ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ
ΔΙΚΤΥΩΝ

Υλοποίηση και βελτιστοποίηση συνδυαστικού γραφοθεωρητικού αλγορίθμου για προσομοίωση πολύ μεγάλης κλίμακας γραμμικών κυκλωμάτων

Development and Optimization of a combinatorial multigrid algorithm for large scale circuit simulation

Διπλωματική Εργασία

Δήμος Π. Ντιούδης

Επιβλέποντες Καθηγητές : Σταμούλης Γεώργιος
Καθηγητής

Ευμορφόπουλος Νέστωρ
Επίκουρος Καθηγητής

Βόλος, Ιούνιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Υλοποίηση και βελτιστοποίηση συνδυαστικού γραφοθεωρητικού αλγορίθμου για προσομοίωση πολύ μεγάλης κλίμακας γραμμικών κυκλωμάτων

Διπλωματική Εργασία

Δήμος Π. Ντιούδης

Επιβλέποντες : Σταμούλης Γεώργιος
Καθηγητής

Ευμορφόπουλος Νέστωρ
Επίκουρος Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 27^{η} Ιουνίου 2013

....	....
Γ. Σταμούλης	Ν.Ευμορφόπουλος
Καθηγητής	Επίκουρος Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας.

……………………………..

Δήμος Ντιούδης

Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων Πανεπιστημίου Θεσσαλίας

To my family and friends

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της διπλωματικής εργασίας κ. Γεώργιο Σταμούλη και κ. Νέστορα Ευμορφόπουλο για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου, την άριστη συνεργασία, την συνεχή καθοδήγηση και τις ουσιώδεις υποδείξεις και παρεμβάσεις, που διευκόλυναν την εκπόνηση της πτυχιακής εργασίας.

Επίσης, θα ήθελα να ευχαριστήσω τους φίλους και συνεργάτες του Εργαστηρίου Ε5 για την υποστήριξη και την δημιουργία ενός ευχάριστου και δημιουργικού κλίματος και ιδιαίτερα τον διδακτορικό φοιτητή Κωνσταντή Νταλούκα για τις εύστοχες υποδείξεις του και την συνεχή στηριξή του. Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατά την διάρκεια των σπουδών μου όσο και κατά την εκπόνηση της διπλωματικής εργασίας.

Δήμος Ντιούδης
Βόλος, 2013

Contents

List of Tables

List of Figures

List of Acronyms

AMG	algebraic multigrid
BiCG	biconjugate gradient
CG	conjugate gradients
CMG	combinatorial multigrid
EDA	electronic design automation
GMG	geometric multigrid
GMRES	generalized minimal residual
IC	integrated circuit
MG	multigrid
PCG	preconditioned conjugate gradient
SDD	symmetric diagonally dominant
SOR	successive overrelaxation
SSOR	symmetric successive overrelaxation
SPD	symmetric positive definite
VLSI	very large scale integration

Abstract

Γραμμικά συστήματα της μορφής $Ax = b$, για συμμετρικούς πίνακες με κυρίαρχη διαγώνιο, προκύπτουν πολύ συχνά σε προβλήματα προσομοίωσης πολύ μεγάλης κλίμακας κυκλωμάτων. Την τελευταία δεκαετία έχουν αναπτυχθεί ένα πλήθος εξειδικευμένων επιλυτών με σκοπό να αντιμετωπίσουν περιορισμένες τοπολογίες από συστήματα τέτοιου είδους που προκύπτουν από μία συλλογή ποικίλων προβλημάτων. Σε αυτή την διπλωματική εργασία θα εξηγήσουμε και θα εφαρμόσουμε τη θεωρία γράφων, ένα σύνολο τεχνικών που αναπτύχθηκαν από την επιστημονική κοινότητα, με σκοπό την κατασκευή επιλυτών με αποδεδειγμένες ιδιότητες. Για να παρουσιάσουμε την ισχύ αυτών των τεχνικών, παρουσιάζουμε και περιγράφουμε έναν αποτελεσματικό γραφοθεωρητικό επιλυτή ο οποίος στηρίζεται στις αρχές της θεωρίας γράφων. Ο επιλυτής αντιμετωπίζεις προβλήματα σε αρκετά γενικές και αυθαίρετα σταθμισμένες τοπολογίες που δεν υποστηρίζονταν από προηγούμενες υλοποιήσεις. Επιτυγχάνει εξαιρετικά αποτελέσματα ενώ παράλληλα παρέχει ισχυρές εγγυήσεις για την ταχύτητα σύγκλισης. Η μέθοδος αξιολογηθηκε σε μια ποικιλία εφαρμογών σχετικές με την προσομοίωση κυκλωμάτων.

Λέξεις Κλειδιά:
γραμμικά συστήματα, προρυθμιστές, μέθοδοι επίλυσης, condition number, θεωρία γράφων, multigrid

Abstract

Linear systems of the form $Ax = b$, on symmetric diagonally dominant matrices (SDDs) occur frequently in very large scale circuit simulation. In the past decade a multitude of specialized solvers have been developed to tackle restricted instances of SDD systems for a diverse collection of problems. In this thesis we explain and apply the support theory of graphs, a set of tecnhiques developed by the computer science theory community, to construct SDD solvers with provable properties. To demonstrate the power of these techniques, we describe an efficient multigrid-like solver which is based on support theory principles. The solver tackles problems in fairly general and arbitrarily weighted topologies not supported by prior solvers. It achieves state of the art empirical results while providing robust guarantees on the speed of convergence. The method is evaluated on a variety of circuit simulation applications.

Keywords:
linear systems, preconditioning, solution methods, condition number, support theory, multigrid

Chapter 1

Introduction

1.1 Problem Description

Circuit simulation is a tecnhique for checking and verifying the design of electrical and electronic circuits and systems prior to manufacturing and deployment. It is used across a wide spectrum of applications, ranging from integrated circuits and microelectronics to electrical power distribution networks and power electronics. Circuit simulation is a mature and established art and also remains an important area of research. It uses mathematical models to replicate the behavior of an actual electronic device or circuit. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs. In particular, for integrated circuits, the tooling is expensive, breadboards are impractical, and probing the behavior of internal signals is extremely difficult. Therefore almost all integrated circuit (IC) design relies heavily on simulation.

1.2 Thesis Contribution

The solution of linear systems of equations of the form $Ax = b$ is at the heart of many computations in science, engineering, and other disciplines. Several algorithms are based on solving such sort of linear systems. These algorithms generally produce results of high quality. However, existing solvers are not always efficient, and in many cases they operate only on restricted topologies. The unavailability of reliably efficient solvers has arguably hindered the adoptability of approaches and algorithms based on symmetric diagonally dominant (SDD) systems, especially in applications involving very large systems.

In this thesis we review and apply the support theory of graphs, a set of of techniques developed by the computer science theory community, to construct SDD solvers with provable properties. To demonstrate the power of these techniques, we describe an efficient multigrid-like solver which is based on support theory principles. The solver tackles problems in fairly general and arbitrarily weighted topologies not supported by prior solvers. It achieves state of the art empirical results while providing robust guarantees on the speed of convergence. The method is evaluated on a variety of circuit simulation like applications. The new method is an extension of the preconditioned conjugate gradient method (PCG), and is characterized by the form of the preconditioner [17].

Starting from a MATLAB implementation [18] of the algorithm we transformed the initial MATLAB code into C code. Our purpose was to embed the preconditioner into various electronic design automation (EDA) algorithms and evaluate how well it performs in the context of these algorithms. The systems that we considered were large, sparse, symmetric, and diagonally dominant with non-positive off-diagonals. The results from the evaluation of combinatorial multigrid (CMG) preconditioner showed that CMG solved our SDD linear sytems in some cases even x8 times faster than the other well-known preconditioner which is the Jacobian preconditioner.

1.3 Formation of this thesis

The formation of this thesis is as follows. In chapter 2 we give background material on the existing solution methods of linear equations.

In chapter 3 we review some basic notions of preconditioner matrices. We discuss about the importance of the preconditioning, how it is used and how it helps to the convergence of the methods.

In chapter 4 we give some background material on support graph theory for graphs. We also review how support combine its strength with the strength of preconditioning.

In chapter 5 we give some background material on solvers and present CMG. The theoretical foundation of CMG has been laid in previous work [29], but the solver itself and its application to EDA tools are new.

Finally in chapter 6 we present the experiments we made that compare CMG preconditioner to the Jacobi preconditioner along with the time results we took after the evaluation.

Chapter 2

Solution Methods of the $Ax = b$

2.1 Introduction

There are two broad categories of methods for solving linear equations of the form $Ax = b$ when A is large and sparse: ***direct*** and ***iterative***. While for some tecnhiques such as direct solvers, we only provide brief descriptions, for iterative solvers, we go into more depth to describe the algorithms, since they are of interest to us here.

