Domain Decomposition for Two-Dimensional Elliptic Operators on Vector and Parallel Machines\textsuperscript{1}

by

Ruth Gonzalez

Technical Report 86-4, April 1986

\textsuperscript{1}A Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy, Rice University.
Domain Decomposition for Two-Dimensional
Elliptic Operators on Vector and Parallel Machines
Ruth Gonzalez
Rice University, 1986

Abstract

The efficient computation of the solution to self-adjoint elliptic operators is the subject of this dissertation. Discretization of this equation by finite differences or finite elements yields a large, sparse, symmetric system of equations, $Ax = b$. We use the preconditioned conjugate gradient method with domain decomposition to develop an effective, vectorizable
preconditioner which is suitable for solving large two-dimensional problems on vector and parallel machines.

The convergence of the preconditioned conjugate gradient method is determined by the condition number of the matrix $M^{-1}A$ where $A$ and $M$ correspond to the matrix for the discretized differential equation and to the preconditioning matrix, respectively. By appropriately preconditioning the system $Ax = b$ we can significantly reduce the computational effort that is required in solving for $x$. 
The basic approach in domain decomposition techniques is to break up the domain of integration into many pieces, solve the appropriate equation on each piece, then somehow construct the global solution from these local solutions. In this dissertation we formulate an effective preconditioner for two-dimensional elliptic partial differential equations using this notion of domain decomposition. We demonstrate that this method is efficient in its vectorized form and present numerical results to support this conclusion.

Acknowledgements

It is a pleasure to express my thanks to several people for their assistance:

To Drs. James Bramble, Joseph Pasciak, and Alfred Schatz for their personal attention and useful comments regarding their method and my research;

To Tom Hewitt for optimizing my code for the CRAY-XMP and for running the XMP examples;

To Dr. Franklyn K. Levin of Exxon Production Research Company for his continued encouragement during my studies;

To Professors William Symes and John Ed Akin for serving on my thesis committee;

And especially to Professor Mary P. Wheeler for serving as my advisor and for her advice and guidance in this research.
TABLE OF CONTENTS

Abstract .......................................... ii
Acknowledgements ................................. iii
List of Figures ................................... v
List of Tables ................................... vi
I. Introduction ................................... 1
II. Discussion of Previous Work ................. 9
   2.1. Capacitance Matrices ..................... 10
   2.2. Concus, Golub, and Meurant
        Preconditioners .......................... 14
   2.3. Meurant Preconditioners ................ 18
   2.4. Bramble, Pasciak, and Schatz
        Preconditioners .......................... 21
   2.5. Glowinski, Periaux, and Fischler
        Preconditioners .......................... 26
III. Description of the Numerical Procedure .... 31
   3.1. The Dirichlet Problem .................... 32
   3.2. The Neumann Problem .................... 42
   3.3. Computational Aspects .................. 46
IV. Numerical Results ........................... 48
V. Conclusions .................................. 77
References ..................................... 79

LIST OF FIGURES

1.1 Natural Ordering of Unknowns ............... 3
2.1 Subdivided T-shaped Domain for
        Capacitance Matrix ...................... 11
2.2 Subdivision of Domain for Bramble, et al. 23
2.3 Subdivision of Domain for Glowinski, et al. 27
3.1 Mesh for Rectangular Domain ................ 34
3.2 Ordering of Subdomains, Edges, and Vertices 35
3.3 Uniform Grids for the Neumann Problem .... 44
3.4 Non-uniform Grids for the Neumann Problem . 45
4.1 Iterations and CPU versus the Number of
        Subregions for $a_4(x,y)$ with $Nx = Ny = 33$. 62
4.2 Iterations and CPU versus the Number of
        Unknowns for $a_4(x,y)$ with
        $Nx = Ny = 4$ and $8$ ..................... 71
4.3 Iterations and CPU versus the Number of
        Unknowns for the Best Cases for $a_4(x,y)$ .. 72
LIST OF TABLES

4.1 Iteration and CPU Statistics for 
\[ a(x,y) = a_1(x,y) \] .......................... 50

4.2 Iteration, CPU, and MFLOPS Statistics for 
\[ a(x,y) = a_1(x,y) \] on the CRAY-XMP .......... 52

4.3 Iteration and CPU Statistics for 
\[ a(x,y) = a_2(x,y) \] .......................... 54

4.4 Iteration and CPU Statistics for 
\[ a(x,y) = a_2(x,y) \] for \( N_x = N_y = 101 \) ......... 55

4.5 Iteration and CPU Statistics for 
\[ a(x,y) = a_3(x,y) \] .......................... 56

4.6 Iteration and CPU Statistics for 
\[ a(x,y) = a_4(x,y) \] .......................... 57

4.7 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 33 \] and \( \bar{N}_x = \bar{N}_y = 2 \) .......... 59

4.8 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 33 \] and \( \bar{N}_x = \bar{N}_y = 4 \) .......... 60

4.9 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 33 \] and \( \bar{N}_x = \bar{N}_y = 8 \) .......... 61

4.10 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 65 \] and \( \bar{N}_x = \bar{N}_y = 4 \) .......... 63

4.11 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 65 \] and \( \bar{N}_x = \bar{N}_y = 8 \) .......... 64

4.12 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 65 \] and \( \bar{N}_x = \bar{N}_y = 16 \) .......... 65

4.13 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 129 \] and \( \bar{N}_x = \bar{N}_y = 4 \) .......... 66

4.14 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 129 \] and \( \bar{N}_x = \bar{N}_y = 8 \) .......... 67

4.15 CPU Statistics for \[ a(x,y) = a_4(x,y) \] with 
\[ N_x = N_y = 129 \] and \( \bar{N}_x = \bar{N}_y = 16 \) .......... 68

4.16 Iteration and CPU Statistics for 
\[ a(x,y) = a_4(x,y) \] for a Fixed Number of 
Subregions .......................................................... 70

4.17 Iteration and CPU Statistics for 
\[ a(x,y) = a_4(x,y) \] for Non-Uniform Mesh. .... 74

4.18 Iteration and CPU Statistics for 
\[ a(x,y) = a_5(x,y) \] .......................... 76
CHAPTER I

INTRODUCTION

The efficient computation of the solution to a two-
dimensional elliptic partial differential equation with
either Dirichlet or Neumann boundary conditions is the
subject of this dissertation. Discretization of these
equations by finite differences or finite elements yields
a large, sparse, and symmetric system of equations. We use
the preconditioned conjugate gradient method (PCG) with do-
main decomposition to develop an efficient, robust, and
vectorizable preconditioner which is suitable for two-
dimensional problems on vector machines.

For simplicity, we consider the following problem

\[ -\nabla \cdot (a(x,y)\nabla u(x,y)) = f(x,y) \quad \text{on } \Omega \]

with Dirichlet boundary conditions

\[ u(x,y) = g(x,y) \quad \text{on } \partial \Omega \]

or Neumann boundary conditions

\[ -a(x,y)\nabla u(x,y) \cdot \nu = 0 \quad \text{on } \partial \Omega, \]

where \( \Omega \) is the rectangle \([0,x_L] \times [0,y_L] \), \( \partial \Omega \) is the
boundary of \( \Omega \), \( 0 < a_0 < a(x,y) < a_1 \) and \( \nu \) is the out-
ward normal vector on \( \partial \Omega \). If boundary conditions (1.3) are
assumed then \( \int_{\partial \Omega} f = 0 \) is required.

Discretization of equations (1.1) and (1.2) by finite
differences or Galerkin finite elements yields a large,
symmetric block tridiagonal system of equations. Suppose
that the domain \( \Omega \) is divided into \( Nx+1 \) intervals in the
x-direction and \( Ny+1 \) intervals in the y-direction. Fur-
ther assume that the natural ordering is used to order the
unknowns, i.e. order from the left end boundary of the
rectangle \( \Omega \) to its right end boundary, starting from the
bottom boundary then to the upper boundary (see figure 1.1).
Apply the point-centered finite differences to (1.1) and
(1.2) to obtain

\[ \frac{1}{a_{1+j,k} (x_{1+j} - x_k)} \left( u_{1+j,k+1} - u_{1+j,k} \right) - \frac{1}{a_{1+k,j} (y_{1+k} - y_j)} \left( u_{1+k,j+1} - u_{1+k,j} \right) \]

\[ = f_{1,1} \]
The point $[x_i, y_j]$ is the corner of the $i$th block in the $x$-direction and the $j$th block in the $y$-direction. We define the coefficients by harmonic averaging:

$$a_{i,j} = \frac{1}{a_{i-1,j} + \frac{1}{a_{i,j+1}}},$$

$$a_{i,j+1} = \frac{1}{a_{i,j} + \frac{1}{a_{i+1,j+1}}},$$

$$a_{i+1,j} = \frac{1}{a_{i,j} + \frac{1}{a_{i+1,j}}},$$

$$a_{i,j-1} = \frac{1}{a_{i,j} + \frac{1}{a_{i,j+1}}},$$

$$a_{i+1,j-1} = \frac{1}{a_{i,j} + \frac{1}{a_{i+1,j-1}}}.$$

The block tridiagonal matrix is composed of $N_{x}$ by $N_{y}$ symmetric matrix blocks with condition number $O(N_{x}N_{y})$. Therefore, the selected method must take advantage of the sparseness and the natural ordering of unknowns.

**Figure 1.1 Natural Ordering Of Unknowns**

![Diagram of natural ordering of unknowns](image)
symmetry of the problem as well as minimizing the computational effort that is needed for solving the system of equations.

The preconditioned conjugate gradient method [7] is specifically formulated to efficiently solve the algebraic system \( Ax = b \) resulting from the above discretization. The PCG method is an iterative algorithm in which the following steps are computed at each iteration:

\[
\begin{align*}
  &k > 1, \\
  &Mz^k = z^k, \\
  &b^k = (Mz^k, z^k) / (Mz^{k-1}, z^{k-1}), \\
  &p^k = z^k + b^k p^{k-1}, \\
  &a_k = (Mz^k, z^k) / (Ap^k, p^k), \\
  &x^{k+1} = x^k + a_k p^k, \\
  &r^{k+1} = r^k - a_k Ap^k,
\end{align*}
\]

(1.5)

where \( x^0 \) is given, \( r^0 = b - Ax^0, p^0 \) is arbitrary, and \( b_0 = 0 \). The convergence of the PCG method is determined both by the clustering of the eigenvalues and the condition number of \( M^{-1}A \), and thus critically depends on the selection of \( M \) in (1.5). Good preconditioners \( M \) are symmetric and positive definite and significantly reduce the condition number of the system, are less expensive in solving \( Mz^k = r^k \) rather than \( Ax = b \), and do not significantly increase the amount of storage relative to the storage needed to solve \( Ax = b \). By appropriately preconditioning the system we greatly reduce the amount of work expended in the computation of the solution \( x \).

