Table 5.7: Pardiso run results of dwg961b matrix.

dwg961b QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		84653	*37.549s*
ILUT	$p = 5,\ \tau = 1.0e-06$	413	*0.510s*
	$p = 5,\ \tau = 1.0e-08$	418	*0.505s*
	$p = 10,\ \tau = 1.0e-06$	62	*0.208s*
	$p = 10,\ \tau = 1.0e-08$	61	*0.209s*
SSOR	$\omega = 1.3$	1910	*1.127s*
	$\omega = 1.6$	2770	*1.698s*
	$\omega = 1.9$	5241	*3.145s*

Table 5.8: QMR(CPL) run results of dwg961b matrix.

dwg961b QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		82228	*32.340s*
ILUT	$p = 5,\ \tau = 1.0e-06$	401	*0.475s*
	$p = 5,\ \tau = 1.0e-08$	429	*0.459s*
	$p = 10,\ \tau = 1.0e-06$	61	*0.184s*
	$p = 10,\ \tau = 1.0e-08$	62	*0.203s*
SSOR	$\omega = 1.3$	1840	*1.068s*
	$\omega = 1.6$	2739	*1.547s*
	$\omega = 1.9$	5432	*2.976s*

Table 5.9: QMR(CPX) run results of dwg961b matrix.

dwg961b Pardiso vs. QMR		
Pardiso best total time		*0.012s*
QMR best total time	CPL	*0.208s*
	CPX	*0.184s*
Pardiso worst total time		*0.014s*
QMR worst total time	CPL	*37.549s*
	CPX	*32.340s*
QMR/Pardiso		*15.3334*

Table 5.10: Pardiso vs. QMR best/worst run results of dwg961b matrix.

5.3 Matrix qc2534

qc2534		
Type		Complex symmetric indefinite
Size		2534 × 2534
Non-zero elements		463360
Longest row/column	Index	182
	Non-zero elements	183
Shortest row/column	Index	1
	Non-zero elements	182
Average non-zeros per row/column		180
Diagonal dominance		No

Table 5.11: qc2534 Matrix statistics.

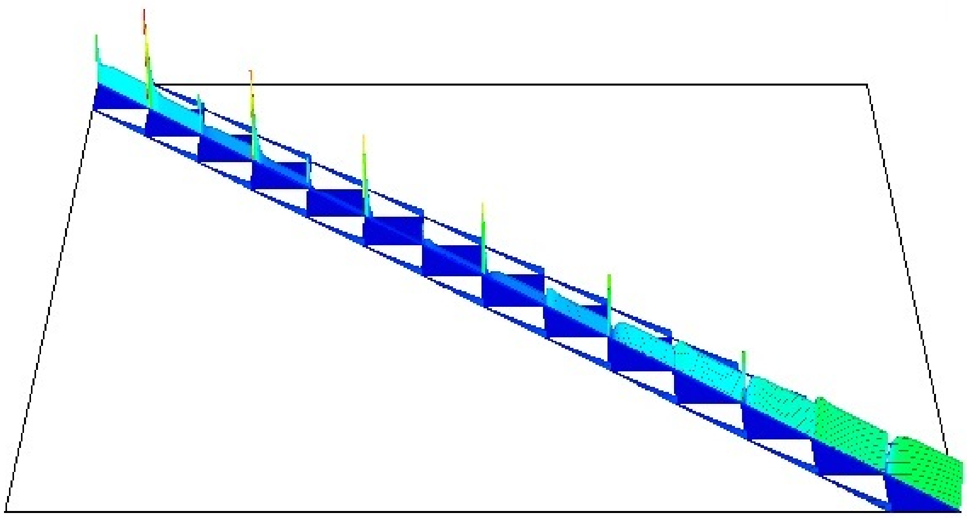


Figure 5.3: 3-D Value-colored sparsity pattern of qc2534 matrix.

qc2534 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.059234s	0.236625s	0.008383s	*0.304242s*
2	0.057660s	0.127300s	0.004846s	*0.189806s*
4	0.061858s	0.077615s	0.005032s	*0.144505s*

Table 5.12: Pardiso run results of qc2534 matrix.

qc2534 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		-	-
ILUT	$p = 5,\ \tau = 1.0e - 06$	85	*3.095s*
	$p = 5,\ \tau = 1.0e - 08$	85	*3.063s*
	$p = 10,\ \tau = 1.0e - 06$	70	*2.769s*
	$p = 10,\ \tau = 1.0e - 08$	70	*2.778s*
SSOR	$\omega = 1.3$	-	-
	$\omega = 1.6$	-	-
	$\omega = 1.9$	-	-

Table 5.13: QMR(CPL) run results of qc2534 matrix.

qc2534 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		-	-
ILUT	$p = 5,\ \tau = 1.0e - 06$	86	*3.067s*
	$p = 5,\ \tau = 1.0e - 08$	86	*3.049s*
	$p = 10,\ \tau = 1.0e - 06$	71	*2.724s*
	$p = 10,\ \tau = 1.0e - 08$	71	*2.790s*
SSOR	$\omega = 1.3$	-	-
	$\omega = 1.6$	-	-
	$\omega = 1.9$	-	-

Table 5.14: QMR(CPX) run results of qc2534 matrix.

qc2534 Pardiso vs. QMR		
Pardiso best total time		*0.145s*
QMR best total time	CPL	*2.769s*
	CPX	*2.724s*
Pardiso worst total time		*0.304s*
QMR worst total time	CPL	*3.095s*
	CPX	*3.067s*
QMR/Pardiso		*18.7862*

Table 5.15: Pardiso vs. QMR best/worst run results of qc2534 matrix.

5.4 Matrix dielFilterV3clx

dielFilterV3clx	
Type	Complex symmetric indefinite
Size	420408 × 420408
Non-zero elements	32886208
Average non-zeros per row/column	78

TABLE 5.16: dielFilterV3clx Matrix statistics.