A direct method for solving the system of equations $Ax = b$ is any method that produces the solution x after a finite number of operations. An example of a direct method is using Gaussian elimination to factor A into matrices L and U where L is lower triangular and U is upper triangular, then solving the triangular systems by forward and back substitution. Direct methods are typically preferred for dense linear systems. The problem with direct methods for sparse systems is that the amount of computational effort and storage required can be prohibitive [7].

An alternative to direct methods of solution are iterative methods, which involve the construction of a sequence $\{x^{(i)}\}$ of approximations to the solution x, for which $x^{(i)} \to x$. Iterative methods for solving general, large sparse linear systems have been gaining popularity in many areas of scientific computing. Until recently, direct solution methods were often preferred to iterative methods in real applications because of their robustness and predictable behavior. However, a number of efficient iterative solvers were discovered and the increased need for solving very large linear systems triggered a noticeable and rapid shift toward iterative techniques in many applications [19].

In this thesis we are interested only in iterative methods on sparse matrices. But before we analyze some of the most well-known, lets see what the term ***sparse*** refers to.

2.1.1 Sparsity Overview

Consider the solution of linear systems of the form

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

where A is an nxn matrix, and both x and b are $nx1$ vectors. Of special interest is the case where A is large and sparse. The term ***sparse*** above refers to the relative number of non-zeros in the matrix A. An nxn matrix A is considered to be ***sparse*** if A has only $O(n)$ non-zero entries. In this case, the majority of the entries in the matrix are zeros, which do not have to be explicitly stored. An nxn dense matrix has $\Omega(n^2)$ non-zeros. There are many ways of storing a sparse matrix. Whichever method is chosen, some form of compact data is required that avoids storing the numerically zero entries in the matrix. It needs to be simple and flexible so that it can be used in a wide range of matrix operations. This need is met by the primary data structure in CSparse[1], a compressed-column matrix [6]. Some basic operations that operate on this data structure are matrix-vector multiplication, matrix-matrix multiplication, matrix addition, and transpose.

The simplest sparse matrix data structure is a list of the nonzero entries in arbitrary order. The list consists of two integer arrays i and j and one real array x of length equal to the number of entries in the matrix. For example, the matrix [5]

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}, \tag{2.2}$$

is presented in ***zero-based triplet*** form below. A zero-based data structure for an m-by-n matrix contains row and column indices in the range 0 to m-1 and n-1, respectively.

$$\begin{aligned} i &= \{2, \quad 1, \quad 3, \quad 0, \quad 1, \quad 3, \quad 3, \quad 1, \quad 0, \quad 2\} \\ j &= \{2, \quad 0, \quad 3, \quad 2, \quad 1, \quad 0, \quad 1, \quad 3, \quad 0, \quad 2\} \\ x &= \{2, \quad 1, \quad 3, \quad 0, \quad 1, \quad 3, \quad 3, \quad 1, \quad 0, \quad 2\} \end{aligned}$$

The triplet form is simple to create but difficult to use in most sparse matrix algorithms. The ***compressed-column*** form is more useful and is used in almost all functions in CSparse. An m-by-n sparse matrix that can contain up to $nzmax$ entries is represented with an integer array p of length $n+1$, an integer array i of length $nzmax$, and a real array x of length $nzmax$. Row indices of entries in column j are stored in $i[p[j]]$ through $i[p[j+1]-1]$, and the corresponding numerical values are stored in the same locations in x. The first entry $p[0]$ is always zero, and $p[n] \leq nzmax$ is the number of actual entries in the matrix. The example matrix (2.2) is represented as

$$\begin{aligned} p &= \{0, \quad 3, \quad 6, \quad 8, \quad 10\} \\ i &= \{0, \quad 1, \quad 3, \quad 1, \quad 2, \quad 3, \quad 0, \quad 2, \quad 1, \quad 3\} \\ x &= \{4.5, \quad 3.1, \quad 3.5, \quad 2.9, \quad 1.7, \quad 0.4, \quad 3.2, \quad 3.0 \quad 0.9, \quad 1.0\} \end{aligned}$$

One of the goals of dealing with sparse matrices is to make efficient use of the sparsity in order to minimize storage throughout the computations, as well as to minimize the required number of operations. Sparse linear systems are often solved using different computational techniques than those employed to solve dense systems.

2.2 Overview of the Methods

Below are short descriptions of each of the methods to be discussed, along with brief notes on the classification of the methods in terms of the class of matrices for which they are most appropriate. In later sections of this chapter more detailed descriptions of these methods are given [1].

[1]CSparse is a C library which implements a number of direct methods for sparse linear systems.

- Stationary Methods
  - Jacobi.
    The Jacobi method is based on solving for every variable locally with respect to the other variables; one iteration of the method corresponds to solving for every variable once. The resulting method is easy to understand and implement, but convergence is slow.
  - Gauss-Seidel
    The Gauss-Seidel method is like the Jacobi method, except that it uses updated values as soon as they are available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, though still relatively slowly.
  - SOR
    Successive Overrelaxation (SOR) can be derived from the Gauss-Seidel method by introducing an extrapolation parameter ω. For the optimal choice of ω, SOR may converge faster than Gauss-Seidel by an order of magnitude.
- Nonstationary Methods
  - Conjugate Gradient (CG).
    The conjugate gradient method derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors. These vectors are the residuals of the iterates. They are also the gradients of a quadratic functional, the minimization of which is equivalent to solving the linear system. conjugate gradients (CG) is an extremely effective method when the coefficient matrix is symmetric positive definite (SPD), since storage for only a limited number of vectors is required.
  - Generalized Minimal Residual (GMRES).
    The Generalized Minimal Residual method computes a sequence of orthogonal vectors, and combines these through a least-squares solve and update. However, it requires storing the whole sequence, so that a large amount of storage is needed. For this reason, restarted versions of this method are used. In restarted versions, computation and storage costs are limited by specifying a fixed number of vectors to be generated. This method is useful for general nonsymmetric matrices.
  - BiConjugate Gradient (BiCG).
    The biconjugate gradient (BiCG) method generates two CG-like sequences of vectors, one based on a system with the original coefficient matrix A, and one on AT . Instead of orthogonalizing each sequence, they are made mutually orthogonal, or “bi-orthogonal”. This method, like CG, uses limited storage. It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular, and there is a possibility that the method will break down. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration.

2.3 Stationary Methods

Iterative methods that can be expressed in the simple form

$$x^{(k)} = Bx^{(k-1)} + c, \tag{2.3}$$

(where neither B nor c depend upon the iteration count k) are called *stationary* iterative methods. In this section, we present the three main stationary iterative methods: the ***Jacobi method***, the ***Gauss-Seidel method*** and the ***Successive Overrelaxation (SOR) method***.

2.3.1 The Jacobi Method

The Jacobi method is easily derived by examining each of the n equations in the linear system $Ax = b$ in isolation. If in the ith equation

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i,$$

we solve for the value of x_i while assuming the other entries of x remain fixed, we obtain

$$x_i = (b_i - \sum_{j \neq i} a_{i,j} x_j)/a_{i,i}. \tag{2.4}$$

This suggests an iterative method defined by

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{i,j} x_j^{(k-1)})/a_{i,i}. \tag{2.5}$$

which is the Jacobi method. Note that the order in which the equations are examined is irrelevant, since the Jacobi method treats them independently. For this reason, the Jacobi method is also known as the method of ***simultaneous displacements***, since the updates could in principle be done simultaneously.

In matrix terms, the definition of the Jacobi method in (2.3) can be expressed as

$$x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b, \tag{2.6}$$

where the matrices D, $-L$ and $-U$ represent the diagonal, the strictly lower-triangular, and the strictly upper-triangular parts of A, respectively.

The pseudocode for the Jacobi method is given in below. Note that an auxiliary storage vector, $\overline{x}$ is used in the algorithm. It is not possible to update the vector x in place, since values from $x^{(k-1)}$ are needed throughout the computation of $x^{(k)}$.