Since each computer has different architectural features, careful consideration must be given to the algorithm selection for a given machine [19]. More arithmetic computations do not necessarily imply higher cpu costs if they are ordered in such a way as to take advantage of the vector and parallel capabilities of the particular machine. For example, the following loop does not vectorize:

\[
\begin{align*}
  &DO 20 K = 1, M \\
  &\quad DO 10 I = 2, M \\
  &\quad \quad X(I,K) = R(I,K) - B(I,K) * X(I-1,K) \\
  &10 \quad CONTINUE \\
  &20 \quad CONTINUE
\end{align*}
\]

However, vectorization is achieved by merely interchanging the order of do loops 10 and 20. Of course, many times additional overhead is the price we pay for this extra bonus; the trick is to maximize the parallel computations while minimizing this overhead. We are interested in
formulating an algorithm which is optimal on computers such as the CRAY-1S and the CRAY-XMP. The 1S has one processor with up to four million 64 bit words of real memory, while the XMP has one, two, or four processors and as much as sixteen million words of memory. Also, each of the CRAY machines loads data onto eight vector registers of 64 words each. The clock period, or cycle time, for the 1S is 12.5 ns, while for the XMP it is 9.5 ns. Since each of the CRAYs has separate arithmetic units for addition, multiplication, and reciprocal approximation, the maximum attainable computing speed on the 1S is 160 MFLOPS (millions of floating-point operations per second) and 210 MFLOPS per processor on the XMP.

Our goal is to develop a good preconditioner by domain decomposition, or substructuring, methods for use on systems whose architectures are similar to that of the CRAY computers. The basic approach in domain decomposition techniques is to break up the domain of integration into many pieces, solve the corresponding equation on each piece, then somehow construct the global solution from these local solutions. Careful ordering of the required computations in order to effectively utilize the parallel architecture of the CRAY machines reduces the total execution time for an algorithm which has parallelism.

In Chapter II of this dissertation we review previous efforts in developing good preconditioners. In particular, we examine some preconditioners that are based on domain decomposition as well as those with vectorized counterparts. In Chapter III we formulate an effective preconditioner for the self-adjoint elliptic operator using this notion of domain decomposition. Additionally, we demonstrate how this method is adapted for efficient use on vector and parallel machines. In Chapter IV, numerical results are offered as evidence to support these claims. Finally, in Chapter V we present some concluding remarks about the two-dimensional case and suggest possible extensions for the three-dimensional case.
CHAPTER II

DISCUSSION OF PREVIOUS WORK

The construction of preconditioners for the PCG method is an intensely active area of research. In this chapter, we discuss some recent formulations of preconditioners which incorporate domain decomposition, or substructuring, and which address questions of vectorization and parallelism. Concus, Golub, and Meurant [8] developed the incomplete block Cholesky preconditioner: INV and MINV. Although neither one of these preconditioners vectorizes, their significance will become apparent when the vectorized preconditioners INV2P and INV4P by Meurant ([15], [16], [17]) are presented. Bramble, Pasciak, and Schatz([2], [3]), created a family of preconditioners that use domain decomposition but are fundamentally different from other such algorithms. Glowinski, Periaux [11], and Fischler [12] developed another preconditioner for use in fluid flow calculations. Golub and Mayers [13], as well as Dryja [10], describe their preconditioners in the framework of general block matrices which arise from the discretization of elliptic partial differential equations using finite differences.

2.1. Capacitance Matrices

For many years domain decomposition has been used in solving structural finite element problems. The goal of the investigators was not to increase parallelism in the solution algorithms but to decouple different physical parts of the problem. But now, with the increasing availability of vector computers, we seek to utilize capacitance matrices in order to increase parallelism in our solution algorithm. Consider the T-shaped domain shown in figure 2.1 with domain \( \Omega \) and boundary \( \partial \Omega \) such that \( \Omega = \Omega_1 \cup \Omega_2 \cup \Gamma \) and \( \Gamma = \partial \Omega_1 \cap \partial \Omega_2 \). The five-point operator applied to (1.1) and (1.2) for the T-shaped domain gives the block matrix equation

\[
\begin{bmatrix}
A_{11} & A_{13} \\
0 & A_{22} & A_{23} \\
A_{13}^T & A_{23}^T & A_{33}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
= \begin{bmatrix}
b_1 \\
b_2 \\
b_3
\end{bmatrix}
\]

(2.1)

where \( x_1, x_2, \) and \( x_3 \) contain the unknowns in subdomains \( \Omega_1, \Omega_2, \) and \( \Gamma, \) respectively. Performing block Gaussian elimination to form an upper triangular matrix, we obtain
The matrix $\tilde{A}_{33}$ is called the capacitance matrix or the Schur complement. Until recently, solving such problems was accomplished by direct solution of the subdomain problems, as well as solving for the capacitance matrix, via fast Poisson solvers ([4], [9], [20]). Most of the recent work has focused on solving the subdomain problems with some fast direct method, but then using preconditioned conjugate gradients to solve the capacitance matrix equation ([1], [5], [6], [10], [11], [12], [13], [23]). The condition number of the capacitance matrix increases linearly as the number of nodes on the interface $\Gamma$ increase. Therefore, it is important to formulate a preconditioner for the capacitance
matrix which reduces effectively the condition number of the pre-conditioned system.

Dryja [10] used as a pre-conditioner the matrix \( K^b \),

\[
K = \begin{bmatrix}
2 & -1 \\
-1 & 2 & -1 \\
-1 & 2 & -1 \\
-1 & 2 & -1
\end{bmatrix},
\]

the one-dimensional discrete Laplacian over the interfaces, and fast Fourier transforms to solve the subdomain problems. Golub and Mayers [13] formulated the capacitance matrix pre-conditioner \( M = \sqrt{4K + K^2} \) by solving the infinite domain Laplace problem with an infinite interface dividing two half-planes. Golub and Mayers determined that their pre-conditioner had a lower condition number estimate, and would therefore outperform, that of Dryja's.

2.2. Concus, Golub, and Neurant Preconditioners

Concus, Golub, and Neurant [8] used block preconditioning methods for solving the Dirichlet problem \(-\nabla \cdot (a \nabla \phi) = f \) on \( \Omega \) with \( \phi = g \) on \( \partial \Omega \). Use of the five-point operator for discretizing the differential equation gives the block tridiagonal matrix

\[
A = \begin{bmatrix}
D_1 & A_2^T & & & \\
A_2 & D_2 & \cdots & & \\
& \cdots & \cdots & \cdots & \\
& & \cdots & \cdots & A_N^T \\
& & & A_N & D_N
\end{bmatrix},
\]

where

\[
D_1 = \begin{bmatrix} a_1 & b_2 & & \\
& b_2 & a_2 & \cdots & \\
& & \cdots & \cdots & b_m \\
& & & b_m & a_m \end{bmatrix} \quad \text{and} \quad A_1 = \begin{bmatrix} c_1 \\
c_2 \\
\vdots \\
c_m \end{bmatrix}.
\]

Since \( A \) is symmetric and positive definite, \( A \) can be decomposed by

\[
A = D + L + L^T,
\]

where
Let $I$ be the symmetric block diagonal matrix with $m_i \times m_i$ blocks $I_i$ satisfying

\begin{align}
(2.6a) & \quad I_1 = D_1, \\
(2.6b) & \quad I_i = D_i - A_i E_i^{-1} A_i^T, \quad 2 < i < N.
\end{align}

Then the block Cholesky factorization of $A$ can be written as

\begin{equation}
A = (I + L) E^{-1} (I + L^T),
\end{equation}

where the factor $I + L$ is block lower bidiagonal. Note that

$I_1$ is tridiagonal, but each $I_i$, $i = 2, 3, \ldots, N$, is a full matrix. Instead of computing $E_i^{-1}$, Concus et al. substitute a sparse approximation $E_i^{-1}$. They then obtain the block diagonal matrix $\Delta$ with $m_i \times m_i$ blocks $\Delta_i$ satisfying

\begin{align}
(2.8a) & \quad \Delta_1 = D_1, \\
(2.8b) & \quad \Delta_i = D_i - A_i E_i^{-1} A_i^T, \quad 2 < i < N,
\end{align}

where for each $2 < i < N$, $E_i^{-1}$ is the sparse approximation to $A_i^{-1}$. The incomplete block Cholesky preconditioning matrix for use with the conjugate gradient method is

\begin{equation}
M = (\Delta + L) \Delta^{-1} (\Delta + L^T) - A + R,
\end{equation}

where $R$ is a block diagonal matrix

\begin{equation}
R = \begin{bmatrix} R_1 & & \\ & \ddots & \\ & & R_N \end{bmatrix}
\end{equation}

with

\begin{align}
(2.12a) & \quad R_1 = \Delta_1 - D_1 = 0, \\
(2.12b) & \quad R_i = \Delta_i - D_i + A_i E_i^{-1} A_i^T, \quad 2 < i < N.
\end{align}

The factor $\Delta + L$ is lower bidiagonal. Using the Cholesky factorization of $\Delta_i$,

\begin{equation}
\Delta_i = L_i L_i^T,
\end{equation}

gives
\( M = \begin{bmatrix} L_1 & W_1^T \\ W_2 & L_2 \\ \vdots & \vdots \\ W_N & L_N \end{bmatrix} \begin{bmatrix} L_1^T & W_1 \\ \vdots & \vdots \\ W_N^T & L_N^T \end{bmatrix} \)

where

\( w_i = a_i L_{i-1}^{-T}, \quad i = 2, \ldots, N. \)