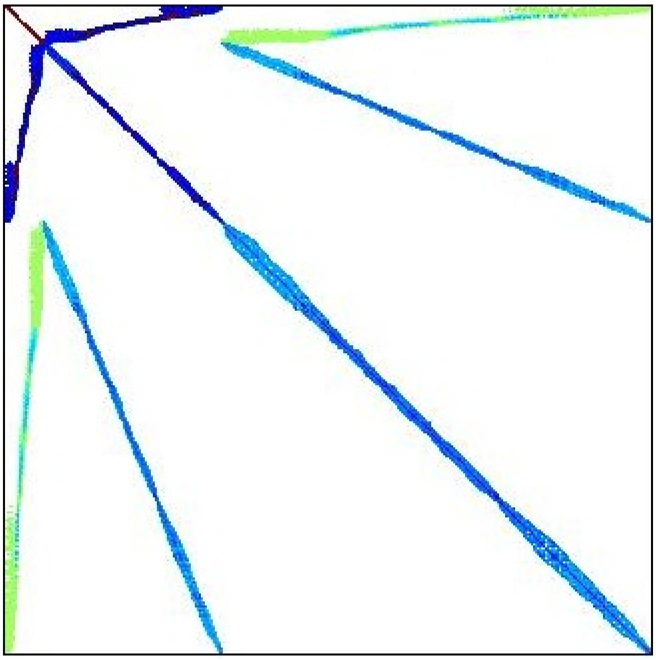


FIGURE 5.4: Value-colored sparsity pattern of dielFilterV3clx matrix.

dielFilterV3clx Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	3.754754s	2m 47.844087s	1.881181s	*2m 53.480022s*
2	3.717047s	1m 28.252591s	0.976366s	*1m 32.946004s*
4	3.785908s	1m 16.750870s	0.789007s	*1m 21.325785s*

TABLE 5.17: Pardiso run results of dielFilterV3clx matrix.

dielFilterV3clx QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
SSOR	$\omega = 1.3$	4880	*70m 44.052s*

TABLE 5.18: QMR(CPL) run results of dielFilterV3clx matrix.

dielFilterV3clx QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
SSOR	$\omega = 1.3$	4924	*70m 8.248s*

TABLE 5.19: QMR(CPX) run results of dielFilterV3clx matrix.

dielFilterV3clx Pardiso vs. QMR		
Pardiso best total time		*1m 21.326s*
QMR best total time	CPL	*70m 44.052s*
	CPX	*70m 8.248s*
QMR/Pardiso		*51.7454*

TABLE 5.20: Pardiso vs. QMR best/worst run results of dielFilterV3clx matrix.

5.5 Matrix femHlmtz

femHlmtz	
Type	Complex symmetric
Size	4681×4681
Non-zero elements	76741
Average non-zeros per row/column	16

Table 5.21: femHlmtz Matrix statistics.

femHlmtz Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.038595s	0.050524s	0.004073s	*0.093192s*
2	0.037591s	0.024672s	0.002338s	*0.064601s*
4	0.039323s	0.021387s	0.002025s	*0.062735s*

Table 5.22: Pardiso run results of femHlmtz matrix.

femHlmtz QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		956	*2.712s*
ILUT	$p = 5,\ \tau = 1.0e-06$	131	*1.568s*
	$p = 5,\ \tau = 1.0e-08$	131	*1.559s*
	$p = 10,\ \tau = 1.0e-06$	111	*1.734s*
	$p = 10,\ \tau = 1.0e-08$	112	*1.740s*
SSOR	$\omega = 1.3$	236	*0.975s*
	$\omega = 1.6$	182	*0.759s*
	$\omega = 1.9$	211	*0.841s*

Table 5.23: QMR(CPL) run results of femHlmtz matrix.

femHlmtz QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		956	*2.524s*
ILUT	$p = 5,\ \tau = 1.0e-06$	131	*1.505s*
	$p = 5,\ \tau = 1.0e-08$	131	*1.519s*
	$p = 10,\ \tau = 1.0e-06$	142	*1.908s*
	$p = 10,\ \tau = 1.0e-08$	110	*1.704s*
SSOR	$\omega = 1.3$	231	*0.860s*
	$\omega = 1.6$	182	*0.714s*
	$\omega = 1.9$	207	*0.772s*

Table 5.24: QMR(CPX) run results of femHlmtz matrix.

femHlmtz Pardiso vs. QMR		
Pardiso best total time		*0.063s*
QMR best total time	CPL	*0.759s*
	CPX	*0.714s*
Pardiso worst total time		*0.093s*
QMR worst total time	CPL	*2.712s*
	CPX	*2.524s*
QMR/Pardiso		*11.3334*

TABLE 5.25: Pardiso vs. QMR best/worst run results of femHlmtz matrix.

5.6 Matrices femSch

femSch2	
Type	Complex symmetric
Size	192×192
Non-zero elements	2160

Table 5.26: femSch2 Matrix statistics.

femSch2 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.001016s	0.000489s	0.000204s	*0.001709s*
2	0.001115s	0.000407s	0.000182s	*0.001704s*
4	0.001141s	0.000354s	0.000190s	*0.001685s*

Table 5.27: Pardiso run results of femSch2 matrix.

femSch2 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		10	*0.030s*
ILUT	$p = 5,\ \tau = 1.0e-06$	3	*0.024s*
	$p = 5,\ \tau = 1.0e-08$	3	*0.023s*
	$p = 10,\ \tau = 1.0e-06$	2	*0.029s*
	$p = 10,\ \tau = 1.0e-08$	2	*0.023s*
SSOR	$\omega = 1.3$	11	*0.021s*
	$\omega = 1.6$	14	*0.022s*
	$\omega = 1.9$	17	*0.023s*

Table 5.28: QMR(CPL) run results of femSch2 matrix.

femSch2 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		10	*0.014s*
ILUT	$p = 5,\ \tau = 1.0e-06$	3	*0.025s*
	$p = 5,\ \tau = 1.0e-08$	3	*0.018s*
	$p = 10,\ \tau = 1.0e-06$	2	*0.019s*
	$p = 10,\ \tau = 1.0e-08$	2	*0.019s*
SSOR	$\omega = 1.3$	11	*0.015s*
	$\omega = 1.6$	14	*0.016s*
	$\omega = 1.9$	17	*0.017s*

Table 5.29: QMR(CPX) run results of femSch2 matrix.

femSch2 Pardiso vs. QMR		
Pardiso best total time		*0.002s*
QMR best total time	CPL	*0.021s*
	CPX	*0.015s*
Pardiso worst total time		*0.002s*
QMR worst total time	CPL	*0.030s*
	CPX	*0.025s*
QMR/Pardiso		*7.5000*

TABLE 5.30: Pardiso vs. QMR best/worst run results of femSch2 matrix.