Algorithm 1 The Jacobi Method.

```
1: Choose an initial guess $x^{(0)}$ to the solution $x$.
2: for $k = 1, 2, ...$ do
3:   for $i = 1, 2, ..., n$ do
4:     $\overline{x}_i = 0$
5:     for $j = 1, 2, ..., i-1, i+1, ...n$ do
6:       $\overline{x}_i = \overline{x}_i + a_{i,j} x_j^{(k-1)}$
7:     end for
8:     $\overline{x}_i = (b_i - \overline{x}_i)/a_{i,i}$
9:   end for
10:  $x^{(k)} = \overline{x}$
11:  check convergence; continue if necessary
12: end for
```

2.3.2 The Gauss-Seidel Method

Consider again the linear equations (2.2). If we proceed as with the Jacobi Method, but now assume that the equations are examined one at a time in sequence, and the previously computed results are used as they are available, we obtain the Gauss-Seidel method:

Algorithm 2 The Gauss-Seidel Method.

```
1: Choose an initial guess $x^{(0)}$ to the solution $x$.
2: for $k = 1, 2, ...$ do
3:   for $i = 1, 2, ..., n$ do
4:     $\sigma = 0$
5:     for $j = 1, 2, ..., i-1$ do
6:       $\sigma = \sigma + a_{i,j} x_j^{(k)}$
7:     end for
8:     for $j = i+1, ..., n$ do
9:       $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
10:    end for
11:    $x_i^{(k)} = (b_i - \sigma)/a_{i,i}$
12:  end for
13:  check convergence; continue if necessary
14: end for
```

$$x_i^{(k)} = (b_i - \sum_{i>j} a_{i,j} x_j^{(k)} - \sum_{j>i} a_{i,j} x_j^{(k-1)})/a_{i,i}. \tag{2.7}$$

Two important facts about the Gauss-Seidel method should be noted. First, the computations in (2.5) appear to be serial. Since each component of the new iterate depends upon all previously computed components, the updates cannot be done simultaneously as in the Jacobi method. Second, the new iterate $x^{(k)}$ depends upon the order in which the equations are examined. The Gauss-Seidel method is sometimes called the method of successive displacements to indicate the dependence of the iterates on the ordering. If this ordering is changed, the components of the new iterate (and not just their order) will also change.

These two points are important because if A is sparse, the dependency of each component of the new iterate on previous components is not absolute. The presence of zeros in the matrix may remove the influence of some of the previous components. Using a judicious ordering of the equations, it may be possible to reduce such dependence, thus restoring the ability to make updates to groups of components in parallel. However, reordering the equations can affect the rate at which the Gauss-Seidel method converges. A poor choice of ordering can degrade the rate of convergence; a good choice can enhance the rate of convergence.

In matrix terms, the definition of the Gauss-Seidel method in (2.5) can be expressed as

$$x^{(k)} = (D - L)^{-1}(Ux^{(k-1)} + b). \tag{2.8}$$

As before D, $-L$ and $-U$ represent the diagonal, lower-triangular, and upper-triangular parts of A, respectively.

The pseudocode for the Gauss-Seidel algorithm is given in at the top of this page.

2.3.3 The Successive Overrelaxation Method (SOR)

The Successive Overrelaxation Method, or SOR, is devised by applying extrapolation to the Gauss-Seidel method. This extrapolation takes the form of a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component:

$$x_i^{(k)} = \omega \overline{x_i}^{(k)} + (1 - \omega)x_i^{(k-1)}.$$

(where $\overline{x_i}$ denotes a Gauss-Seidel iterate, and ω is the extrapolation factor). The idea is to choose a value for ω that will accelerate the rate of convergence of the iterates to the solution.

Algorithm 3 The SOR Method.

```
1: Choose an initial guess $x^{(0)}$ to the solution $x$.
2: for $k = 1, 2, ...$ do
3:   for $i = 1, 2, ..., n$ do
4:     $\sigma = 0$
5:     for $j = 1, 2, ..., i - 1$ do
6:       $\sigma = \sigma + a_{i,j}x_j^{(k)}$
7:     end for
8:     for $j = i + 1, ..., n$ do
9:       $\sigma = \sigma + a_{i,j}x_j^{(k-1)}$
10:    end for
11:    $\sigma = (b_i - \sigma)/a_{i,i}$
12:    $x_i^{(k)} = x_i^{(k-1)} + \omega(\sigma - x_i^{(k-1)})$
13:  end for
14:  check convergence; continue if necessary
15: end for
```

In matrix terms, the successive overrelaxation (SOR) algorithm can be written as follows:

$$x^{(k)} = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)x^{(k-1)} + \omega(D - \omega L)^{-1}b. \tag{2.9}$$

The pseudocode for the SOR algorithm is given above.

2.4 Nonstationary Methods

Nonstationary methods differ from stationary methods in that the computations involve information that changes at each iteration. Typically, constants are computed by taking inner products of residuals or other vectors arising from the iterative method.

2.4.1 Generalized Minimal Residual (GMRES)

The GMRES method generates a sequence of orthogonal vectors, but in the absence of symmetry this can no longer be done with short recurrences; instead, all previously computed vectors in the orthogonal sequence have to be retained. For this reason are used restarted versions of the method . The GMRES algorithm has the property that residual norm $\|b - Ax^i\|$ can be computed without the iterate having been formed. Thus, the expensive action of forming the iterate can be postponed until the residual norm is deemed small enough. The GMRES iterates are constructed as:

$$x^i = x^0 + y_1 u^1 + ... + y_i u^i, \tag{2.10}$$

The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand.

The pseudocode for the restarted GMRES algorithm with preconditioner M is given in next page.

Algorithm 4 The Preconditioned GMRES Method.

```
$x^{(0)}$ is an initial guess
for $i = 1, 2, ...$ do
    Solve $r$ from $Mr = b - Ax^{(0)}$
    $v^{(1)} = r/\|r\|_2$
    $s := \|r\|_2 e_1$
    for $i = 1, 2, ..., m$ do
        Solve $w$ from $Mw = Av^{(i)}$
        for $k = 1, ..., i$ do
            $h_{k,i} = (w, v^{(k)})$
            $w = w - h_{k,i} v^{(k)}$
        end for
        $h_{i+1,j} = \|w\|_2$
        $v^{(i+1)} = w/h_{i+1,i}$
        apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
        construct $J_i$ acting on $i$th and $(i+1)$st component
        of $h_{.,i}$ such that $(i+1)$st component of $J_i h_{.,i}$ is 0
        $s := J_i s$
        if $s(s+1)$ is small enough then (UPDATE($\overline{x}, i$) and quit)
    end for
    UPDATE($\overline{x}, m$)
end for

In this schem UPDATE($\overline{x}, i$)
replaces the following computations:

Compute $y$ as the solution of $Hy = \overline{s}$, in which
the upper $ixi$ triangular part of $H$ has $h_{i,j}$ as
its elements (in least squares sense if $H$ is singular),
$\overline{s}$ represents the first $i$ components of $s$
$s^{(i+1)} = \|b - A\overline{x}\|_2$
if $\overline{x}$ is an accurate enough approximation then quit
else $x^{(0)} = \overline{x}$
```

2.4.2 Conjugate Gradient (CG)

The Conjugate Gradient method is an effective method for symmetric positive definite systems. It is the oldest and best known of the nonstationary methods discussed here. The method proceeds by generating vector sequences of iterates (***i.e.***, successsive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors needs to be kept in memory. In every iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.

Algorithm 5 The Preconditioned Conjugate Gradient Method.

```
1: Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2: for $i = 1, 2, ...$ do
3:   solve $Mz^{(i-1)} = r^{(i-1)}$
4:   $\varrho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
5:   if $i = 1$ then
6:     $p^{(1)} = z^{(0)}$
7:   else
8:     $\beta_{i-1} = \varrho_{i-1}/\varrho i - 2$
9:     $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
10:  end if
11:  $q^{(i)} = Ap^{(i)}$
12:  $\alpha_i = \varrho_{i-1}/p^{(i)^T} q^{(i)}$
13:  $x^{(i)} = x^{(i-1)} + \alpha p^{(i)}$
14:  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
15:  check convergence; continue if necessary
16: end for
```

The pseudocode for the Preconditioned Conjugate Gradient Method is given above. It uses a preconditioner M; for $M = I$ one obtains the unpreconditioned version of the Conjugate Gradient Algorithm.

2.4.3 BiConjugate Gradient (BiCG)

The Conjugate Gradient method is not suitable for nonsymmetric systems because the residual vectors cannot be made orthogonal with short recurrences. The GMRES method retains orthogonality of the residuals by using long recurrences, at the cost of a larger storage demand. The BiConjugate Gradient method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization. The update relations for residuals in the Conjugate Gradient method are augmented in the BiConjugate Gradient method by relations that are similar but based on A^T instead of A.

The pseudocode for the Preconditioned BiConjugate Gradient Method with preconditioner M is given in the top of the next page.