It can be shown that if \( T \) is a symmetric, positive definite tridiagonal matrix, then the inverse \( T^{-1} \) is

\[
T^{-1} = \begin{bmatrix} u_{11}v_1 & u_{11}v_2 & \cdots & u_{11}v_m \\ u_{12}v_1 & u_{12}v_2 & \cdots & u_{12}v_m \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1}v_1 & u_{m2}v_2 & \cdots & u_{mm}v_m \end{bmatrix},
\]

where

\[
u_1 = 1, \quad u_2 = a_1/b_1,
\]

\[
u_i = \frac{a_{i-1}u_{i-1} - b_{i-2}u_{i-2}}{b_{i-1}}, \quad 3 \leq i \leq m,
\]

\[
v_m = \frac{1}{b_{m-1}u_{m-1} + a_m u_m},
\]

\[
v_i = \frac{1 + b_i u_i v_{i+1}}{a_i u_i - D_{i-1} b_{i-1}}, \quad 2 \leq i \leq m-1,
\]

\[
v_1 = \frac{1 + b_1 u_1 v_2}{a_1 u_1}.
\]

Concus et al.'s method INV uses the incomplete block Cholesky factorization with the three main diagonals of the inverse of \( A_i \)'s to form the \( A_i \)'s in the matrix \( M \). Concus et al. also modify the method INV so that the row sums of \( M \) are equal to the corresponding row sums of \( A \). This modification gives an improvement in the condition number of \( M^{-1}A \) for the natural ordering of the unknowns. We set

\[
\begin{align}
R_1 &= 0, \\
R_i &= A_i (\Delta_{i-1}^{-1} - A_{i-1} A_i^T), \quad 2 \leq i \leq N.
\end{align}
\]

Experimental results [21] indicate that MINV is quite robust for a large class of problems. Unfortunately, MINV is an inherently recursive algorithm and therefore does not vectorize in its present formulation.

### 2.3. Meurant Preconditioner

It is easy to see that the PCG method vectorizes trivially with the exception of solving the linear system \( Mz^k = r^k \). Meurant modified the preconditioner INV so that computing each new \( z^k \) was also fully vectorizable. Recall that the incomplete block Cholesky factorization is given by

\[
\Lambda_i = D_i,
\]

\[
\Lambda_i = D_i - A_i \Lambda_{i-1} A_i^T, \quad 2 \leq i \leq N
\]

where \( \Lambda_{i-1} \) is the approximation of \( \Delta_{i-1}^{-1} \) consisting of the
three main diagonals of $\Delta^{-1}_{i-1}$. Meurant uses the same strategy and replaces $\Delta^{-1}_{i}$ by its seven diagonals for INVW3 [15], [16], [17]. Therefore, where in INV one solves tridiagonals systems using the Cholesky factors of $\Delta_{i}$, in Meurant’s method one simply multiplies the right-hand side by the matrix consisting of the seven main diagonals. The algorithm is fully vectorizable but very little parallelism exists.

In order to utilize the multi-processing capabilities of modern computers, Meurant modifies both the PCG and the vectorized INV algorithms. By using the orthogonality relationships and the fact that $z^{k} = M^{-1}r^{k}$, he formulates an approximate or Modified PCG (MPCG) algorithm. Let $x^{0}$ be the initial estimate of $x$. Then $r^{0} = b - Ax^{0}$, $Mz^{0} = r^{0}$, $p^{0} = z^{0}$, and $s^{0} = (r^{0}, z^{0})$. Compute the following sequence of steps until convergence is achieved:

For $k \geq 1$

1) $Mv^{k} = Ap^{k}$,  
2) $(v^{k}, Ap_{k}^{k})$, $(Ap^{k}, p^{k})$, and $(z^{k}, z^{k})$,  
3) $a_{k} = (z^{k}, z^{k})/(Ap^{k}, p^{k})$,  
4) $s^{k+1} = s^{k} - a_{k}v^{k}$,  
5) $r^{k+1} = r^{k} - a_{k}Ap^{k}$,  
6) $z^{k+1} = z^{k} - a_{k}v^{k}$,  
7) $p^{k+1} = (z^{k}+a_{k}v^{k}) + \beta_{k+1}p^{k}$.

There are two additional inner products in (2.19), when compared to (1.5), which need to be computed, but Meurant gains parallelism by computing them simultaneously and by updating the vectors in step iv) with as many processors as are available. He also achieves parallelism in step i) by allowing $n$ available processors to each compute $\frac{1}{n}$ values of $v^{k}$ and ignoring some of the coupling between the terms from the matrix $M$. For example, if only two processors are available, the first processor computes

(2.20a) $\Delta_{1} = D_{1}^{\top}$,  
(2.20b) $\Delta_{i} = D_{1}^{\top} - A_{i}A_{i-1}^{\top}A_{i}^{\top}$,  
(2.21a) $\Delta_{N} = D_{N}^{\top}$,  
(2.21b) $\Delta_{i} = D_{i}^{\top} - A_{i+1}^{\top}A_{i+1}A_{i}$,  

The coupling term for computing $\Delta_{N/2}$ has been ignored for the sake of parallelism. Obviously as the number of processors increases, more coupling terms are ignored; and therefore the accuracy of the solution is degraded. But by sacrificing a little accuracy, Meurant has reduced the execution time for each iteration by incorporating parallelism in his algorithm. For some class of problems the MPCG method with parallelism can perform much better, in terms of CPU time, than the PCG method in its original formulation. In
his paper, Meurant refers to the MPCG method using two and four processors with the vectorized INV method as INVU2P and INVU4P, respectively.

2.4. Bramble, Pasciak, and Schatz Preconditioner

Bramble, Pasciak, and Schatz [2], [3] constructed preconditioners by re-examining the elliptic partial differential equation. They consider the following Dirichlet problem:

\[ \begin{align*}
L u &= \frac{2}{\lambda_{i,j}=1} \frac{\lambda_{i,j}}{\lambda_{i,j}} \left( a_{i,j} \frac{\partial u}{\partial \nu} \right) = f & \text{on } \Omega, \\
u &= 0 & \text{on } \partial \Omega,
\end{align*} \]

with \( a_{i,j} \) uniformly positive definite, bounded, and piecewise smooth. The generalized Dirichlet form is given by

\[ A(v, \phi) = \frac{2}{\lambda_{i,j}=1} \int_{\Omega} a_{i,j} \frac{\partial v}{\partial \nu} \frac{\partial \phi}{\partial \nu} \, dx, \]

which is defined for all \( v \) and \( \phi \) in the Sobolev space \( H^1(\Omega) \), where \( H^1(\Omega) \) is the space of functions with square integrable first derivatives. The weak formulation of the problem is given as follows: Find \( u \in H^1_0(\Omega) \) such that

\[ A(v, \phi) = (f, \phi) \]

for all \( \phi \in H^1_0(\Omega) \). Here \( H^1_0(\Omega) \) denotes all those functions in \( H^1(\Omega) \) which vanish on \( \partial \Omega \). The standard Galerkin approximation is defined by \( U \in S^0_h(\Omega) \) such that

\[ A(U, \chi) = (f, \chi) \]

for all \( \chi \in S^0_h(\Omega) \), where \( S^0_h(\Omega) \) is a finite dimensional subspace of \( H^1_0(\Omega) \). Let \( \{ \chi_i \}_{i=1}^N \) be a basis for the \( S^0_h(\Omega) \). Then

\[ U = \sum_{i=1}^N a_i \chi_i. \]

The coefficients \( a_i, i = 1, 2, \ldots, N \), are determined by the linear system

\[ \sum_{i=1}^N a_i A(\chi_i, \chi_j) = (f, \chi_j). \]

By preconditioning this system, Bramble et al. solve

\[ M(W, \chi) = (g, \chi) \quad \text{for all } \chi \in S^0_h \]

at each iteration of the conjugate gradient algorithm. Here \( M(W, \chi) \) is another bilinear form.

Bramble et al. impose a quasi-uniform triangulation on the domain \( \Omega \) and define the subspace \( S^0_h \) to consist of piecewise continuous linear functions. Additionally, they define the union of one or more contiguous triangles as a subregion \( \Omega_k \) such that the domain \( \Omega \) can be expressed as the union of all of the \( \Omega_k \). The boundaries of the \( \Omega_k \) correspond to the mesh lines of the triangulation of \( \Omega \). Let \( v_1, v_2, \ldots, v_M \) be the vertices that are determined by the vertices of the \( \Omega_k \) and \( \Gamma_{ij} \) be the segments connecting vertices \( v_i \) and \( v_j \) (see figure 2.1).
To construct the preconditioner for the resulting algebraic system, decompose the solution into $U = U_p + U_h$, where $U_p = 0$ on each $\Omega_k$ and satisfies

$$\lambda_k(U_p, x) = \lambda_k(U, x), \quad \forall x \in S^0_h(\Omega),$$

with

$$\lambda_k(U_p, x) = \frac{1}{2} \sum_{i,j=1}^{n} a_{ij} \frac{\partial U_p}{\partial x_i} \frac{\partial x_j}{\partial x_j} \, dx.$$ 

$a^{k} = [a_{ij}^{k}]$ is a positive definite matrix for each $k$. Since $U_p$ is determined on each $\Omega_k$ by the values of $U$ on each $\Omega_k$, $U_h$ satisfies

$$\lambda_k(U_h, x) = 0, \quad \forall x \in S^0_h(\Omega_k).$$

The solution $U$ is defined to be the sum of $U_p$ which vanishes on $\partial \Omega_k$ and $U_h$ which is the solution to the homogeneous equation. The function $U_h$ on $\Omega_k$ is again decomposed into $U_h = U_e + U_v$ where $U_v$ is the discrete harmonic function whose values on $\partial \Omega_k$ are the linear functions along each $\Gamma_{ij}$ with the same values as $U$ at the vertices. After $U_e$ and $U_v$ are computed for all of the $\Gamma_{ij}$'s, $U_h = U_e + U_v$ is determined in each $\Omega_k$ by solving a Dirichlet problem on each $\Omega_k$.