femSch3	
Type	Complex symmetric
Size	768×768
Non-zero elements	8928

TABLE 5.31: femSch3 Matrix statistics.

femSch3 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.002826s	0.002394s	0.000695s	*0.005915s*
2	0.002755s	0.001698s	0.000515s	*0.004968s*
4	0.002910s	0.001500s	0.000510s	*0.004920s*

TABLE 5.32: Pardiso run results of femSch3 matrix.

femSch3 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		15	*0.043s*
ILUT	$p = 5,\ \tau = 1.0e-06$	4	*0.067s*
	$p = 5,\ \tau = 1.0e-08$	4	*0.070s*
	$p = 10,\ \tau = 1.0e-06$	3	*0.071s*
	$p = 10,\ \tau = 1.0e-08$	3	*0.076s*
SSOR	$\omega = 1.3$	14	*0.047s*
	$\omega = 1.6$	18	*0.057s*
	$\omega = 1.9$	21	*0.055s*

TABLE 5.33: QMR(CPL) run results of femSch3 matrix.

femSch3 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		15	*0.040s*
ILUT	$p = 5,\ \tau = 1.0e-06$	4	*0.065s*
	$p = 5,\ \tau = 1.0e-08$	4	*0.063s*
	$p = 10,\ \tau = 1.0e-06$	3	*0.071s*
	$p = 10,\ \tau = 1.0e-08$	3	*0.072s*
SSOR	$\omega = 1.3$	14	*0.042s*
	$\omega = 1.6$	18	*0.044s*
	$\omega = 1.9$	21	*0.046s*

Table 5.34: QMR(CPX) run results of femSch3 matrix.

femSch3 Pardiso vs. QMR		
Pardiso best total time		*0.005s*
QMR best total time	CPL	*0.043s*
	CPX	*0.040s*
Pardiso worst total time		*0.006s*
QMR worst total time	CPL	*0.076s*
	CPX	*0.072s*
QMR/Pardiso		*8.0000*

Table 5.35: Pardiso vs. QMR best/worst run results of femSch3 matrix.

femSch4	
Type	Complex symmetric
Size	3072×3072
Non-zero elements	36288

Table 5.36: femSch4 Matrix statistics.

femSch4 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.011393s	0.015956s	0.002633s	*0.029982s*
2	0.011555s	0.008870s	0.001794s	*0.022219s*
4	0.011397s	0.006911s	0.001454s	*0.019762s*

Table 5.37: Pardiso run results of femSch4 matrix.

femSch4 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		26	*0.144s*
ILUT	$p = 5,\ \tau = 1.0e-06$	6	*0.340s*
	$p = 5,\ \tau = 1.0e-08$	6	*0.317s*
	$p = 10,\ \tau = 1.0e-06$	4	*0.381s*
	$p = 10,\ \tau = 1.0e-08$	4	*0.388s*
SSOR	$\omega = 1.3$	18	*0.148s*
	$\omega = 1.6$	24	*0.163s*
	$\omega = 1.9$	33	*0.185s*

Table 5.38: QMR(CPL) run results of femSch4 matrix.

femSch4 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		26	*0.134s*
ILUT	$p = 5,\ \tau = 1.0e-06$	6	*0.347s*
	$p = 5,\ \tau = 1.0e-08$	6	*0.351s*
	$p = 10,\ \tau = 1.0e-06$	4	*0.382s*
	$p = 10,\ \tau = 1.0e-08$	4	*0.383s*
SSOR	$\omega = 1.3$	18	*0.141s*
	$\omega = 1.6$	24	*0.154s*
	$\omega = 1.9$	33	*0.178s*

Table 5.39: QMR(CPX) run results of femSch4 matrix.

femSch4 Pardiso vs. QMR		
Pardiso best total time		*0.020s*
QMR best total time	CPL	*0.144s*
	CPX	*0.134s*
Pardiso worst total time		*0.030s*
QMR worst total time	CPL	*0.388s*
	CPX	*0.383s*
QMR/Pardiso		*6.7000*

Table 5.40: Pardiso vs. QMR best/worst run results of femSch4 matrix.

femSch5	
Type	Complex symmetric
Size	12288 × 12288
Non-zero elements	146304

TABLE 5.41: femSch5 Matrix statistics.

femSch5 Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.034950s	0.070085s	0.003807s	*0.108842s*
2	0.035627s	0.032648s	0.004615s	*0.072890s*
4	0.031429s	0.022870s	0.005173s	*0.059472s*

TABLE 5.42: Pardiso run results of femSch5 matrix.

femSch5 QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		48	*0.536s*
ILUT	$p = 5,\ \tau = 1.0e-06$	9	*3.051s*
	$p = 5,\ \tau = 1.0e-08$	9	*3.068s*
	$p = 10,\ \tau = 1.0e-06$	7	*3.450s*
	$p = 10,\ \tau = 1.0e-08$	7	*3.343s*
SSOR	$\omega = 1.3$	24	*0.464s*
	$\omega = 1.6$	35	*0.530s*
	$\omega = 1.9$	52	*0.638s*

TABLE 5.43: QMR(CPL) run results of femSch5 matrix.

femSch5 QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		48	*0.509s*
ILUT	$p = 5,\ \tau = 1.0e-06$	9	*3.051s*
	$p = 5,\ \tau = 1.0e-08$	9	*3.056s*
	$p = 10,\ \tau = 1.0e-06$	7	*3.435s*
	$p = 10,\ \tau = 1.0e-08$	7	*3.449s*
SSOR	$\omega = 1.3$	24	*0.451s*
	$\omega = 1.6$	35	*0.512s*
	$\omega = 1.9$	52	*0.603s*

TABLE 5.44: QMR(CPX) run results of femSch5 matrix.

femSch5 Pardiso vs. QMR		
Pardiso best total time		*0.060s*
QMR best total time	CPL	*0.464s*
	CPX	*0.451s*
Pardiso worst total time		*0.109s*
QMR worst total time	CPL	*3.450s*
	CPX	*3.449s*
QMR/Pardiso		*7.5167*

TABLE 5.45: Pardiso vs. QMR best/worst run results of femSch5 matrix.