Algorithm 6 The Preconditioned BiConjugate Gradient Method.

```
1: Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2: Choose $\overline{r}^{(0)}$ (for example, $\overline{r}^{(0)} = r^{(0)}$)
3: for $i = 1, 2, ...$ do
4:     solve $Mz^{(i-1)} = r^{(i-1)}$
5:     solve $M^T\overline{z}^{(i-1)} = \overline{r}^{(i-1)}$
6:     $\varrho_{i-1} = z^{(i-1)^T}\overline{r}^{(i-1)}$
7:     if $\varrho_{i-1} = 1$ then
8:         method fails
9:     end if
10:    if $i = 1$ then
11:        $p^{(i)} = z^{(i-1)}$
12:        $\overline{p}^{(i)} = \overline{z}^{(i-1)}$
13:    else
14:        $\beta_{i-1} = \varrho_{i-1}/\varrho i - 2$
15:        $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
16:        $\overline{p}^{(i)} = \overline{z}^{(i-1)} + \beta_{i-1}\overline{p}^{(i-1)}$
17:    end if
18:    $q^{(i)} = Ap^{(i)}$
19:    $\overline{q}^{(i)} = A^T\overline{p}^{(i)}$
20:    $\alpha_i = \varrho_{i-1}/\overline{p}^{(i)^T}q^{(i)}$
21:    $x^{(i)} = x^{(i-1)} + \alpha p^{(i)}$
22:    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
23:    $\overline{r}^{(i)} = \overline{r}^{(i-1)} - \alpha_i\overline{q}^{(i)}$
24:    check convergence; continue if necessary
25: end for
```

2.5 Computational Aspects of the Methods

Efficient solution of a linear system includes the selection of the proper choice of iterative method. However, to obtain good performance, consideration must also be given to the computational kernels of the method and how efficient they can be executed on the target architecture. The performance of direct methods, is largely that of the factorization of the matrix. However, this lower efficiency of execution does not imply anything about the total solution time for a given system. Furthermore, iterative methods are usually simpler to implement than direct methods, and since no full factorization has to be stored, they can handle much larger sytems than direct methods.

Method	Inner Product	SAXPY	Matrix-Vector Product	Precond Solve	Storage Reqmnts
JACOBI			1^a		matrix+$3n$
Gauss Seidel		1	1^a		
SOR		1	1^a		matrix+$2n$
GMRES	$i+1$	$i+1$	1	1	matrix + $(i+5)n$
CG	2	3	1	1	matrix + $6n$
BiCG	2	5	1/1	1/1	matrix + $10n$

Table 2.1: Summary of Operations for Iteration i. "a/b" means "a" multiplications with the matrix and "b" with its transpose, and storage requirements for the methods in iteration i: n denotes the order of the matrix.

2.6 Multigrid Methods

Before closing this chapter we would like to discuss about the multigrid (MG) methods. MG methods in numerical analysis is defined as a group of algorithms for solving differential equations using a hierarchy of discretizations. They are an example of a class of techniques called multiresolution methods, very useful in problems exhibiting multiple scales of behavior. For example, many basic relaxation methods exhibit different rates of convergence for short- and long-wavelength components, suggesting these different scales be treated differently, as in a Fourier analysis approach to multigrid. MG methods can be used as solvers as well as preconditioners.

The main idea of MG is to accelerate the convergence of a basic iterative method by global correction from time to time, accomplished by solving a coarse problem[2]. This principle is similar to interpolation between coarser and finer grids. The typical application for multigrid is in the numerical solution of elliptic partial differential equations in two or more dimensions.

Multigrid can be applied in combination with any of the common discretization techniques. MG methods are among the fastest solution techniques known today. In contrast to other methods, multigrid methods are general in that they can treat arbitrary regions and boundary conditions. They do not depend on the separability of the equations or other special properties of the equation.

[2]Coarse problem is an auxiliary system of equations used in an iterative method for the solution of a given larger system of equations. It is basically a version of the same problem at a lower resolution, retaining its essential characteristics, but with fewer variables.

Chapter 3

Introduction to Preconditoners

3.1 Introduction

In chapter 2 we discussed about many iterative methods. The convergence rate of iterative methods depends on spectral properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent in the sense that it has the same solution, but that has more favorable spectral properties. A ***preconditioner*** is a matrix that effects such a transformation. For SPD systems, the rate of convergence of the conjugate gradient method depends on the distribution of the eigenvalues of A. The purpose of preconditioning is that the transformed matrix in question will have a smaller spectral condition number, and/or eigenvalues clustered around 1. For nonsymmetric problems the situation is more complicated, and the eigen-values may not describe the convergence of nonsymmetric matrix iterations like GMRES. On parallel machines there is a further tradeoff between the efficacy of a preconditioner in the classical sense, and its parallel efficiency. Many of the traditional preconditioners have a large sequential component.

If M is a nonsingular matrix that approximates A, then the linear system (3.1) has the same solution as (2.1) but must be significantly easier to solve.

$$M^{-1}Ax = M^{-1}b, \tag{3.1}$$

$$AM^{-1}y = b, x = M^{-1}y \tag{3.2}$$

$$M_1^{-1}AM_2^{-1}y = M_1^{-1}b, x = M_2^{-1}y \tag{3.3}$$

The system (3.1) is preconditioned from the left, (3.2) is preconditioned from the right. At (3.3) is performed split preconditioning where the preconditioner is $M = M1M2$.

Iterative algorithms such as the the Conjugate Gradient method, converge to a solution using only matrix-vector products with A. It is well known that iterative algorithms suffer from slow convergence properties when the condition number of A, $\kappa(A)$, which is defined as the ration of the largest over the minimum eigenvalue of A, is large. What preconditioned iterative methods attempt to do is to remedy the problem by changing the linear system to $M^{-1}Ax = M^{-1}b$. In this case, the algorithms use matrix-vector products with A, and solve linear systems of the form $My = z$. So now the speed of convergence depends on the ***condition number*** $\kappa(A, M)$.

The condition number is defined as:

$$\kappa(A, M) = \max_x \frac{x^T Ax}{x^T Mx} \cdot \max_x \frac{x^T Mx}{x^T Ax} \tag{3.4}$$

where x is taken to be outside the null space of A. There are two contradictory goals one has to deal in constructing a preconditioner M: (i) The linear systems in M must be easier than those in A to solve, (ii) The condition number must be small so it will minimize the number of iterations.

Historically, preconditioners were natural parts of the matrix A. We analyze some of the most well-known preconditioners below.

3.2 Jacobi Preconditioner

The simplest preconditioner consists of just the diagonal of the matrix

$$m_{i,j} = \begin{cases} a_{i,i} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

This is known as the (point) Jacobi preconditioner.

For the model problem, $\kappa(B^{-1}A) = O(n) = \kappa(A)$, so the asymptotic rate of convergence is not improved with diagonal scaling. B in this case does not need to be factored. The storage required for the preconditioner is $O(n)$ since it is a sparse matrix. And, the preconditioner system is very easy to solve, since it simply requires dividing each vector entry by the corresponding diagonal entry of B.

Even through the asymptotic rate of convergence is not improved, diagonal scaling can sometimes make the difference between convergence and non-convergence for an ill-conditioned matrix A. Moreover, diagonal scaling generally achieves some reduction in the number of iterations, and is so cheap to apply that it might as well be done.

3.3 SSOR Preconditioner

Another example of a preconditioner is the SSOR preconditioner which like the Jacobi preconditioner, can be easily derived from the coefficient matrix without any work.

Assume we have a symmetric matrix A. If this matrix is decomposed as

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

in its diagonal, lower, and upper triangular part the SSOR matrix is defined as

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

The SSOR matrix is given in factored form, so this preconditioner shares many properties of other factorization-based methods. For example, its suitability for vector processors or parallel architectures depends strongly on the ordering of the variables.

3.4 Incomplete Factorization Preconditioners

A broad class of preconditioners is based on incomplete factorizations of the coefficient matrix. We call a factorization incomplete if during the factorization process certain fill elements, nonzero elements in the factorization in positions where the original matrix had a zero, have been ignored. Such a preconditioner is then given in factored form $M + LU$ with L lower and U upper triangular. The efficacy of the preconditioner depends on how well M^{-1} approximates A^{-1}.

When a sparse matrix is factored by Gaussian elimination, fill-in usually takes place. In that case, sparsity-preserving pivoting techniques can be used to reduce it. The triangular factors L and U of the coefficient matrix A are considerably less sparse than A.

Sparse direct methods are not considered viable for solving very large linear systems due to time and space limitations , however, by discarding part of the fill-in in the course of the factorization process, simple but powerful preconditioners can be obtained in the form $M = \overline{LU}$ m where $\overline{L}$ and $\overline{U}$ are the incomplete (approximate) LU factors.

Summarizing, it can be said that existing solutions to the problem for incomplete factorization preconditioners for general SPD matrices follow one of two cases: simple inexpensive fixes that result in low quality preconditioners in terms of convergence rating, or sophisticated, expensive strategies that produce high quality preconditioners.

Chapter 4

Support Theory for Graphs

4.1 Introduction

Support theory, is a recent methodology for bounding condition numbers of preconditioned systems. More specifically, it is a set of tools and techniques for bounding extremal eigenvalues. For some iterative methods (conjugate gradients in particular), the ratio of largest to smallest eigenvalues provides an upper bound on the number of iterations.In this section we review fragments of support theory that are relevant to the design of the CMG solver. For an extensive exposition of support theory we refer the reader to [3].