Bramble et al. prove that for any discrete harmonic function $W$ on $\Omega_k$,
\[(2.33) \quad a_0 \lambda_k(W, W) < |W|_{H^1, \Omega_k}^2 < a_1 \lambda_k(W, W),\]

where \(a_0\) and \(a_1\) are positive constants and \(|\cdot|_{H^1, \Omega_k}\) is the norm on the Sobolev space \(H^1(\Omega_k)\). If \(W = 0\) at the vertices then

\[(2.34) \quad |W|_{H^1, \Omega_k}^2 = \langle \delta_0^W, W \rangle_{\Gamma_{ij}},\]

with new values of \(a_0\) and \(a_1\) such that \(a_1/a_0 = O(1 + \ln(d/h)^2)\), where \(d\) is the maximum diagonal through the \(\Omega_k\)'s. The operator \(\ell_0\) is defined by

\[(2.35) \quad \langle \ell_0 w, \chi \rangle_{\Gamma} = \langle w', \chi' \rangle_{\Gamma}.

Bramble et al. also prove that \(\lambda_k(U_k, U_k)\) is bounded by

\[(2.36) \quad \sum_{i,j} (U_k(V_i) - U_k(V_j))^2,\]

finally, they construct the preconditioner DDL

\[(2.37) \quad \lambda(U_p, \phi_p) + \sum_{i,j} \langle \ell_0^E w, \phi \rangle_{\Gamma_{ij}} + \sum_{i,j} (U_k(V_i) - U_k(V_j)) (\phi(V_i) - \phi(V_j))\]

and prove that the condition number grows at most like \(1 + \ln(d/h)^2\). There are five steps in the construction of DDL.

**Step 1:** Compute \(U_p\) on each \(\Omega_k\) by fast Fourier transform (since each \(a_k\) is constant for \(\Omega_k\)).

**Step 2:** Compute \(U_e\) on \(\Gamma_{ij}\) by fast Fourier transform (approximate an edge coefficient by a constant).

**Step 3:** Compute \(U_v\) at the vertices \(V_i\) by direct solution of a coarse finite difference grid.

**Step 4:** Compute \(U_H\) on each \(\Omega_k\) by fast Fourier transforms.

**Step 5:** Form \(U = U_p + U_H\).

2.5. Glowinski, Periaux, and Fischler Preconditioner

Glowinski and Periaux [11] and Fischler [12] solve the following Dirichlet problem:

\[(2.38) \quad -\nabla \cdot (a \nabla \phi) = f \text{ in } \Omega,\]

\[(2.39) \quad \phi = g \text{ on } \Gamma,\]

where \(a > 0\) and \(\Gamma\) is the boundary of \(\Omega\). Consider the two domain problem with subdomains \(\Omega_1\) and \(\Omega_2\), where \(\gamma = \partial \Omega_1 \cap \partial \Omega_2\) (see Figure 2.3).

The solution to the problem of (2.38) and (2.39) also satisfies the following relations:
(2.40) \(-\nabla \cdot (a_i \nabla \phi_i) = f_i \) in \( \Omega_i, \quad i = 1, 2, \)

(2.41) \(\phi_i = g_i \) on \( \Gamma \cap \partial \Omega_i, \quad i = 1, 2, \)

(2.42) \(\phi_1 = \phi_2 \) on \( \gamma. \)

and

(2.43) \[ \sum_{i=1}^{2} a_i \frac{\partial \phi_i}{\partial n_i} = 0 \] on \( \gamma, \)

with \( f_i = f |_{\Omega_i}, \quad g_i = g |_{\Gamma \cap \partial \Omega_i}, \quad a_i = a |_{\Omega_i}, \quad \) and \( \phi_i = \phi |_{\Omega_i}. \)

Consider \( \mu \) defined over \( \gamma, \) such that \( \mu = \bar{\mu} |_{\gamma}, \)

where \( \bar{\mu} \in H_0^1(\Omega). \) Define \( \phi_i(\mu), \quad i = 1, 2, \) as the unique solution in \( H^1(\Omega_i) \) of

(2.44) \(-\nabla \cdot (a_i \nabla \phi_i) = 0 \) on \( \Omega_i, \quad i = 1, 2, \)

\( \phi_i = 0 \) on \( \Gamma \cap \partial \Omega_i, \quad i = 1, 2, \)

\( \phi_i = \mu \) on \( \gamma. \)

Glowinski and Periaux and Fischler define an operator \( A \) by

(2.44) \[ A\mu = \sum_{i=1}^{2} a_i \frac{\partial \phi_i}{\partial n_i}(\mu), \]

then, using Green's formula, prove that

(2.45) \[ \int_{\gamma} (A\mu) \nu \, d\gamma = \sum_{i=1}^{2} \int_{\Omega_i} \nabla \phi_i(\mu) \cdot \nabla \phi_i(\mu') \, dx, \quad \forall \mu, \mu' \in \Lambda_0, \]

where

\[ \Lambda_0 = \left\{ \mu | \mu \in L^2(\gamma), \exists \bar{\mu} \in H_0^1(\Omega) \text{ such that } \mu = \bar{\mu} |_{\gamma} \right\}. \]
Denote, as \( \lambda \) and \( \lambda_0 \), \( \lambda = \phi \gamma \) and \( \lambda_0 = \bar{\lambda}_0 \gamma \) with \( \bar{\lambda}_0 \in H^1_0(\Omega) \) such that \( \bar{\lambda}_0 = g \) on \( \Gamma \).

These authors' method is then to let \( \lambda^0 = \lambda_0 \) and to solve

\[
- \nabla \cdot (a_i \nabla \phi_i^0) = f_i \quad \text{in} \quad \Omega_i,
\]
\[
\phi_i^0 = g_i \quad \text{on} \quad \Gamma \cap \partial \Omega_i
\]
\[
\phi_i^0 = \lambda^0 \quad \text{on} \quad \Gamma.
\]

The steps of the method are the following: Compute \( g^0 \in \Lambda_0 \) as the solution of

\[
(2.46) \quad \sum_{i=1}^{2} a_i \nabla \phi_i^0 \cdot \nabla \phi_i^0 \, dx = \sum_{i=1}^{2} \left( a_i \nabla \phi_i^0 \cdot \nabla \phi_i^* \, dx - \int_{\partial \Omega_i} f_i \, du \, dx \right)
\]

with \( (\phi_i^0)_{i=1}^2 \in \Lambda_0 \) and \( (\phi_i^*)_{i=1}^2 \in \Lambda_0 \) satisfying

\[
\phi_i \in H^1(\Omega_i),
\]
\[
\phi_i^* = 0 \quad \text{on} \quad \partial \Omega_i \cap \Gamma,
\]
\[
\phi_i^* = \phi \quad \text{on} \quad \Gamma.
\]

Initialize \( w^0 = g^0 \).

Glowinski and Periaux and Fischler use the preconditioned conjugate gradient algorithm to compute updated values of \( \lambda_i^{n+1} \), \( g_i^{n+1} \), and \( w_i^{n+1} \); the updated values correspond to \( x_i^{k+1} \), \( z_i^{k+1} \), and \( p_i^{k+1} \) in the notation of the PCG algorithm as given in equation (1.5). The linear system \( M x^{k+1} = r^k \) is just

\[
\sum_{i=1}^{2} \int_{\Omega_i} a_i \nabla \phi_i^{k+1} \cdot \nabla \phi_i^0 \, dx = \sum_{i=1}^{2} \int_{\Omega_i} a_i \nabla \phi_i^k \cdot \nabla \phi_i^0 \, dx
\]
\[
- \sum_{i=1}^{2} \int_{\partial \Omega_i} a_i \nabla \phi_i^k \cdot \nabla \phi_i^0 \, dx
\]

with \( \chi_i^k \) the solution to the Poisson problem

\[
- \nabla \cdot (a_i \nabla \chi_i^k) = 0 \quad \text{in} \quad \Omega_i, \quad i = 1,2,
\]
\[
\chi_i^k = 0 \quad \text{on} \quad \Gamma \cap \partial \Omega_i, \quad i = 1,2,
\]
\[
\chi_i^k = w^k \quad \text{on} \quad \Gamma, \quad i = 1,2.
\]

Note that for this method the preconditioner \( M \) is equivalent to the matrix \( A_{33} \) in (2.1) and therefore is a preconditioning matrix for the capacitance matrix given in (2.2). Again the condition number of this method grows as the number of unknowns along \( \gamma \) increase.
CHAPTER III

DESCRIPTION OF THE NUMERICAL PROCEDURE

In Chapter I we indicated that one could use finite differences or finite elements to approximate a symmetric elliptic partial differential equation \( Ax = b \). The algebraic system that arises from the discretization of this equation is very large, sparse, and not well-conditioned. As we indicated in Chapter I, we seek an efficient, robust, vectorizable, and parallelizable preconditioner to economically compute the solution of the linear system.

In Chapter II substructuring was cast in terms of the Dirichlet problem as presented by Bramble et al. Recall that the underlying idea is to break up the domain of integration into many pieces, solve the appropriate equation on each of the pieces, then somehow construct the global solution from these local solutions. In this chapter we introduce additional approximations into this basic approach and then demonstrate how the ideas of Bramble et al., modified by the additional approximations, can be applied to the Neumann problem.

3.1. The Dirichlet Problem

Consider the Dirichlet problem

\[
\begin{align*}
(3.1a) & \quad -\nabla \cdot a(x,y) \nabla u(x,y) = f(x,y) \quad \text{on } \Omega, \\
(3.1b) & \quad u(x,y) = 0 \quad \text{on } \partial \Omega,
\end{align*}
\]

where \( 0 < a_0 \leq a(x,y) \leq a_1 \) and is piecewise smooth. The function \( f \in L^2(\Omega) \) where \( L^2 \) is the space of square integrable functions. The Dirichlet form is given by

\[
(3.2) \quad A(v,\phi) = \int_{\Omega} a(x,y) \triangledown v \cdot \triangledown \phi \, dx,
\]

which is defined for all \( v \) and \( \phi \) in the Sobolev space \( H^1(\Omega) \). The weak formulation of the problem is given by

\[
(3.3) \quad A(v,\phi) = (f,\phi)
\]

for all \( \phi \in H_0^1(\Omega) \). In the standard Galerkin approximation we seek \( U \in S_h^0(\Omega) \) such that

\[
(3.4) \quad A(U,\chi) = (f,\chi)
\]

for all \( \chi \in S_h^0(\Omega) \), where \( S_h^0(\Omega) \) is a finite dimensional subspace of \( H_0^1(\Omega) \). Letting \( \{\chi_i\}_{i=1}^N \) be a basis for \( S_h^0(\Omega) \), we have

\[
(3.5) \quad U = \sum_{i=1}^N a_i \chi_i.
\]

The linear system defining the coefficients \( a_i \), \( i = 1,2, \ldots, N \), is then
(3.6) \[ \sum_{i=1}^{N} a_i A(x_i, X_j) = (f, x_i). \]

By preconditioning this system, we solve

(3.7) \[ M(W, x) = (g, x), \quad \forall x \in S_h, \]

at each iteration. We proceed as follows.