5.7 Matrices femSch(γ, d)

femSch($\gamma, 1$)	
Type	Complex symmetric
Size	3072 × 3072
Non-zero elements	36288

TABLE 5.46: femSch($\gamma, 1$) Matrix statistics.

femSch($18, 1$) **Pardiso results**				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.010691s	0.015418s	0.002626s	*0.028735s*
2	0.010624s	0.008984s	0.001884s	*0.021492s*
4	0.011302s	0.006897s	0.001401s	*0.019600s*

TABLE 5.47: Pardiso run results of femSch($18, 1$) matrix.

femSch($18, 1$) **QMR(CPL) results**			
Preconditioner		**Iterations**	***Total time***
-		84	*0.230s*
ILUT	$p = 5,\ \tau = 1.0e-06$	12	*0.354s*
	$p = 5,\ \tau = 1.0e-08$	12	*0.374s*
	$p = 10,\ \tau = 1.0e-06$	9	*0.376s*
	$p = 10,\ \tau = 1.0e-08$	9	*0.394s*
SSOR	$\omega = 1.3$	35	*0.181s*
	$\omega = 1.6$	47	*0.198s*
	$\omega = 1.9$	72	*0.240s*

TABLE 5.48: QMR(CPL) run results of femSch($18, 1$) matrix.

femSch($18, 1$) **QMR(CPX) results**			
Preconditioner		**Iterations**	***Total time***
-		84	*0.223s*
ILUT	$p = 5,\ \tau = 1.0e-06$	12	*0.367s*
	$p = 5,\ \tau = 1.0e-08$	12	*0.363s*
	$p = 10,\ \tau = 1.0e-06$	9	*0.389s*
	$p = 10,\ \tau = 1.0e-08$	9	*0.387s*
SSOR	$\omega = 1.3$	35	*0.180s*
	$\omega = 1.6$	47	*0.200s*
	$\omega = 1.9$	72	*0.250s*

TABLE 5.49: QMR(CPX) run results of femSch($18, 1$) matrix.

femSch$(18, 1)$ Pardiso vs. QMR		
Pardiso best total time		*0.020s*
QMR best total time	CPL	*0.181s*
	CPX	*0.180s*
Pardiso worst total time		*0.029s*
QMR worst total time	CPL	*0.394s*
	CPX	*0.389s*
QMR/Pardiso		*9.0000*

Table 5.50: Pardiso vs. QMR best/worst run results of femSch$(18, 1)$ matrix.

femSch$(30, 1)$ Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.010818s	0.015240s	0.002535s	*0.028593s*
2	0.010833s	0.009573s	0.001415s	*0.021821s*
4	0.010247s	0.005773s	0.001200s	*0.017220s*

Table 5.51: Pardiso run results of femSch$(30, 1)$ matrix.

femSch$(30, 1)$ QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		106	*0.266s*
ILUT	$p = 5,\ \tau = 1.0e-06$	13	*0.362s*
	$p = 5,\ \tau = 1.0e-08$	13	*0.356s*
	$p = 10,\ \tau = 1.0e-06$	10	*0.407s*
	$p = 10,\ \tau = 1.0e-08$	10	*0.390s*
SSOR	$\omega = 1.3$	44	*0.204s*
	$\omega = 1.6$	58	*0.236s*
	$\omega = 1.9$	89	*0.276s*

Table 5.52: QMR(CPL) run results of femSch$(30, 1)$ matrix.

femSch$(30, 1)$ QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		106	*0.245s*
ILUT	$p = 5,\ \tau = 1.0e-06$	13	*0.355s*
	$p = 5,\ \tau = 1.0e-08$	13	*0.370s*
	$p = 10,\ \tau = 1.0e-06$	10	*0.396s*
	$p = 10,\ \tau = 1.0e-08$	10	*0.404s*
SSOR	$\omega = 1.3$	44	*0.193s*
	$\omega = 1.6$	58	*0.226s*
	$\omega = 1.9$	89	*0.277s*

Table 5.53: QMR(CPX) run results of femSch$(30, 1)$ matrix.

femSch$(30, 1)$ Pardiso vs. QMR		
Pardiso best total time		*0.017s*
QMR best total time	CPL	*0.204s*
	CPX	*0.193s*
Pardiso worst total time		*0.029s*
QMR worst total time	CPL	*0.407s*
	CPX	*0.404s*
QMR/Pardiso		*11.3529*

TABLE 5.54: Pardiso vs. QMR best/worst run results of femSch$(30, 1)$ matrix.

femSch$(60, 1)$ Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.010955s	0.015910s	0.002264s	*0.029129s*
2	0.011376s	0.009786s	0.001892s	*0.023054s*
4	0.010822s	0.006499s	0.001584s	*0.018905s*

TABLE 5.55: Pardiso run results of femSch$(60, 1)$ matrix.

femSch$(60, 1)$ QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		143	*0.310s*
ILUT	$p = 5,\ \tau = 1.0e - 06$	16	*0.379s*
	$p = 5,\ \tau = 1.0e - 08$	16	*0.360s*
	$p = 10,\ \tau = 1.0e - 06$	11	*0.384s*
	$p = 10,\ \tau = 1.0e - 08$	11	*0.400s*
SSOR	$\omega = 1.3$	56	*0.235s*
	$\omega = 1.6$	77	*0.264s*
	$\omega = 1.9$	115	*0.336s*

TABLE 5.56: QMR(CPL) run results of femSch$(60, 1)$ matrix.

femSch$(60, 1)$ QMR(CPX) results			
Preconditioner		**Iterations**	***Total time***
-		143	*0.303s*
ILUT	$p = 5,\ \tau = 1.0e - 06$	16	*0.378s*
	$p = 5,\ \tau = 1.0e - 08$	16	*0.368s*
	$p = 10,\ \tau = 1.0e - 06$	11	*0.388s*
	$p = 10,\ \tau = 1.0e - 08$	11	*0.374s*
SSOR	$\omega = 1.3$	44	*0.224s*
	$\omega = 1.6$	77	*0.256s*
	$\omega = 1.9$	115	*0.319s*

TABLE 5.57: QMR(CPX) run results of femSch$(60, 1)$ matrix.

femSch$(60, 1)$ Pardiso vs. QMR		
Pardiso best total time		*0.019s*
QMR best total time	CPL	*0.235s*
	CPX	*0.224s*
Pardiso worst total time		*0.029s*
QMR worst total time	CPL	*0.400s*
	CPX	*0.388s*
QMR/Pardiso		*11.7895*

Table 5.58: Pardiso vs. QMR best/worst run results of femSch$(60, 1)$ matrix.