4.2 Graph Theory

In this section, we will review some basic, relevant results in graph theory. First we start with the following basic definitions.

An *undirected graph* $G = (V, E)$ is a collection V of *nodes* or *vertices*, together with a set E of *edges* where each edge in E is an unordered pair of nodes. A $self - loop$ is an edge in wich the vertices are identical. We denote the cardinality of a set of vertices S by $\|S\|$, and the cardinality of a set of edges E by $\|E\|$. An udirected graph is depicted as a set of points connected by lines.

A graph $G = (V, E)$, is said to be *ordered* if each of the n vertices in V is assigned a unique number in the range $1, ..., n$; such an assignment is called ordering. Given an ordered graph G, we will denote vertices by $V = \{v_1, ...v_n\}$, and the edges $E = \{e_1, ...e_n\}$, where $e_i = (v_j, v_k) = (v_k, v_j)$, for some j, k. We will assume that all graphs are ordered.

Let $G = (V, E)$ be a graph. If $e_i = (v_k, v_j) \in E$, then vertices v_j and v_k are called *adjacent*, denoted v_j and v_k. Let $v \in V$; the *degree* of v, $deg(v)$, is the number of distinct vertices adjacent to v.

A *complete* graph is a graph in which all vertices are pairwise adjacent. We denote by K_n the complete graph on n vertices.

A *walk* is an alternating sequence of vertices and edges that begins and ends with a vertex, such that any edge in the sequence connects the vertex preceding it to the vertex following it.

A *path* is a walk in which all the vertices are distinct.

A *cycle* is a walk in which the first and the last vertex are the same.

A graph is *connected* if there exists a path between every pair of vertices. Let $G_1, ..., G_m$ be subgraphs of G such that each G_i is connected and there exists no edges between G_j and G_k for $j \neq k$; then the G_j are called the connected components of G.

A *tree* is a connected graph with no cycles. A $forest$ is a graph with nocycles, and is therefore a collection of trees.

A *directed graph* G is a graph in which the edges are ordered pairs; that is, $v_j, v_k) \neq (v_k, v_j)$. For an edge $e = (v_j, v_k)$, v_j is termed to tail of the edge, and v_k is the head. A directed graph G is depicted as a set of points connected by lines with arrowheads denoting the orientation from tail to head.

A *weighted graph* G (directed or undirected) is a graph together with function $w : E \rightarrow \Re$, which assigns weights to edges.

Let $G = (V(G), E(G))$ and $H = (V(H), E(H))$ be graphs. H is *subgraph* of G if $V(H) \subseteq V(G)$, and $E(H) \subseteq E(G)$. G is then a *supergraph* of H.

Let $S \subseteq V(G)$. Let H be the subgraph of G given by $V(H) = S$, and $(v_i, v_j) \in E(H)$ iff $v_i \in S$ and $v_j \in S$. Then H is the subgraph of G induced by the set S.

Let $G = (V, E)$ be a graph and $S \subseteq V$. Let H be the subgraph of G induced by S. The the $frontier$, or *boundary*, of H is the set of edges (v_i, v_j) such that either $v_i \in S$ and $v_j \ni S$, or $v_i \ni S$ and $v_j \in S$.

Let G and H be graphs. An *embedding* of H into G is a mapping of vertices of H onto vertices of G, and edges of H onto paths in G. The *dilation* of the embedding is the length of the longest path in G onto wich an edge of H is mapped; we denote the dilation of the embedding by $\delta(G, H)$. The *conjestion* of an edge e in G is the number of paths of the embedding that contain e. The conjestion of the embedding is the maximum conjestion of the edges in G. We denote the conjestion of the embedding by $\gamma(G, H)$.

4.3 Graphs as electric networks

The cornerstone of combinatorial preconditioners is the following intuitive yet paradigm-shifting idea explicitly proposed by Vaidya [22]: ***A preconditioner of a graph A should be the Laplacian of a simpler graph B, derived in a principled fashion from A***.

There is an analogy between graph Laplacians and resistive networks [8] . If G is seen as an electrical network with the resistance between nodes i and j being $1/w_{i,j}$, then in the equation $Au = i$, if u is the vector of voltages at the node, i is the vector of currents. Also, the quadratic form $u^T Au = \sum_{i,j} w_{i,j}(v_i - v_j)^2$ expresses the ***power dissipation*** on G, given the node voltages v. In view of this, the construction of a good preconditioner B amounts to the construction of a simpler resistive network with an energy profile close to that of A.

The **support** of A by B, defined as $s(A/B) = \max_v \frac{v^T Av}{v^T Bv}$ is the number of copies of B that are needed to support the power dissipation in A, for all settings of voltages. The principlar reason behind the introduction of the notion of support, is to express its local nature, captured by the Splitting Lemma.

Lemma (Splitting Lemma) If $A = \sum_{i=1}^{m} A_i$ and $B = \sum_{i=1}^{m} B_i$, where A_i, B_i are Laplacians, then $s(A, B) \leq \max_i s(A_i, B_i)$

The Splitting Lemma allows us to bound the support of A by B, by splitting the power dissipation in A into small local pieces, and "supporting" them by also local pieces in B.

For instance, in his work Vaidya proposed to take B as the maximal weight spanning tree of A. Then, it is easy to show that $s(A, B) \leq 1$, intuitively because more resistances always dissipate more power. In order to bound $s(A, B)$, the basic idea to let A_i be edges on A (the ones not existing in B), an let B_i be the unique path in the tree that connects the two end-points of A_i. Then one can bound separately each $s(A_i, B_i)$. In fact, it can be shown that any edge in A that doesn't exist in B, can be supported only by the path B_i.

As a toy example, consider the example in Figure 4.1 of the two (dashed) edges A1, A2 and their two paths in the spanning tree (solid) that share one edge e.

In this example, the **dilation** of the mapping is equal to 3, i.e. the length of the longest of two paths. Also, as e is uses two times, we say that the **congestion** of the mapping is equal to 2. A core Lemma in Support Theory [[3], [2]] is that the support can be upper bounded by the product congestion$*$dilation.

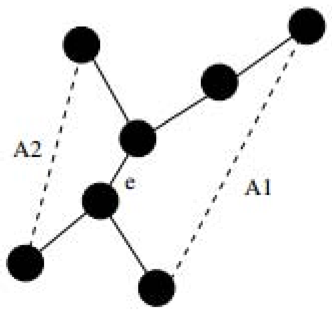

Figure 4.1: A graph and its spanning tree - obtained by deleting the dashed edges.

4.4 Graphs as SDD linear sytems

In this Section we discuss how SDD linear systems can be viewed entirely as graphs. Combinatorial preconditioning advocates a principled approach to the solution of linear systems. The core of CMG and all other solvers designed in the context of combinatorial preconditioning is in fact a solver for a special class of matrices, graph Laplacians. The *Laplacian* A of a graph $G = (V, E, w)$ with *positive weights*, is defined by:

$$A_{i,j} = A_{j,i} \ - w_{i,j} \quad and \quad A_{i,i} = -\sum_{i \neq j} A_{i,j}.$$

More general systems are solved via light-weight transformations to Laplacians. Consider for example the case where the matrix A has a number of positive off-diagonal entries, and the property $A_{i,i} = \sum_{i \neq j} |A_{i,j}|$. Positive off-diagonal entries have been a source of confusion for algebraic multigrid (AMG) solvers, and various heuristics have been proposed. Instead, CMG uses a reduction known as double-cover [10]. Let $A = A_p + A_n + D$, where D is the diagonal of A an A_p is the matrix consisting only of the positive off-diagonal entries of A. It is easy to verify that

$$Ax = b \Leftrightarrow \begin{pmatrix} D + A_n & -A_p \\ -A_p & D + A_n \end{pmatrix} \begin{pmatrix} x \\ -x \end{pmatrix} = \begin{pmatrix} b \\ -b \end{pmatrix}$$

In this way, we reduce the original system to a Laplacian system, while at most doubling the size. In practice it is possible to exploit the obvious symmetries of the new system, to solve it with an even smaller space and time overhead.

Matrices of the form $A + D_e$, where A is a Laplacian and D_e is a positive diagonal matrix have also been addressed in various ways by different AMG implementations. In CMG, we again reduce the system to a Laplacian. If d_e is the vector of the diagonal elements of D, we have

$$Ax = b \Leftrightarrow \begin{pmatrix} A + D_e & 0 & -d_e \\ 0 & A + D_e & -d_e \\ -d_e^T & -d_e^T & \sum_i d_e(i) \end{pmatrix} \begin{pmatrix} x \\ -x \\ 0 \end{pmatrix} = \begin{pmatrix} b \\ -b \\ 0 \end{pmatrix}$$

Again it's possible to implement the reduction in a way that exploits the symmetry of the new system, and with a small space and time overhead work only implicitly with the new system.