Partition the domain of integration \( \Omega = [0, x_L] \times [0, y_L] \) by the grid blocks defined by

(3.8a) \[ \delta_x : 0 = x_0 < x_1 < \ldots < x_{N_x} = x_L, \]

(3.8b) \[ \delta_y : 0 = y_0 < y_1 < \ldots < y_{N_y} = y_L, \]

as illustrated in Figure 3.1.

Define two additional partitions \( \Delta x \) and \( \Delta y \) such that \( \Delta x \) and \( \Delta y \) are subsets of \( \delta_x \) and \( \delta_y \), respectively.

(3.9a) \[ \Delta x : 0 = \bar{x}_0 < \bar{x}_1 < \ldots < \bar{x}_{N_x} = x_L, \]

(3.9b) \[ \Delta y : 0 = \bar{y}_0 < \bar{y}_1 < \ldots < \bar{y}_{N_y} = y_L. \]

Define the set of \( \Omega_k \) for \( k = 1, 2, \ldots, N_x N_y \) to be the subregions which are determined by the tensor product of \( \Delta x \) and \( \Delta y \). Let the set of \( V_i \) for \( i = 1, 2, \ldots, (N_x) (N_y) \) be the vertices of the \( \Omega_k \)'s and \( \Gamma_{ij} \) be the segments connecting vertices \( V_i \) and \( V_j \). As shown in figure 3.2, all elements of each of the sets \( \delta_x, \delta_y, \Delta x, \Delta y, \{V_i\}, \) and \( \{\Gamma_{ij}\} \) are ordered using the natural ordering. Here /// denotes the boundary of the \( \Omega_k \).

Figure 3.1 Mesh for Rectangular Domain
To construct the preconditioner for the algebraic system, we decompose the solution into \( U = U_p + U_H \) where \( U_p = 0 \) on \( \partial \Omega_k \) and satisfies
\[
A_k(U_p, x) = A_k(U, x), \quad \forall \ x \in S_h^0(\Omega_k),
\]
with
\[
A_k(U_p, x) = \int_{\Omega_k} a_k(u_p, v_x) \, dx.
\]

An important difference in the bilinear form \( A_k \) when compared to \( A_k \) from equation (2.31) is that in (3.11) the coefficient \( a_k(x, y) = a(x, y) \), i.e., \( a_k \) models the coefficient in the differential equation within each \( \Omega_k \). In the formulation of Bramble et al., \( a_k \) is approximated by a single constant over the entire region \( \Omega_k \).

\( U_p \) is determined on each \( \Omega_k \) by the values of \( U \) on each \( \Omega_k \), therefore \( U_H \) must satisfy
\[
A_k(U_H, x) = 0, \quad \forall \ x \in S_h^0(\Omega_k).
\]

The solution \( U \) is defined to be the sum of \( U_p \), which vanishes on \( \partial \Omega_k \), and \( U_H \), which is the solution to the homogeneous equation. The function \( U_H \) on \( \Omega_k \) is again decomposed into \( U_H = U_e + U_v \) where \( U_v \) is the discrete harmonic function whose values on \( \partial \Omega_k \) are the linear functions along each \( \Gamma_{ij} \) with the same values as \( U \) at the vertices. \( U_e \) must be the discrete harmonic in \( \Omega_k \) which
vanishes at all the vertices. After \( U_e \) and \( U_v \) are computed for all of the \( \Gamma_{ij} \)'s, \( U_H = U_e + U_v \) is determined in each \( \Omega_k \) by solving a Dirichlet problem on each \( \Omega_k \).

If we assume that we accurately solve for \( U_p \) and \( U_H \) on each \( \Omega_k \), then we have the same relationship between the solution on the boundary and the interior of each \( \Omega_k \) that Bramble et al. derived for their preconditioner. We know that if \( U_H = 0 \) at the vertices then

\[
|U_H|^2_{\Omega_k} = \| \mathbf{u} \|^2_{\Omega_k} \leq \mathbf{K} \mathbf{u} = \mathbf{r} .
\]

Again, this indicates that we can solve for \( V_e \) on each \( \Omega_k \) as a separate one-dimensional problem for each \( \Gamma_{ij} \). Additionally, we can utilize their expression that bounds

\[
A_k(U_v,U_v) = \sum_{\Gamma_{ij} \in \Omega_k} (u_v(V_j)-u_v(V_j))^2.
\]

At each iteration of the conjugate gradient method we compute as follows:

**Step 1:** Find \( U_p \) on each \( \Omega_k \).

**Step 2:** Find \( U_e \) on each \( \Gamma_{ij} \).

**Step 3:** Find \( U_v \) for all \( \Gamma_{ij} \).

**Step 4:** Find \( U_H \) on each \( \Omega_k \).

**Step 5:** Form \( U = U_p + U_H \).

Although we have the same basic steps as in DD1, there are some fundamental differences in exactly what are computed and how they are computed.

Throughout the formulation of this preconditioner we expressed the calculations in finite element terms. Since finite element approximations and finite difference approximations differ only in the choice of quadrature rules [21], we approximate (3.1) by point-centered finite differences instead of finite elements. This yields a block tridiagonal matrix

\[
A = \begin{bmatrix}
D_1 & A_1^T \\
A_2 & D_2 \\
& & \ddots & \ddots \\
& & \ddots & A_N^T \\
& & & & D_N
\end{bmatrix},
\]

where

\[
D_1 = \begin{bmatrix}
a_1 & b_2 \\
b_2 & a_2 & \ddots \\
& \ddots & \ddots & b_{N_X} \\
& & \ddots & a_{N_X}
\end{bmatrix}, \quad A_i = \begin{bmatrix}
c_1 \\
c_2 \\
& \ddots \\
& & \ddots \\
& & & c_{N_X}
\end{bmatrix}
\]
In Step 1 we use the incomplete factorization method MINV to compute $U_p$ on each $\Omega_k$. This allows the coefficient $a$ in the differential equation to vary with respect to $x$ and $y$ within each subregion as well as allowing for non-uniform mesh. It is clear that MINV cannot be vectorized for one subregion. But if the number of $x$ variables remains constant for all $\Omega_k$ and the number of $y$ variables remains constant for all $\Omega_k$, the solutions $U_p$ can be simultaneously computed for all $\Omega_k$ by vectorizing across subregions. Note that this procedure does not force the number of parameters in each direction to be the same nor does it restrict the method to uniform mesh or square domains. Instead of applying MINV iteratively until the error within each $\Omega_k$ has been reduced to a specified tolerance, we take only the first step.

After the $U_p$ have been computed for each $\Omega_k$, the residual is formed over the entire domain $\Omega_k$. Note that the errors should be smallest within the interior of the $\Omega_k$ and largest in the vicinity of the $\Gamma_{ij}$. The residual becomes the right hand side of the equation when solving for $U_e$ on the $\Gamma_{ij}$. As in DDL we choose a single constant $\bar{a}$ to approximate the coefficient $a$ in the differential equation along $\Gamma_{ij}$. This allows us to solve for the $U_e$ using fast Fourier transforms. In fact, since all of the horizontal edges have the same number of unknowns, the Fourier transform calculations vectorize across all of the horizontal edges. Similarly, the vertical Fourier transforms vectorize across all of the vertical edges. If meshes that are non-uniform but quasi-uniform on each $\Gamma_{ij}$ are used, the error committed in assuming the uniform spacing via the Fourier transform is small.

The right hand side needed for step 3 of page 38 is computed as for DDL, i.e., integrate the residual with $U_p$ against the set of basis functions that are defined over the entire domain $\Omega$. Some savings in CPU time can be achieved here by processing all interior points at the same time, then processing the vertical and horizontal edges. Note that the solution $U_v$ at the vertices $V_i$ in step 3 is just the finite difference solution of (3.1) using the coarse grid of $\Delta x$ and $\Delta y$. The resulting linear system is symmetric, positive definite, and banded. The LINPACK routines SPBFA and SPBSL, for banded Cholesky factorization and solution, are used first to factor, then to forward and backward solve the system of equations. Once $U_v$ has been determined at the vertices, we linearly interpolate to compute the values at the interior nodes.

At this point we have $U_p$ defined on the $\Omega_k$ and $U_H$ defined on the $3\Omega_k$. We compute the residual for $U_p + U_H$, then solve the homogeneous equation (3.12) with the residual as the boundary conditions. Just as we vectorized over the
subregions \( \Omega_k \) to compute \( U_p \), we vectorize across the \( \Omega_k \) to compute \( U_H \) on \( \Omega_k \).

The final step is to add the various components \( U_p, U_e, U_v \), and \( U_H \) to form the solution \( U \). At every iteration of the conjugate gradient we have the quantities that are needed in steps 1 to 5. Since we solve for \( U_p \) and \( U_H \) at each iteration, it is evident that there is an optimal number of subregions for a given mesh that minimize the number of operations in each iteration. In Chapter IV this aspect of the algorithm is discussed.

Ortega and Voigt [19] state that "effective utilization of these machines will require decomposition of the problem into a small number of large, relatively independent parts, and vectorization of the individual parts." This, in fact, is the approach followed by all published domain decomposition methods. But here we claim that if we impose some restrictions, we can formulate an effective preconditioner by using a non-vectorized iterative method for the subproblems while vectorizing across subregions, provided that the number of subdomains is large. Unlike preconditioners for capacitance matrices, where the condition number of the system grows as the number of nodes on the interfaces increase, the method developed by Bramble, et al. reduces the condition number of the system by increasing the number of subdomains.