femSch$(\gamma, 2)$	
Type	Complex symmetric
Size	9216×9216
Non-zero elements	142084

Table 5.59: femSch$(\gamma, 2)$ Matrix statistics.

femSch$(30, 2)$ **Pardiso results**				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.018581s	0.060454s	0.006072s	*0.085107s*
2	0.018466s	0.029492s	0.004009s	*0.051967s*
4	0.018678s	0.018928s	0.002984s	*0.040590s*

Table 5.60: Pardiso run results of femSch$(30, 2)$ matrix.

femSch$(30, 2)$ **QMR(CPL) results**			
Preconditioner		**Iterations**	***Total time***
-		170	*0.765s*
ILUT	$p = 5,\ \tau = 1.0e-06$	17	*1.204s*
	$p = 5,\ \tau = 1.0e-08$	17	*1.188s*
	$p = 10,\ \tau = 1.0e-06$	14	*1.358s*
	$p = 10,\ \tau = 1.0e-08$	14	*1.348s*
SSOR	$\omega = 1.3$	64	*0.482s*
	$\omega = 1.6$	84	*0.560s*
	$\omega = 1.9$	123	*0.739s*

Table 5.61: QMR(CPL) run results of femSch$(30, 2)$ matrix.

femSch$(30, 2)$ **QMR(CPX) results**			
Preconditioner		**Iterations**	***Total time***
-		170	*0.714s*
ILUT	$p = 5,\ \tau = 1.0e-06$	17	*1.191s*
	$p = 5,\ \tau = 1.0e-08$	17	*1.193s*
	$p = 10,\ \tau = 1.0e-06$	14	*1.348s*
	$p = 10,\ \tau = 1.0e-08$	14	*1.334s*
SSOR	$\omega = 1.3$	64	*0.461s*
	$\omega = 1.6$	84	*0.534s*
	$\omega = 1.9$	123	*0.708s*

Table 5.62: QMR(CPX) run results of femSch$(30, 2)$ matrix.

femSch$(30, 2)$ Pardiso vs. QMR		
Pardiso best total time		*0.041s*
QMR best total time	CPL	*0.482s*
	CPX	*0.461s*
Pardiso worst total time		*0.085s*
QMR worst total time	CPL	*1.358s*
	CPX	*1.348s*
QMR/Pardiso		*11.2440*

Table 5.63: Pardiso vs. QMR best/worst run results of femSch$(30, 2)$ matrix.

femSch$(60, 2)$ Pardiso results				
#Cores	**Reorder**	**LU**	**Solve**	***Total time***
1	0.018259s	0.059870s	0.006006s	*0.001709s*
2	0.017974s	0.028569s	0.003383s	*0.001704s*
4	0.017708s	0.020542s	0.002938s	*0.001685s*

Table 5.64: Pardiso run results of femSch$(60, 2)$ matrix.

femSch$(60, 2)$ QMR(CPL) results			
Preconditioner		**Iterations**	***Total time***
-		217	*0.904s*
ILUT	$p = 5,\ \tau = 1.0e - 06$	20	*1.223*
	$p = 5,\ \tau = 1.0e - 08$	20	*1.197s*
	$p = 10,\ \tau = 1.0e - 06$	15	*1.366s*
	$p = 10,\ \tau = 1.0e - 08$	15	*1.365s*
SSOR	$\omega = 1.3$	87	*0.578s*
	$\omega = 1.6$	120	*0.741s*
	$\omega = 1.9$	179	*0.943s*

Table 5.65: QMR(CPL) run results of femSch$(60, 2)$ matrix.

femSch$(60,2)$ **QMR(CPX) results**			
Preconditioner		**Iterations**	***Total time***
-		217	*0.865s*
ILUT	$p = 5,\ \tau = 1.0e-06$	20	*1.218s*
	$p = 5,\ \tau = 1.0e-08$	20	*1.222s*
	$p = 10,\ \tau = 1.0e-06$	15	*1.341s*
	$p = 10,\ \tau = 1.0e-08$	15	*1.345s*
SSOR	$\omega = 1.3$	87	*0.542s*
	$\omega = 1.6$	120	*0.659s*
	$\omega = 1.9$	179	*0.883s*

Table 5.66: QMR(CPX) run results of femSch$(60,2)$ matrix.

femSch$(60,2)$ **Pardiso vs. QMR**		
Pardiso best total time		*0.002s*
QMR best total time	CPL	*0.578s*
	CPX	*0.542s*
Pardiso worst total time		*0.002s*
QMR worst total time	CPL	*1.366s*
	CPX	*1.345s*
QMR/Pardiso		*271.0000*

Table 5.67: Pardiso vs. QMR best/worst run results of femSch$(60,2)$ matrix.

5.8 Graphic representation of run results

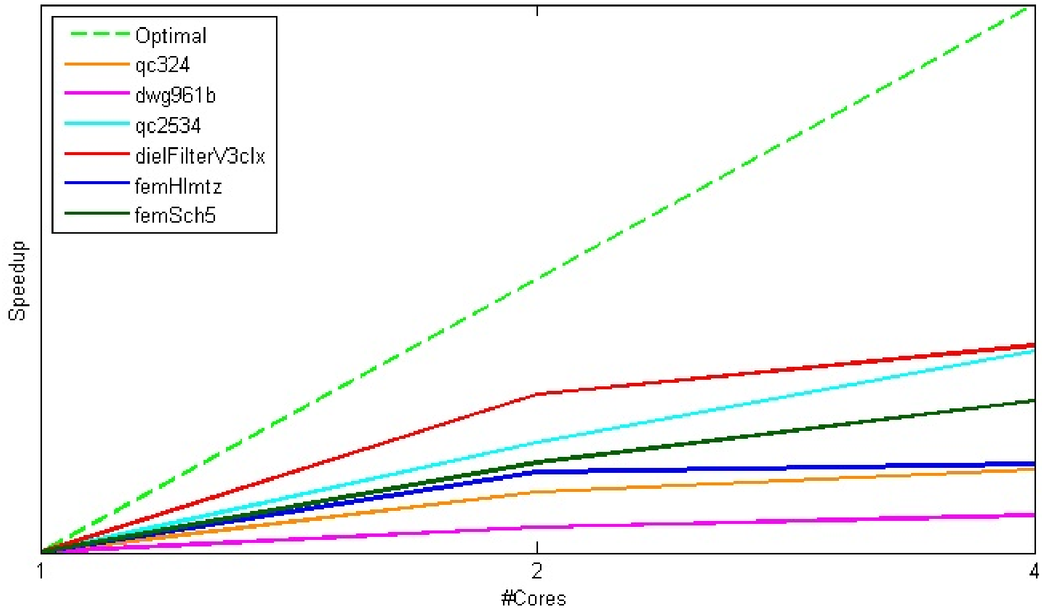


FIGURE 5.5: Pardiso parallel speedup.