A symmetric Matrix A is diagonally dominant (SDD), if $A_{i,i} \geq \sum_{i \neq j} |A_{i,j}|$. The two reductions above can reduce any SDD linear system to a Laplacian system. Symmetric positive definite matrices with non-positive off-diagonals are known as M-matrices. It is well known that if A is an M-matrix, there is a positive diagonal matrix D such that $A = DLD$ where L is a Laplacian. Assuming D is known, an M-system can also be reduced to a Laplacian system via a simple change of variables. In many application D is given, or it can be recovered with some additional work [4].

The reduction of SDD systems to Laplacians allows us to concentrate on them for the rest of the paper. There is a one-to-one correspondence between Laplacians and graphs, so we will be often using the terms interchangeably.

4.5 Graph Partitioning

Partitioning weighted graphs into disjoint and dissimilar clusters of similar vertices is arguably one of the most important algorithmic problems. Naturally, in applications, one is interested in obtaining good clusterings with as few clusters as possible, i.e. with a large ***reduction factor*** p, defined as the number of vertices in the given graph over the number of clusters.

Let $G = (V, E, w)$ be a weighted graph. The Laplacian of G is the matrix A_G defined by $A_{ii} = -w_{ij}$ and $A_{ii} = \sum_{j \neq i} A_{ij}$. If $G_1 = (V, E, w_1)$, $G_2 = (V, E, w_2)$ and $G = (V, E, w_1 + w_2)$, we have $A_G = A_{G1} + A_{G2}$. We will often identify graphs with their Laplacians using this natural one-to-one correspondence. The total incident weight $\sum_{u \in N(v)} w(u, v)$ of vertex v is denoted by $vol(v)$. For any $V' \subseteq V$ we let $vol(V') = \sum_{v \in V'} vol(v)$, and $out(V') = \sum_{v \in V', u \ni V'} w(u, v)$. We also let

$$cap(U, V) = \sum_{u \in U, v \in V} w(u, v)$$

denote the total weight connecting the nodes of the disjoints set U, V. The ***sparsity*** of an edge cut into V' and $V - V'$ is defined as the ratio

$$\frac{cap(V, V - V')}{min(vol(V'), vol(V - V'))},$$

4.6 Steiner preconditioners

This section requires some well known definitions and facts from the support theory for preconditioning.

Definition 4.1. [Support and condition numbers]
The support $\sigma(A,B)$ of two Laplacians (A,B) is defined as

$$\sigma(A,B) = min\{t \in \mathbb{R} : x^t(\tau B - A)x \geq 0, \forall x, \forall \tau \geq t\}.$$

The condition number is defined as

$$\kappa(A,B) = \sigma_{max}(A,B)\sigma(B,A).$$

Definition 4.2. [Generalized eigenvalues]
The set of generalized eigenvalues $\Lambda(A,B)$ of pair Laplacians is defined by

$$\Lambda(A,B) = \{\lambda : \text{there is real vector } x \text{ such that} Ax = \lambda Bx\}.$$

Lemma 4.3. [Rayleigh quotient characterization of support]
if A, B have the same size, we have

$$\lambda_{max}(A,B) = \sigma(A,B) = max_{x^T j \neq j}(x^T Ax)/(x^T Bx),$$

where j denotes the constant vector.

Definition 4.4. [Schur complement]
Let T be a weighted star graph with $n+1$ vertices and edge weights $d_1, ..., d_n$. The Scur complement $S(T,v)$ of T with respect to its root v, is the graph defined by the weights $S_{ij}(T,v) = d_i d_j / D$ where $D = \sum_i d_i$. Let A be any graph, $A[V-v]$ be the graph induced in A by the vertices in $V-v$, and T_u be the star graph consisting of the edges incident to v in A. The Scur complement $S(A,v)$ of A with respect to vertex v is the graph $A[V-v] + S(T_v, v)$. Let $W \subset V$ and v be any vertex in W. The Scur complement with $S(A,W)$ is recursively defined as

$$S(A,W) = S(S(A,v), W-v) = S(S(A,W-v),v).$$

Let A, B be positive definite matrices. We let $\lambda_i \leq ... \leq \lambda_n$ denote the eigenvalues of A and $\mu_1 \leq ... \leq \mu_n$ denote the eigenvalues of B. Let κ_{max} and κ_{min} denote $\lambda_{max}(A,B)$ and $\lambda_{min}(A,B)$. We therefore have $\lambda_{max}(B,A) = 1/\kappa_{min}$ and $\lambda_{min}(B,A) = 1/\kappa_{max}$.

Steiner preconditioners, introduced in [10] and then extended in [15], introduce external nodes into preconditioners. The steiner preconditioner is based on a partitioning of the n vertices in V into m vertex-disjoint clusters V_i. So, for each V_i, the preconditioner contains a star graph S_i, with leaves corresponding to the vertices in V_i rooted at vertex r_i. The roots r_i are connected and form the **quotient** graph Q. This general setting is illustrated in Figure 3.1(b).

Let D' be the total degree of the leaves in the Steiner preconditioner S. Let the **restriction** R be an nxm matrix, where $R(i,j) = 1$ if vertex i is in cluster j and 0 otherwise. Then, the Laplacian os S has $n+m$ vertices, and the algebraic form

$$S = \begin{pmatrix} D' & -D'R \\ -R^T D' & Q + R^T D' R \end{pmatrix}, \tag{4.1}$$

A concerned feature of the Steiner preconditioner S is the extra number of dimensions/vertices. Gremban and Miller [10] that every time a system of the form $Bz = y$ is solved in an usual preconditioned method, the system

$$S \begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} y \\ 0 \end{pmatrix}$$

should be solved instead, for a set of don't care variables z'. They also showed that the operation is equivalent to preconditioning with the dense matrix

$$B = D' - V(Q + D_Q)^{-1} V^T, \tag{4.2}$$

where $V = D'R$, and $D_Q = R^T D' R$. The matrix B is called the Schur complement of S with respect to the elimination of the roots r_i. It is a well known fact that B is also a Laplacian.

The analysis of the support σ(A/S), is identical to that for the case of subgraph preconditioners. For instance, going to Figure 4.2, the edge (v_1, v_4) can only be supported by the path (v_1, r_1, v_4), and the edge (v_4, v_7) only by the path (v_4, r_1, r_2, v_7). Similarly we can see the mappings from edges in A to paths in S for every edge in A. In the example, the **dilation** of the mapping 3, it can be seen that to minimize the **conjestion** on every edge of S (i.e. make it equal to 1), we need to take $D' = D$, where D are the total degrees of the nodes in A, and $w(r_1, r_2) = w(v_3, v_5) + w(v_4, v_7)$. More generally, for two roots r_i, r_j we should have

$$W(r_i, r_j) = \sum_{i' \in V_i, j' \in V_j} w_{i,j}.$$

Under this construction, the algebraic form of the quotient Q can be seen to be $Q = R^T A R$.

In [15] it was shown that the support $\sigma(S/A)$ reduces to bounding the support of $\sigma(S_i, A[V_i])$, for all i, where $A[V_i]$ denotes the graph induced in A by the vertices V_i.

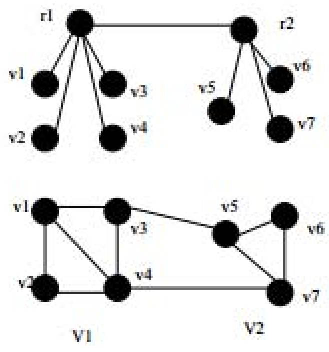

Figure 4.2: A graph and its Steiner preconditioner.

4.7 Graph conductance

The conductance $\phi(A)$ of a graph $A = (V, E, w)$ is defined as

$$\phi(A) = \min_{S \subseteq V} \frac{w(S, V - s)}{\min(w(S), w(V - S))}$$

where $w(S, V - S)$ denotes the total weight connecting the sets S and $V - S$, and where $w(S)$ denotes the total weight incident to the vertices in S.

The main result of [15] is captured by the following Theorem.

Theorem 4.1 The support $\sigma(S/A)$ is bounded by a constant c independent from n, if and only if for all i the conductance of the graph $A^o[V_i]$ induced by the nodes in V_i augmented by the edges leaving V_i is bounded by a constant c'.

Although Theorem doesn't give a way to pick to clusters, it does provide a way to avoid bad clusterings.

4.8 Support theory and Grady's clusterings

In hir recent work [9], Grady proposed a multigrid method where the construction of the "coarse" grid follows exactly the construction of the **quotient** graph in the previous section. Specifically, Grady proposes a clustering such that every cluster contains exactly one of certain pre-specified "coarse" nodes. He then defines the restriction matrix R and he lets the coarse grid be $Q = R^T AR$, identically to the construction of the previous Section. The question then is whether the proposed clustering provides the guarantees that by the previous Theorem are necessary to construct a good Steiner preconditioner. In the following Figure, we replicate Figure 3.2 of [9], with a choice of weights that force the depicted clustering.