3.2. The Neumann Problem

The substructuring procedure was defined for the Dirichlet problem but can be adapted to the Neumann problem. Recall that the process included the use of nodal basis functions and point-centered finite differences. We consider the Neumann problem

\begin{align}
\frac{\partial}{\partial n} a(x,y) u(x,y) &= f(x,y) \quad \text{on } \Omega \\
- a(x,y) \nabla u(x,y) \cdot \nu &= 0 \quad \text{on } \partial \Omega,
\end{align}

where \( \nu \) is the outward normal vector on \( \partial \Omega \) with \( \int_{\Omega} f = 0 \) and \( 0 < a_0 < a(x,y) < a_1 \). Here we do not require \( f \in L^2(\Omega) \) but \( f \in H^{-1/2}, \varepsilon > 0 \). We partition the domain of integration as in the Dirichlet case with \( \delta_x \) and \( \delta_y \) defined in (3.8).

The important difference here is that we need the solution \( U \) at the center of the grid blocks, not at the vertices of the grid block. By superimposing an additional grid such that the centers of the original grid now correspond to the corners of the new grid, we can apply the method developed in the previous section for the Dirichlet problem. Define this new grid by

\begin{align}
\delta_x & : \frac{x_0 + x_1}{2} = \hat{x}_0 < \hat{x}_1 < \ldots < \hat{x}_{N_x-1} = \frac{x_N + x_{N-1}}{2} \\
\delta_y & : \frac{y_0 + y_1}{2} = \hat{y}_0 < \hat{y}_1 < \ldots < \hat{y}_{N_y-1} = \frac{y_N + y_{N-1}}{2}
\end{align}
(see figure 3.3). Block-centered finite differences are applied to the fine mesh problem described by (1.1) and (1.3) to give

\[
- \frac{1}{x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}} \left( a_{i+\frac{1}{2}, j} \frac{u_{i+1, j} - u_{i, j}}{x_{i+1} - x_i} - a_{i-\frac{1}{2}, j} \frac{u_{i+1, j} - u_{i, j}}{x_{i-1} - x_i} \right)
\]

\[
- \frac{1}{y_{j+\frac{1}{2}} - y_{j-\frac{1}{2}}} \left( a_{i, j+\frac{1}{2}} \frac{u_{i, j+1} - u_{i, j}}{y_{j+1} - y_j} - a_{i, j-\frac{1}{2}} \frac{u_{i, j+1} - u_{i, j}}{y_{j-1} - y_j} \right)
\]

\[= f_{ij} \]

The coarse mesh problem is also approximated by block-centered finite differences to give an equation similar to (3.19). As long as the two grids defined by \( \delta_x \times \delta_y \) and \( \delta_x' \times \delta_y' \) use uniform sampling, the cell centers of one grid correspond to the corners of the other grid. Thus, the unknowns of the coarse mesh are a subset of the unknowns for the fine mesh; otherwise, there is some offset between the corresponding unknowns (see figure 3.4). This offset implies that the convergence rate for the non-uniform case is slightly decreased to account for this discrepancy. In Chapter IV we demonstrate that this slower rate does not significantly alter the convergence properties of the algorithm.
3.3. Computational Aspects

Developing algorithms for vector or parallel machines requires additional considerations such as the specific hardware capabilities of the machine, the type of suitable algorithms, the parallelism which exists or could exist, and possible increased overhead. Parallelism is incorporated into domain decomposition algorithms in the domain splitting to generate many smaller, independent subproblems. For the method proposed in this dissertation, the price we pay includes additional memory requirements and increased data movement.

Computation of the inner products for the conjugate gradient algorithm requires that the array elements be in the global natural ordering of the problem. Conversely, the computational phase of solving many similar subproblems requires that the local natural ordering be used for the unknowns. This difference has both memory and data flow implications. First, extra memory is needed for the finite difference matrix, right hand sides, and unknowns for each subproblem. An additional $3NxNy$ words of memory are required for the subproblems. Second, extra data movement is needed to go back and forth between each of the different orderings; data movement increases as the number of unknowns in the $x$-direction decreases for each subdomain. Additional
memory and data movement are also needed for each of the local problems on the edges. An extra $N_x N_y N_p$ words are needed for the local edge problems, where $N_p = 2^{\#} M = \max(N_x/N_x, N_y/N_y)$. Again the data movement increases as the number of interior edge nodes in both x- and y-directions decrease. Use of the banded Cholesky algorithm for the coarse problem requires $N_x N_y N_p$ extra words for the factored finite difference matrix. This need for extra words, of course, can be reduced if instead of using the direct solver, we apply the iterative method MINV to this matrix.

In this chapter we have illustrated how the DDI algorithm of Bramble, et al. is the basic framework for a vectorizable domain decomposition method. We used only the first iteration of the MINV incomplete block Cholesky factorizations for approximately solving for $U_p$ and $U_H$ on each subdomain. By vectorizing across all subdomains we were able to simultaneously solve all of the subproblems. The edge solutions were also computed concurrently by using a parallel fast Fourier transform routine. Finally, we showed that this method was easily modified for solving the Neumann problem. In Chapter IV we present the numerical results for various problems in order to confirm some of the claims we have made in this chapter.

CHAPTER IV

NUMERICAL RESULTS

In this chapter we present numerical results for several test problems using the procedure which was discussed in Chapter III. Results are presented for the coefficient $a(x,y)$ when it is (a) constant, (b) piecewise discontinuous, or (c) continuously varying throughout the domain of integration. Both uniform and non-uniform meshes are considered for both square and rectangular domains. For each of the test problems we use the following stopping criteria for convergence:

$$
\frac{\|e_n\|_2}{\|e_0\|_2} < \varepsilon = 10^{-6},
$$

where the $l_2$ norm (Euclidean norm) of the residual at the $n$th iteration is reduced by six orders of magnitude. Except where noted otherwise, all of the results which are presented were run on a Cray-1S. Both the cpu time in seconds and the iteration count required for convergence are summarized in tables and graphs for each of the problems.

We look first at the following problem:
\begin{align}
(4.2a) \quad -\nabla \cdot (a(x,y) \nabla u(x,y)) &= f \quad \text{on } \Omega,
(4.2b) \quad -a(x,y) \nabla u(x,y) \cdot n &= 0 \quad \text{on } \partial \Omega,
\end{align}

with \( a(x,y) = a_{11}(x,y) = 1 \), \( \Omega = (0,1) \times (0,1) \), \( f = \delta(0,0) - \delta(1,1) \), and \( \delta(\cdot,\cdot) \) denotes the Dirac delta function. The right hand side \( f \) will remain constant for all of the numerical examples presented in this chapter. We choose \( N_x = N_y \) from eq. (3.8) to be 65 or 129, and \( \bar{N}_x = \bar{N}_y \) from eq. (3.9) to be 2, 4, 8, 16, or 32, with the number of unknowns in each subregion being \((N_x-1)/\bar{N}_x\) for the \( x \)-direction and \((N_y-1)/\bar{N}_y\) for the \( y \)-direction. Table 4.1 summarizes the numerical results for \( N_x = N_y = 65 \) and for \( N_x = N_y = 129 \).

<table>
<thead>
<tr>
<th>( N_x \times N_y )</th>
<th>( \bar{N}_x \times \bar{N}_y )</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 \times 65</td>
<td>2 \times 2</td>
<td>39</td>
<td>2.25</td>
</tr>
<tr>
<td></td>
<td>4 \times 4</td>
<td>33</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>8 \times 8</td>
<td>31</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>16 \times 16</td>
<td>27</td>
<td>1.1</td>
</tr>
<tr>
<td>129 \times 129</td>
<td>4 \times 4</td>
<td>46</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>8 \times 8</td>
<td>39</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>16 \times 16</td>
<td>37</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>32 \times 32</td>
<td>34</td>
<td>5.3</td>
</tr>
</tbody>
</table>

It is evident from studying the table that for a given \( N_x \) and \( N_y \), the number of iterations decreases as the number of subregions \( \bar{N}_x \) and \( \bar{N}_y \) increases. It is also apparent that the cpu time initially decreases as the number of subregions increases, but then it increases again. This is because the vectorized portions of the algorithm compete with one another. The algorithm vectorizes across the total number of subregions when the individual subproblems are solved, but the data movement part of the process vectorizes over the number of \( x \)-variables in a given subdomain (e.g. \((N_x-1)/\bar{N}_x\)). Therefore when the number of total subregions increases, the cpu time needed for the solution of the
individual problems decreases; but the cpu time that is
needed in moving the data in order to solve these subprob-
lems is increased. It appears that the best compromise
is to select $\overline{N_x} = \sqrt{N_x}$ and $\overline{N_y} = \sqrt{N_y}$, so that one phase
never dominates the other phase. For $N_x = N_y = 65$ we
would choose $\overline{N_x} = \overline{N_y} = 8$ and for $N_x = N_y = 129$ we would
choose $\overline{N_x} = \overline{N_y} = 8$; even though fewer iterations are
needed for other choices of $N_x$ and $N_y$. Other examples
include detailed statistics to support this argument.

Tom Hewitt of CRAY Research optimized some of the code
for the CRAY-XMP machine. Table 4.2 indicates what kind of
performance is possible on the XMP with only one processor.
The average MFLOP rate for the $N_x = 129$ by $N_y = 129$ problem
was 90.4 MFLOPS, while the average for an $N_x = 257$ by $N_y = 259$
problem was 94.4 MFLOPS. Since about twenty-five percent of
the cpu time was devoted to data movement, use of the hard-
ware GATHER/SCATTER equipment would essentially eliminate
all of the data movement time. Modifying the code to support
all four processors would also decrease the cpu time by a
factor of about four, since no extra data communication is
needed to utilize all the processors.

For the second example we solve eq. (4.2) with
\[ a = a_2(x, y) \]
\[
a_2(x, y) = \begin{cases} 
1, & 0.0 \leq x \leq 0.5, \\
10^{-6}, & 0.5 < x \leq 1.0
\end{cases}
\]

with domain \( \Omega \) and \( f \) defined as for the above problem.

Table 4.3 summarizes convergence results for \( Nx \times Ny = 33 \), \( 65 \), and \( 129 \). Again we see the same behavior in the iteration count versus the cpu time as in the previous problem.