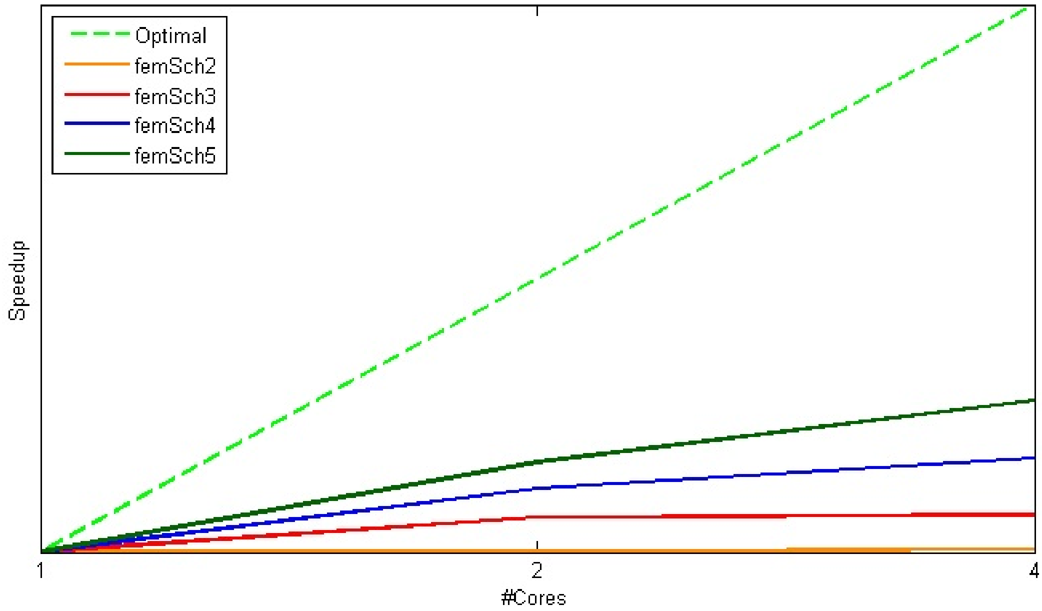


FIGURE 5.6: Pardiso parallel speedup for the femSch matrices.

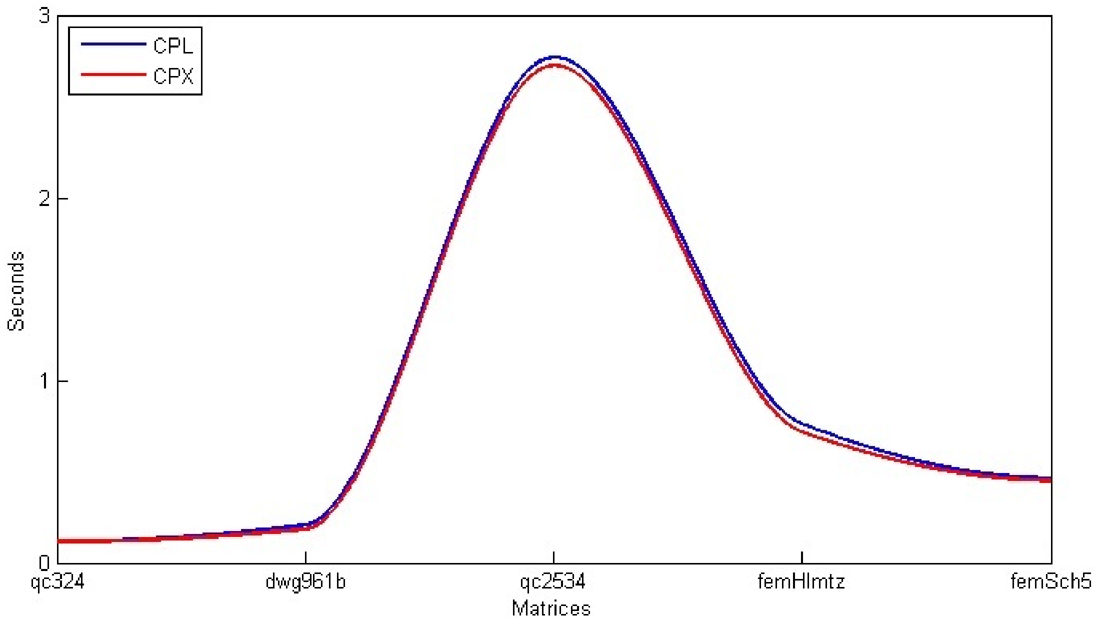


FIGURE 5.7: QMR: CPL vs. CPX best time (ascending matrix size).

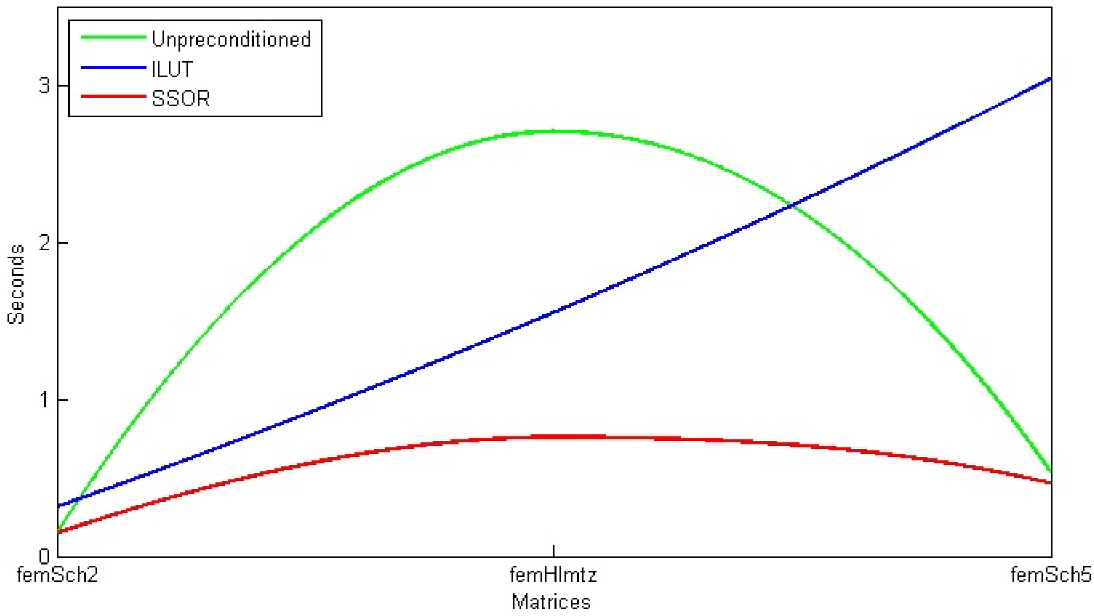


FIGURE 5.8: QMR: Preconditioners comparison (ascending matrix size).