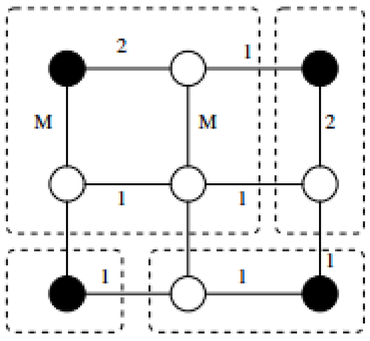

Figure 4.3: A bad clustering

Every cluster in Figure 4.3 contains exactly one black/coarse node. The problem with the clustering is that the top left cluster, has a very low conductance when M » 1. In general, in order to satisfy the requirement of the previous Theorem, there are cases where the clustering has to contain clusters with no coarse nodes in them. As we will discuss in later the behavior of the multigrid algorithm proposed in [9] is closely related to the quality of the Steiner preconditioner induced by the clustering. This implies that the multigrid of [20] can suffer bad convergence.

The canonical clustering in Grady's algorithm is very suitable for GPU implementations, when other solvers may be less suitable. This gives to it an advantage on this type of hardware. Even in the presence of a number of relatively bad clusters, it can be faster relative to a solver that uses better clusters. However the advantage is lost when the computed clusters cross a negative threshold in quality, a threshold that depends on several hardware-dependent factors. The value of Support Theory is evident in this case. Grady's algorithm can be instrumented with a very fast routine that measures the quality of the formed clusters and predicts its performance, and reverts to another solver when needed. One can also imagine hybrid clustering algorithms where the majority clusters are formed using the algorithm [9] and the 'sensitive' parts of the system are treated seperately.

Chapter 5

The Combinatorial Multigrid Solver

In this chapter we decribe the Combinatorial Multigrid Solver (CMG) which for a given nxn matrix A with m non-zero elements, has a time complexity $O(mlogn)$. We start with a short review of multigrid algorithms and other SDD solvers, which we think is necessary to explain why CMG is different from previous implementations of multigrid algorithms.

5.1 Related work on SDD solvers

Multigrid was firstly conceived as a method to solve linear systems that generated by the discretization of the Laplace equation over relatively nice domains [21]. The geometry that lies under those domains leads to a hierarchy of grids $A = A_0, ..., A_d$, that look similar ata different levels of detail; someone can think of multigrid as a tower od 2D grids, with sizes $2^{d-i}x2^{d-i}$ for $i = 0, ..., d$. Its provably asymptotically optimal behavior for certain classes of problems soon lead to an effort -known as Algebraic Multigrid (AMG)- to generalize its principles to arbitrary matrices. In contrast to classical Geometric Multigrid (GMG) where the hierarchy of grids is generated by the discretization process, AMG constructs the hierarchy of 'coarse' grids/matrices based only on the algebraic information contained in the matrix. Various flavors of AMG –based on different heuristic coarsening strategies– have been proposed in the literature. AMG has been proven successful in solving more problems than GMG, though some times at the expense of robustness, a by-product of the limited theoretical understanding.

A solver with provable properties for arbitrary SDD matrices, was discovered only recently. The path to it was Support Theory [3], a set of mathematical tools developed for the study of combinatorial subgraph preconditioners, originally introduced by Vaidya [[22], [11]]. It has been at the heart of the seminal work of Spielman and Teng [20] who proved that SDD systems can be solved in nearly-linear time. Koutis and and Miller [14] proved that SDD matrices with planar connection topologies can be solved asymptotically optimally, in $O(n)$ time for n-dimensional matrices. The complexity of the Spielman and Teng solver was recently significantly improved by Koutis, Miller and Peng [[16], [13]], who described an $O(mlogn)$ algorithm for the solution of general SDD systems with m non-zero entries.

These theoretically described solvers have to deal with large hidden constants and with the complicated nature of the underlying algorithms, which makes them impractical. Combinatorial Multigrid (CMG) is a variant of multigrid that reconciles theory with practice. CMG also builds a hierarchy of matrices/graphs as AMG does. But the main and essential difference from AMG is that the hierarchy is constructes after viewing the matrix as a graph, and using the discrete geometry of the graph, for example notions like graph separators and expansion. The re-introduction of geometry into the problem allows us to prove sufficient and necessary conditions for the construction of a good hierarchy and claim strong convergence guarantees for symmetric diagonally dominant (SDD) matrices based on recent progress in Steiner preconditioning. [[10], [12], [15]].

5.2 A graph decomposition algorithm

The crusial step of constructing a good Steiner preconditioner is the computation of a group decomposition that satisfies, as best as possible, the requirements of Theorem 1. Before the presentation of the **DecomposeGraph** algorithm, that extends the ideas of [15], we need to introduce a couple of definitions. Let $volG(v)$ denote the total weight incident to node v in graph G. The weighted degree of a vertex v is defined as the ratio

$$wd(v) = \frac{vol(v)}{\max_{u \in N(v)} w(u,v)}$$

The average weighted degree of the graph is defines as $awd(G) = (1/n)\sum_{y \in N} wd(v)$.

Algorithm 7 Algorithm Decompose Graph.

```
1: Algorithm **Decompose Graph**
2:
3: Input: Graph $A = (V, E, w)$
4: Output: Disjoint Clusters $V_i$ with $V = \bigcup_i V_i$
5: Let $W \subseteq V$ be the set of nodes satisfying $wd(v) > \kappa \cdot awd(A)$, for some constant $\kappa > 4$.
6: Form a forest graph $F$, by keeping the heaviest incident edge of $v$ for each vertex $v \in V$ in $A$.
7: For every vertex $w \in W$ such that $vol_T(w) < vol_G(w)/awd(A)$ remove from $F$ the edge contributed by $w$ in Step 2.
8: Decompose each tree $T$ in $F$ into vertex-disjoint trees of constant conductance.
```

It is not very difficult to prove that the algorithm Decompose-Graph produces a partitioning where the conductance of each cluster depends only on $awd(A)$ and the constant κ. In fairly general topologies that allow high degree nodes, $awd(A)$ is constant and the number of clusters m returned by the algorithm is such that $n = m > 2$ (and in practice larger than 3 or 4). There are many easy ways to implement Step 3. Our current implementation makes about three passes of A. Of course, one can imagine variations of the algorithm (i.e. a correction step, etc) that may make the clustering phase a little more expensive with the goal of getting a better conductance and an improved condition number, if the application at hand requires many iterations of the solver.

5.3 From Steiner preconditioners to Multigrid

Algebraically, any of the classic preconditioned iterative methods, such as the Jacobi and Gauss-Seidel iteration, is nothing but a matrix S, which gets applied implicitly to the current error vector e, to produce a new error vector $e' = Se$. For example, in the Jacobi iteration we have $S = (I - D^1 A)$. This has the effect that it reduces effectively only part of the error in a given iterate, namely the components that lie in the low eigenspaces of S (usually referred to as high frequencies of A). The main idea behind a two-level multigrid is that the current smooth residual error $r = b - Ax$, can be used to calculate a correction $P^T Q^{-1} Pr$, where Q is a smaller graph and P is an mxn restriction operator. The correction is then added to the iterate x. The hope here is that for smooth residuals, the low-rank matrix $P^T Q^{-1} P$ is a good approximation of A^{-1}. Algebraically, this correction is the application of the operator $T = (I - P^T Q^{-1} PA)$ to the error vector e. The choice of P and Q is such that T is a projection operator with respect to the A-inner product, a construction known as the *Galerkin* condition. Twolevel convergece proofs are then based on bounds on the angle between the subspace $Null(P)$ and the high frequency subspace of S.

At a high level, the key idea behind CMG is that the provably small condition number $\kappa(A, B)$, is equal to the condition number $\kappa(\hat{A}, \hat{B})$ where $\hat{A} = D^{-1/2}AD^{-1/2}$ and $\hat{B} = D^{-1/2}BD^{-1/2}$. This in turn implies a bound on the angle betwqeen the low frequency of $\hat{A}$ and the high frequency of $\hat{B}$. The latter subspace includes $Null(R^T D^{1/2})$. THis fact suggests to choose $R^T D^{1/2}$ as the projection operator with performing relaxation with $(I - \hat{A})$ on the system $\hat{A}y = D^{-1/2}b$, with $y = D^{1/2}x$. Combining everything we get the following two-level algorithm.