Based on this information, the optimal selection is \( Nx = Ny = 8 \) for \( Nx = Ny = 65 \) and \( Nx = Ny = 8 \) for \( Nx = Ny = 129 \). Even though the coefficient in the differential equation \( a_2(x, y) \) varies drastically from one side of the domain to the other side, the number of iterations needed for convergence is only about thirty percent higher than needed for \( a_1(x, y) \). In Table 4.4 we demonstrate that the mesh selection for both the fine and coarse mesh need not be related to powers of two.

In the next example the function \( a(x, y) \) varies continuously across the domain of integration,

\[
a_3(x, y) = \frac{1}{1 + 10^6(x^2 + y^2)}.
\]

Table 4.5 indicates the number of iterations and cpu time for \( Nx = Ny = 65 \) and \( 129 \). The same dependency on the number of subdomains used versus the corresponding cpu time is observed for this example as in the previous tests. It is

<table>
<thead>
<tr>
<th>( Nx \times Ny )</th>
<th>( Nx \times Ny )</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 \times 33</td>
<td>2 \times 2</td>
<td>32</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>4 \times 4</td>
<td>38</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>8 \times 8</td>
<td>37</td>
<td>0.44</td>
</tr>
<tr>
<td>65 \times 65</td>
<td>2 \times 2</td>
<td>44</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>4 \times 4</td>
<td>41</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>8 \times 8</td>
<td>44</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>16 \times 16</td>
<td>46</td>
<td>1.9</td>
</tr>
<tr>
<td>129 \times 129</td>
<td>4 \times 4</td>
<td>53</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>8 \times 8</td>
<td>53</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>16 \times 16</td>
<td>49</td>
<td>3.8</td>
</tr>
</tbody>
</table>
### Table 4.4

Iteration and CPU Statistics for $a(x,y) = a_2(x,y)$

for $Nx = Ny = 101$

<table>
<thead>
<tr>
<th>$Nx \times Ny$</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 $\times$ 4</td>
<td>61</td>
<td>4.8</td>
</tr>
<tr>
<td>10 $\times$ 10</td>
<td>60</td>
<td>2.9</td>
</tr>
<tr>
<td>20 $\times$ 20</td>
<td>59</td>
<td>5.4</td>
</tr>
</tbody>
</table>

### Table 4.5

Iteration and CPU Statistics for $a(x,y) = a_3(x,y)$

<table>
<thead>
<tr>
<th>$Nx \times Ny$</th>
<th>$Nx \times Ny$</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 $\times$ 65</td>
<td>2 $\times$ 2</td>
<td>$&gt; 100$</td>
<td>$&gt; 4.0$</td>
</tr>
<tr>
<td>4 $\times$ 4</td>
<td>4 $\times$ 4</td>
<td>70</td>
<td>1.75</td>
</tr>
<tr>
<td>8 $\times$ 8</td>
<td>8 $\times$ 8</td>
<td>50</td>
<td>1.15</td>
</tr>
<tr>
<td>16 $\times$ 16</td>
<td>16 $\times$ 16</td>
<td>40</td>
<td>1.65</td>
</tr>
<tr>
<td>129 $\times$ 129</td>
<td>4 $\times$ 4</td>
<td>$&gt; 100$</td>
<td>$&gt; 8.0$</td>
</tr>
<tr>
<td>8 $\times$ 8</td>
<td>16 $\times$ 16</td>
<td>62</td>
<td>4.8</td>
</tr>
</tbody>
</table>
Table 4.6

Iteration and CPU Statistics for $a(x,y) = a_4(x,y)$

<table>
<thead>
<tr>
<th>$N_x \times N_y$</th>
<th>$\bar{N}_x \times \bar{N}_y$</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$33 \times 33$</td>
<td>$1 \times 1$</td>
<td>71</td>
<td>3.70</td>
</tr>
<tr>
<td></td>
<td>$2 \times 2$</td>
<td>44</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>$4 \times 4$</td>
<td>34</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>$8 \times 8$</td>
<td>29</td>
<td>0.35</td>
</tr>
<tr>
<td>$65 \times 65$</td>
<td>$2 \times 2$</td>
<td>62</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>$4 \times 4$</td>
<td>45</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>$8 \times 8$</td>
<td>36</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>$16 \times 16$</td>
<td>35</td>
<td>1.5</td>
</tr>
<tr>
<td>$129 \times 129$</td>
<td>$4 \times 4$</td>
<td>55</td>
<td>4.7</td>
</tr>
<tr>
<td></td>
<td>$8 \times 8$</td>
<td>43</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>$16 \times 16$</td>
<td>46</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>$32 \times 32$</td>
<td>44</td>
<td>6.8</td>
</tr>
</tbody>
</table>

Interesting to note that the reduction in terms of iteration count versus total number of subdomains is greater than for either $a_1(x,y)$ or $a_2(x,y)$, although the total cpu for the best choice of $N_x$ and $N_y$ is higher. From Bramble, et al. we know that if subproblems were solved to convergence, the total number of iterations should decrease as the number of subdomains increase. Bramble et al.'s results appear true for the present procedure provided that some constraints are adhered to. These constraints will become obvious in later examples.

The fourth example is similar to (4.4),

$$(4.5) \quad a_4(x,y) = \frac{1}{1 + 100(x^2 + y^2)}$$

but varies only two orders of magnitude across the domain of integration instead of the six orders of (4.4). Table 4.6 illustrates the convergence properties of this problem for $N_x = N_y = 33, 65, \text{ and } 129$. Tables 4.7, 4.8, and 4.9 provide a breakdown of the cpu time for $N_x = N_y = 33$, and Tables 4.10, 4.11, and 4.12 supply the details for $N_x = N_y = 65$. The information for $N_x = N_y = 33$ is plotted in Figure 4.1. From this figure it is clear that the number of iterations increases as the total number of subregions $(N_x \times N_y)$ goes to unity, but that the number of iterations decreases rapidly for more than twelve subdomains. Tables 4.13, 4.14,
### Table 4.7
CPU Statistics for $a(x,y) = a_4(x,y)$
with $N_x = N_y = 33$ and $N_x = N_y = 2$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>.01</td>
<td>1.4</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>.01</td>
<td>1.4</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>.05</td>
<td>6.9</td>
</tr>
<tr>
<td>Matrix-Vector</td>
<td>.02</td>
<td>2.8</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_p$ and $V_H$</td>
<td>.52</td>
<td>72.2</td>
</tr>
<tr>
<td>$V_e$</td>
<td>.09</td>
<td>12.5</td>
</tr>
<tr>
<td>$V_v$</td>
<td>.02</td>
<td>2.8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>.72</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>

### Table 4.8
CPU Statistics for $a(x,y) = a_4(x,y)$
with $N_x = N_y = 33$ and $N_x = N_y = 4$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.00</td>
<td>0.0</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.01</td>
<td>3.7</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.06</td>
<td>22.2</td>
</tr>
<tr>
<td>Matrix-Vector</td>
<td>0.02</td>
<td>7.4</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_p$ and $V_H$</td>
<td>0.09</td>
<td>33.3</td>
</tr>
<tr>
<td>$V_e$</td>
<td>0.05</td>
<td>18.5</td>
</tr>
<tr>
<td>$V_v$</td>
<td>0.04</td>
<td>14.8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.27</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>
Table 4.9

CPU Statistics for \( a(x, y) = a_4(x, y) \)

with \( N_x = N_y = 33 \) and \( \bar{N}_x = \bar{N}_y = 8 \)

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.00</td>
<td>0.0</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.01</td>
<td>2.9</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.11</td>
<td>32.4</td>
</tr>
<tr>
<td>Matrix-Vector</td>
<td>0.01</td>
<td>2.9</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( V_p ) and ( V_H )</td>
<td>0.03</td>
<td>8.8</td>
</tr>
<tr>
<td>( V_e )</td>
<td>0.07</td>
<td>20.8</td>
</tr>
<tr>
<td>( V_v )</td>
<td>0.11</td>
<td>32.4</td>
</tr>
<tr>
<td>Total</td>
<td>0.34</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Figure 4.1 Iterations and CPU versus Number of Subregions for \( a_4(x, y) \) with \( N_x = N_y = 33 \).
### Table 4.10

CPU Statistics for \( a(x,y) = a_4(x,y) \)  
with \( N_x = N_y = 65 \) and \( \bar{N}_x = \bar{N}_y = 4 \)

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.01</td>
<td>0.8</td>
</tr>
<tr>
<td>Inner Products, Updating Vectors</td>
<td>0.04</td>
<td>3.3</td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.28</td>
<td>23.7</td>
</tr>
<tr>
<td>Matrix-Vector Multiply</td>
<td>0.08</td>
<td>6.8</td>
</tr>
<tr>
<td>( V_P ) and ( V_H )</td>
<td>0.61</td>
<td>51.7</td>
</tr>
<tr>
<td>( V_e )</td>
<td>0.13</td>
<td>11.0</td>
</tr>
<tr>
<td>( V_H )</td>
<td>0.03</td>
<td>2.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1.18</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>

### Table 4.11

CPU Statistics for \( a(x,y) = a_4(x,y) \)  
with \( N_x = N_y = 65 \) and \( \bar{N}_x = \bar{N}_y = 8 \)

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.00</td>
<td>0.0</td>
</tr>
<tr>
<td>Inner Products, Updating Vectors</td>
<td>0.03</td>
<td>3.7</td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.24</td>
<td>30.0</td>
</tr>
<tr>
<td>Matrix-Vector Multiply</td>
<td>0.07</td>
<td>8.8</td>
</tr>
<tr>
<td>( V_P ) and ( V_H )</td>
<td>0.18</td>
<td>22.5</td>
</tr>
<tr>
<td>( V_e )</td>
<td>0.14</td>
<td>17.5</td>
</tr>
<tr>
<td>( V_H )</td>
<td>0.14</td>
<td>17.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.80</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>
Table 4.12

CPU Statistics for $a(x,y) = a_4(x,y)$  
with $N_x = N_y = 65$ and $\bar{N}_x = \bar{N}_y = 16$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.00</td>
<td>0.0</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.04</td>
<td>2.8</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.46</td>
<td>32.4</td>
</tr>
<tr>
<td>Matrix-Vector Multiply</td>
<td>0.07</td>
<td>4.9</td>
</tr>
<tr>
<td>$V_P$ and $V_H$</td>
<td>0.09</td>
<td>6.3</td>
</tr>
<tr>
<td>$V_e$</td>
<td>0.26</td>
<td>18.3</td>
</tr>
<tr>
<td>$V_V$</td>
<td>0.50</td>
<td>35.2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1.42</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>