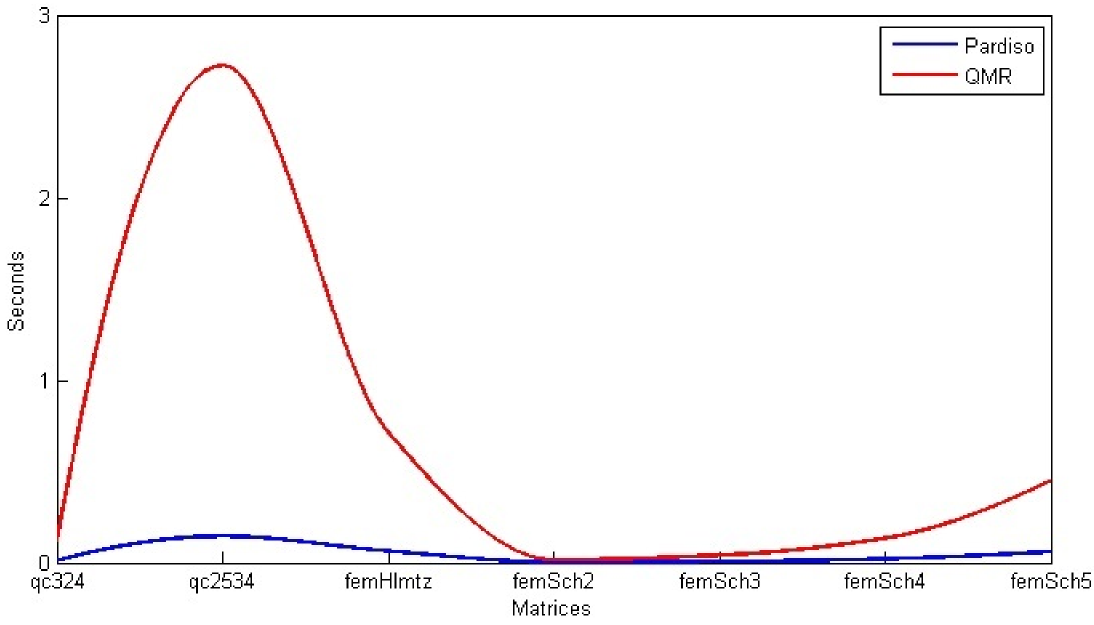


FIGURE 5.9: Pardiso (best times) vs. QMR (best times).

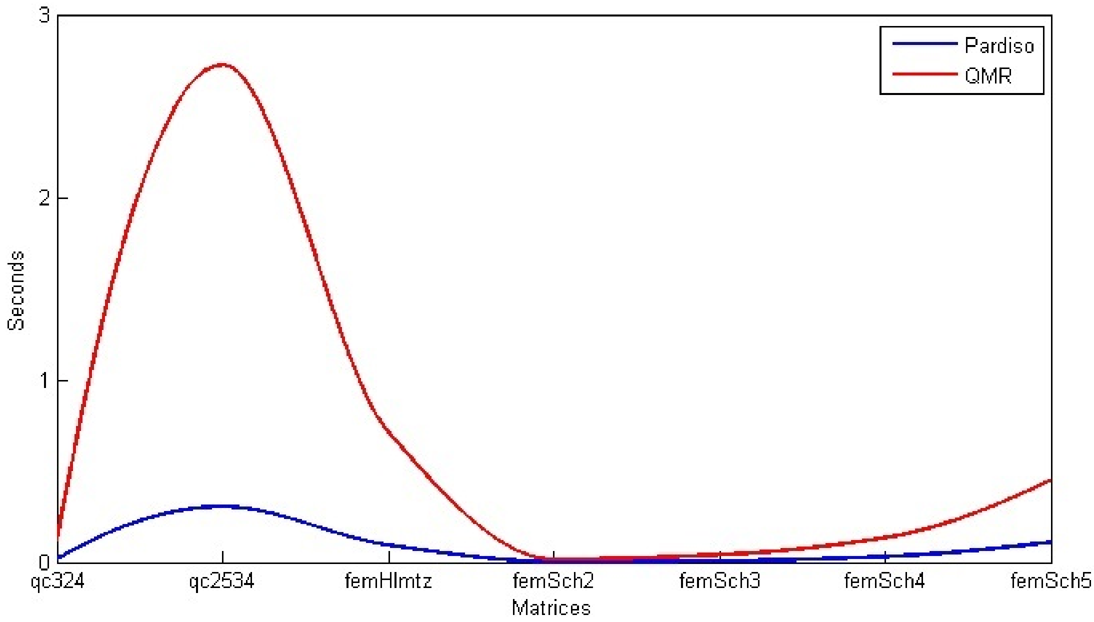


FIGURE 5.10: Pardiso (1 Core) vs. QMR (best times).