Algorithm 8 Two-level Combinatorial Multigrid.

```
1: **Two-level Combinatorial Multigrid**
2:
3: **Input**: Laplacian $A = (V, E, w)$, vector $b$, approximate solution $x$, $nxm$ restriction matrix $R$
4: **Output**: Updated solution $x$ for $Ax = b$
5: $D := diag(A); \hat{A} := D^{-1/2}AD^{-1/2}$;
6: $z := (I - \hat{A})D^{1/2}x + D^{-1/2}b$;
7: $r := D^{-1/2}b - \hat{A}z; w := R^T D1/2r$;
8: $Q := R^T AR$; Solve $Qy = w$.
9: $x := D^{-1/2}((I - \hat{A})z + D^{-1/2}b)z$;
```

The two-level algorithm can anturally be extended into a full multigrid algorithm, by recursively calling the algorithm when the solution to the system with Q is requested. This produces a hierarchy of graphs $A = A_0, ..., A_d$. The full multigrid algorithm we use, after simplifications in the algebra of the two-level scheme is as follows.

Algorithm 9 Full Multigrid Algorithm.

```
1: **function** $x := CMG(A_i, b_i)$
2: $D := diag(A)$
3: $x := D^{-1}b$
4: $r_i := b_i - A_i(D^{-1}b)$
5: $b_{i+1} := Rr_i$
6: $z := CMG(A_{i+1}, b_{i+1})$
7: **for** $i = 1 \to t_i - 1$ **do**
8:     $r_{i+1} := b_{i+1} - A_{i+i}z$
9:     $z := z + CMG(A_{i+1}, r_{i+1})$
10: **end for**
11: $x := x + R^T z$
12: $x := r_i - D^{-1}(A_i x - b)$
```

If $nnz(A)$ denotes the number of non-zero entries in matrix A, we pick

$$t_i = \max\{\lceil \frac{nnz(A_i)}{nnz(A_{i+1})} \rceil, 1\}$$

This choice for the number of recursive calls, combined with the fast geometric decrease of the matrix sizes, targets a geometric decrease in the total work per level, while optimizing the condition number.

Chapter 6

Results and conclusions

Our implementation was written in C and compiled with the Intel Parallel Studio XE 2013 suite. It relies on standard C Libraries, Csparse Library and Intel Math Kernel Library. CMG consists of two phases. The setup up phase where a process of the initial matrix takes place and the hierarchy-preconditioner is prepared, and the solution phase where Conjugate Gradient method uses the preconditioner to solve the system of equations. Our benchmark runs were performed on an Intel Core i7 at 2.4GHz, with 6 cores and 24GB main memory. All benchmarks were solved for a specific b-side, and the *stopping criterion* for convergence was taken to be $\|Ax - b\| \leq 1e - 04 * \|b\|$. In this chapter is made a comparative analysis between CMG preconditioner and Jacobi preconditioner.

The table below shows the results of the benchmarks where an initial guess of the solution x (where $x \neq 0$) is given.

Matrix	Dimensions	Non-zero elements	Hierarchy time (sec)	CMG iterations	CMG time (sec)	Jacobi iterations	Jacobi time (sec)
ad1	210904	2112590	0.66	9	0.30	171	1.59
ad3	450927	4191415	1.01	10	0.75	202	3.92
ad4	494716	4076449	1.15	10	0.73	219	4.32
bb2	534782	4407059	1.13	9	0.70	196	4.09
bb4	2169183	19437167	5.48	10	3.68	258	25.41
nb6	1248150	11591932	2.97	12	2.41	187	10.24
nb7	2481272	21370078	4.81	9	3.74	237	26.86

Table 6.1: Summary of the results for the given sets (matrix,initial guess $x \neq 0$,right hand side b), including time performance and number of iterations of the CG method in both cases.

In the following table is given the reduction of the iterations that comes from the CMG preconditioner and the execution time speedup.

Matrix	Iteration Reduction	Exec. Time Speedup	Exec. Time Speedup (w/o hierarchy)
ad1	19	1.65	5.30
ad3	20.2	2.22	5.22
ad4	21.9	2.29	5.91
bb2	21.77	2.23	5.84
bb4	25.8	2.77	6.90
nb6	15.58	1.90	5.24
nb7	26.33	3.14	7.18

Table 6.2: Iteration Reduction of CMG and Execution Speedup Time

The table below shows the results of the benchmarks where an initial guess of the solution x (where $x = 0$) is given.

Matrix	Dimensions	Non-zero elements	Hierarchy time (sec)	CMG iterations	CMG time (sec)	Jacobi iterations	Jacobi time (sec)
ad1	210904	2112590	0.66	9	0.27	216	1.79
ad3	450927	4191415	1.01	9	0.61	223	4.27
ad4	494716	4076448	1.15	10	0.65	226	5.14
bb2	534782	4407058	1.13	10	0.71	220	4.50
bb4	2169183	19437167	5.48	10	3.62	307	29.97
nb6	1248150	11591932	2.97	12	2.40	269	14.65
nb7	2481272	21370078	4.81	10	4.16	257	29.41

Table 6.3: Summary of the results for the given sets (matrix,initial guess $x = 0$,right hand side b), including time performance and number of iterations of the CG method in both cases.

In the following table is given the reduction of the iterations that comes from the CMG preconditioner and the execution time speedup.

Matrix	Iteration Reduction	Exec. Time Speedup	Exec. Time Speedup (w/o hierarchy)
ad1	24	1.94	6.62
ad3	24.77	2.63	7
ad4	26.6	2.85	7.90
bb2	22	2.44	6.33
bb4	30.7	3.29	8.27
nb6	22.41	2.72	6.10
nb7	25.7	3.27	7.06

Table 6.4: Iteration Reduction of CMG and Execution Speedup Time

Bibliography

[1] R. Barrett. ***Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods***. Miscellaneous Titles in Applied Mathematics Series No 43. Society for Industrial and Applied Mathematics, 1994.

[2] M. Bern, J. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support-graph preconditioners.

[3] Erik G. Boman and Bruce Hendrickson. Support Theory for Preconditioning. ***SIAM J. Matrix Anal. Appl.***, 25(3):694–717, 2003.

[4] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. ***CoRR***, abs/0803.0988, 2008.

[5] T.A. Davis. ***Direct Methods for Sparse Linear Systems***. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2006.

[6] Timothy Davis. **CSPARSE**: a concise sparse matrix package. http://people.sc.fsu.edu/ jburkardt/c_src/csparse/csparse.html.

[7] Carnegie-Mellon University. Computer Science Dept and K.D. Gremban. ***Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems***. Research paper. School of Computer Science, Carnegie Mellon University, 1996.

[8] Peter G. Doyle and J. Laurie Snell. Random Walks and Electric Networks, January 2000.

[9] Leo Grady. A Lattice-Preserving Multigrid Method for Solving the Inhomogeneous Poisson Equations Used in Image Analysis. In David Forsyth, Philip Torr, and Andrew Zisserman, editors, ***Computer Vision – ECCV 2008***, volume 5303 of ***Lecture Notes in Computer Science***, chapter 19, pages 252–264. Springer-Verlag, Berlin, Heidelberg, 2008.

[10] Keith Gremban. ***Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems***. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123.

[11] Anil Joshi. ***Topics in optimization and sparse linear systems***. PhD thesis, Champaign, IL, USA, 1997. UMI Order No. GAX97-17289.

[12] Ioannis Koutis. Combinatorial and algebraic tools for optimal multilevel algorithms, 2007.

[13] Ioannis Koutis, Gary Miller, and Richard Peng. Solving sdd linear systems in time $\tilde{O}(m \log n \log(1/\epsilon))$. April 2011.

[14] Ioannis Koutis and Gary L. Miller. A linear work, $O(n^{1/6})$ time parallel algorithm for solving planar Laplacians. In ***Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms***, SODA '07, pages 1002–1011, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[15] Ioannis Koutis and Gary L. Miller. Graph partitioning into isolated, high conductance clusters: theory, computation and applications to preconditioning. In ***Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures***, SPAA '08, pages 137–145, New York, NY, USA, 2008. ACM.

[16] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD systems. August 2010.

[17] Ioannis Koutis, Gary L. Miller, and David Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. ***Computer Vision and Image Understanding***, 115(12):1638–1646, 2011.

[18] Yiannis Koutis. Matlab implementation of the combinatorial multigrid algorithm. http://www.cs.cmu.edu/ jkoutis/cmg.html.

[19] Y. Saad. ***Iterative Methods for Sparse Linear Systems***. Society for Industrial and Applied Mathematics, 2003.

[20] Daniel A. Spielman and Shang-Hua Teng. Nearly-Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems. May 2007.

[21] Ulrich Trottenberg, Cornelius W. Oosterlee, and Anton Schuller. ***Multigrid***. Academic Press, 1st edition, December 2000.

[22] Preadeep M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. a talk based on this manuscript was presented at the ima workshop on graph theory and sparse matrix computation. October 1991.