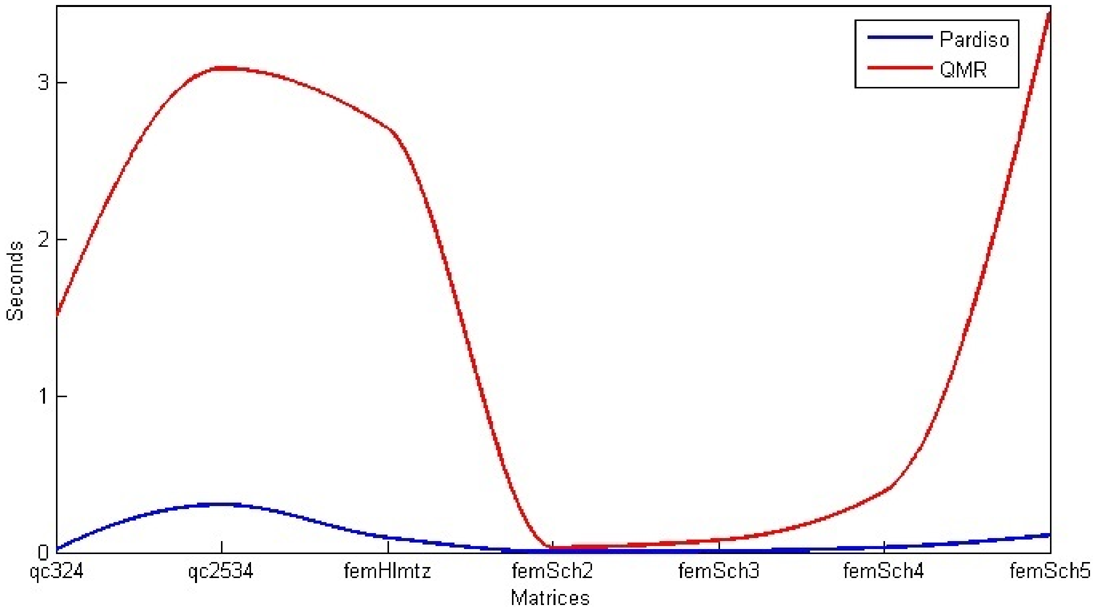


FIGURE 5.11: Pardiso (worst times) vs. QMR (worst times).

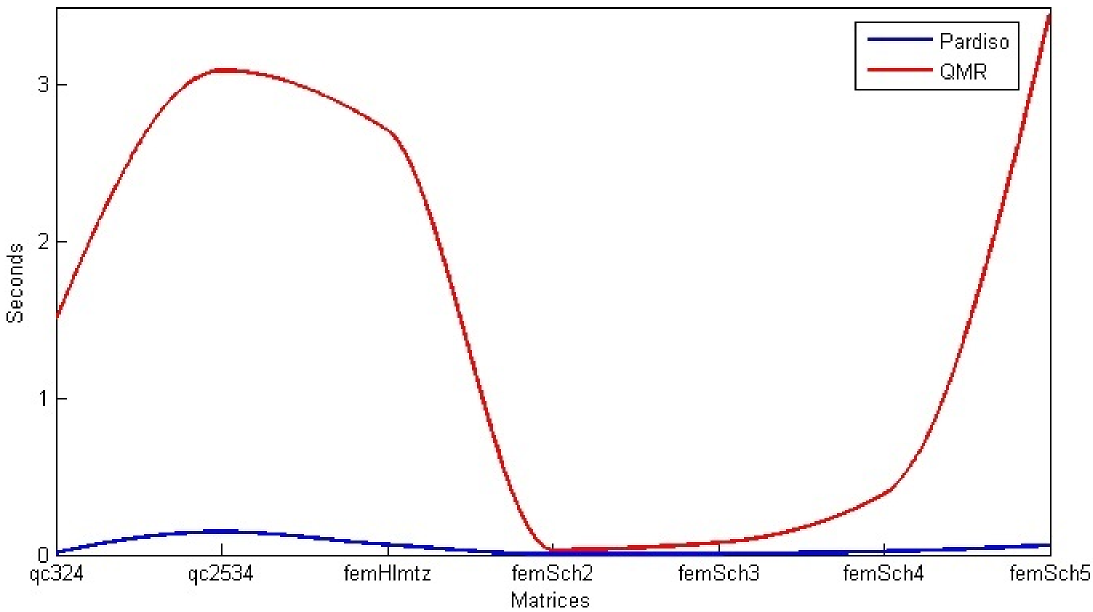


FIGURE 5.12: Pardiso (best times) vs. QMR (worst times).

Bibliography

[1] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. *http://math.nist.gov/MatrixMarket/*, 2007.

[2] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):163–179, 1977.

[3] Ke Chen. *Matrix Preconditioning Techniques and Applications*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2005.

[4] Roland W. Freund and Noel M. Nachtigal. An implementation of the QMR method based on coupled two-term recurrences. *SIAM Journal on Scientific Computing*, 15:313–337, 1994.

[5] Roland W. Freund, Noel M. Nachtigal, and Jennifer C. Reeb. *QMRPACK User's Guide*, 1.3 edition, June 10 1994.

[6] Alan George and Joseph W. H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, 1978.

[7] Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[8] Nikolas J. Higham. Factorizing complex symmetric matrices with positive definite real and imaginary parts. *Mathematics of Computation*, 67(224):1591–1599, 1998.

[9] George Karypis and Vipin Kumar. *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, User Guide*, 5.1.0 edition, March 30 2013.

[10] Scott MacLachlan, Daniel Osei-Kuffuor, and Yousef Saad. Modification and compensation strategies for threshold-based incomplete factorizations. *SIAM Journal on Scientific Computing*, 34(1):A48–A75, 2012.

[11] Dimitrios Mitsoudis, Charalambos Makridakis, and Michael Plexousakis. Helmholtz equation with artificial boundary conditions in a two-dimensional waveguide. *SIAM Journal on Mathematical Analysis*, 44(6):4320–4344, 2012.

[12] Michael Plexousakis. *An Adaptive Nonconforming Finite Element Method for the Nonlinear Schrödinger Equation.* PhD thesis, The University of Tennessee, Knoxville, December 1996.

[13] Michael Plexousakis. Nemesis – an adaptive discontinuous galerkin finite element method for the nonlinear (cubic) schrödinger equation. 2014.

[14] Yousef Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[15] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, second edition, 2003.

[16] Olaf Schenk, Klaus Gärtner, George Karypis, Robert Luce, and Peter Carbonetto. Pardiso 5.0.0 solver project. `http://www.pardiso-project.org/`, 2004.

[17] Olaf Schenk and Gärtner Klaus. *PARDISO User Guide*, 5.0.0 edition, February 07 2014.

[18] Ivan P. Staminirović and Milan B. Tasić. Performance comparison of storage formats for sparse matrices. *Facta Universitatis (NIŠ), Ser. Math. Inform*, 24:39–51, 2009.

[19] James H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, Englewood Cliffs, N.J., 1963.

[20] David M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, 1971.