Table 4.13

CPU Statistics for $a(x,y) = a_4(x,y)$  
with $N_x = N_y = 129$ and $\bar{N}_x = \bar{N}_y = 4$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.09</td>
<td>1.9</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.22</td>
<td>4.7</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.57</td>
<td>12.1</td>
</tr>
<tr>
<td>Matrix-Vector Multiply</td>
<td>0.40</td>
<td>8.5</td>
</tr>
<tr>
<td>$V_P$ and $V_H$</td>
<td>3.01</td>
<td>63.9</td>
</tr>
<tr>
<td>$V_e$</td>
<td>0.36</td>
<td>7.6</td>
</tr>
<tr>
<td>$V_V$</td>
<td>0.07</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4.71</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>
Table 4.14

CPU Statistics for $a(x,y) = a_4(x,y)$
with $N_x = N_y = 129$ and $\bar{N}_x = \bar{N}_y = 8$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.03</td>
<td>1.1</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.17</td>
<td>6.3</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>0.68</td>
<td>25.4</td>
</tr>
<tr>
<td>Matrix-Vector</td>
<td>0.31</td>
<td>11.6</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_p$ and $V_H$</td>
<td>0.99</td>
<td>36.9</td>
</tr>
<tr>
<td>$V_e$</td>
<td>0.33</td>
<td>12.3</td>
</tr>
<tr>
<td>$V_V$</td>
<td>0.17</td>
<td>6.3</td>
</tr>
<tr>
<td>Total</td>
<td>2.68</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 4.15

CPU Statistics for $a(x,y) = a_4(x,y)$
with $N_x = N_y = 129$ and $\bar{N}_x = \bar{N}_y = 16$

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU (seconds)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINV Construction</td>
<td>0.01</td>
<td>0.3</td>
</tr>
<tr>
<td>Inner Products,</td>
<td>0.18</td>
<td>5.0</td>
</tr>
<tr>
<td>Updating Vectors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Movement</td>
<td>1.22</td>
<td>34.0</td>
</tr>
<tr>
<td>Matrix-Vector</td>
<td>0.33</td>
<td>9.2</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_p$ and $V_H$</td>
<td>0.63</td>
<td>17.5</td>
</tr>
<tr>
<td>$V_e$</td>
<td>0.57</td>
<td>15.8</td>
</tr>
<tr>
<td>$V_V$</td>
<td>0.65</td>
<td>18.1</td>
</tr>
<tr>
<td>Total</td>
<td>3.59</td>
<td>100.0</td>
</tr>
</tbody>
</table>
4.15 summarize the statistics for $N_x = N_y = 129$. The same pattern for the iteration count and the cpu are observed as for the previous examples.

Table 4.16 provides the above information but tabulated for $N_x = N_y = 4$ and $N_x = N_y = 8$. Figure 4.3 also illustrates the convergence results. The curve for cpu time increases almost linearly with respect to the increase in mesh, and the curve for number of iterations appears to increase logarithmically with respect to the increase in mesh. In fact, this is the same logarithmic behavior that Bramble, et al. derived, even though we only use the first step of the MINV block preconditioner for each subdomain for the computation of $U_0$ and $U_H$.

From Table 4.6 we determine the "best cases" for each of the meshes (here the best case means choosing $N_x = \sqrt{N_x}$ and $N_y = \sqrt{N_y}$). In Figure 4.3 we see this same information as in Table 4.6. It is interesting to observe that both the number of iterations and the cpu time increase almost linearly with respect to the total number of unknowns and not quadratically. From these calculations we conclude that, especially for large problems, the method of this dissertation is a practical method for solving the two-dimensional elliptic equation.

The next example demonstrates that non-uniform sampling can be used in connection with our numerical procedure.
We define \( a(x,y) \) as in eq. (4.5). The mesh spacing on each edge of the domain is specified to be one-half of the mesh spacing on the remaining interior of the domain. We let \( Nx = Ny = 33, 65, \) and \( 129 \) and let \( Nx = Ny = 2, 4, 8, \) and \( 16 \). Table 4.17 shows very little change in the convergence results; they indicate the same pattern that is observed for all of the previous examples. Since we use the block preconditioner MINV for each subregion, we automatically compensate for the variation in mesh. Therefore neither the fine nor sparse finite difference problem that is solved at each iteration is adversely affected by the selected non-uniform mesh.

We also present some results for the rectangular domain \( \Omega = (0,8) \times (0,1) \). The coefficient in the differential equation is layered in the x-direction and is given by

\[
\begin{align*}
160, & \quad 0 < x < 1, \\
100, & \quad 1 < x < 2, \\
550, & \quad 2 < x < 3, \\
160, & \quad 3 < x < 4, \\
5, & \quad 4 < x < 6, \\
15, & \quad 6 < x < 7, \\
60, & \quad 7 < x < 8.
\end{align*}
\]

We choose \( Nx = 65 \) and \( 129 \) and \( Ny = 17 \) and \( 33 \). For each combination of mesh we vary the number of subdomains in the x-direction from 8, 16, to 32 and in the y-direction from

<table>
<thead>
<tr>
<th>( Nx \times Ny )</th>
<th>( Nx \times Ny )</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 x 33</td>
<td>2 x 2</td>
<td>46</td>
<td>.77</td>
</tr>
<tr>
<td></td>
<td>4 x 4</td>
<td>40</td>
<td>.37</td>
</tr>
<tr>
<td></td>
<td>8 x 8</td>
<td>34</td>
<td>.40</td>
</tr>
<tr>
<td>65 x 65</td>
<td>4 x 4</td>
<td>47</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>8 x 8</td>
<td>41</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>16 x 16</td>
<td>39</td>
<td>1.60</td>
</tr>
<tr>
<td>129 x 129</td>
<td>4 x 4</td>
<td>58</td>
<td>4.95</td>
</tr>
<tr>
<td></td>
<td>8 x 8</td>
<td>47</td>
<td>2.9</td>
</tr>
<tr>
<td></td>
<td>16 x 16</td>
<td>46</td>
<td>3.65</td>
</tr>
</tbody>
</table>
2, 4, and 8. Table 4.18 summarizes the convergence results for $a_5(x,y)$. We see some of the same patterns as before, but we also see something new. When the mesh rectangles long and thin, they began to degrade the performance of the algorithm. From studying the information given in these tables, we see that we require quasi-uniform sampling for both the fine and the coarse mesh in order to achieve the desired optimum performance of this algorithm. Simply over sampling in a given direction without justification will destroy the nice convergence properties of this method.

The above examples document and support the claims which were made in the previous chapter. They demonstrate that a wide variety of functions $a(x,y)$ are easily handled and that indeed the numerical results are proof that we do observe optimal performance for reasonable choices of parameters. From the described examples we observe that both the cpu time and the number of iterations increase almost linearly when compared with the total number of unknowns. An additional observation was the requirement of quasi-uniform meshes for both the fine and coarse mesh.

<table>
<thead>
<tr>
<th>$N_x \times N_y$</th>
<th>$N_x \times \overline{N_y}$</th>
<th>Number of Iterations</th>
<th>CPU (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 × 9</td>
<td>8 × 2</td>
<td>50</td>
<td>.27</td>
</tr>
<tr>
<td></td>
<td>8 × 1</td>
<td>46</td>
<td>.32</td>
</tr>
<tr>
<td></td>
<td>16 × 2</td>
<td>42</td>
<td>.30</td>
</tr>
<tr>
<td></td>
<td>4 × 2</td>
<td>91</td>
<td>.50</td>
</tr>
<tr>
<td>65 × 17</td>
<td>8 × 4</td>
<td>75</td>
<td>.63</td>
</tr>
<tr>
<td></td>
<td>8 × 2</td>
<td>56</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>16 × 4</td>
<td>58</td>
<td>.69</td>
</tr>
<tr>
<td></td>
<td>4 × 2</td>
<td>93</td>
<td>.94</td>
</tr>
<tr>
<td></td>
<td>16 × 2</td>
<td>51</td>
<td>.56</td>
</tr>
<tr>
<td>129 × 17</td>
<td>8 × 2</td>
<td>58</td>
<td>.8</td>
</tr>
<tr>
<td></td>
<td>16 × 2</td>
<td>52</td>
<td>.73</td>
</tr>
<tr>
<td></td>
<td>8 × 4</td>
<td>82</td>
<td>.97</td>
</tr>
<tr>
<td></td>
<td>32 × 2</td>
<td>55</td>
<td>1.05</td>
</tr>
<tr>
<td>129 × 33</td>
<td>8 × 4</td>
<td>87</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>16 × 4</td>
<td>65</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>16 × 8</td>
<td>100</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>32 × 8</td>
<td>62</td>
<td>2.0</td>
</tr>
</tbody>
</table>
CHAPTER V

CONCLUSIONS

An efficient solution of the two-dimensional self-adjoint elliptic partial differential equation via domain decomposition using vector or parallel machines has been described in this dissertation. We demonstrated that it works on many kinds of problems including those with non-uniform mesh, variable coefficients, discontinuous coefficients, and rectangular domains. We determined the optimal relationship between the fine mesh which is used to discretize the domain of integration and the number of subdomains. We were able to determine what factors inhibited or enhanced the convergence of this new numerical process. We conclude that standard block methods are better suited for small problems but that this new procedure is ideal for large problems. Finally, it was demonstrated that this method effectively utilizes vector or parallel machines with CRAY-like architecture which are designed to be efficient even for relatively short vector lengths.

One could further study the effect of utilizing additional capabilities of the CRAY-XMP. GATHER/SCATTER hardware could be used to virtually eliminate the data movement problems. One could essentially reduce the computation time by a factor of four by multi-tasking for up to four processors when the number of unknowns is large. The overhead for using the four processors would be minimal since additional data communication would not be necessary. Also, note that this would have absolutely no effect on the convergence of the algorithm since exactly the same calculations would be carried out. These considerations would shift the recommend optimal number of subdomains for a given mesh. All of these modifications would make a practical extension to three-dimensional problems.
REFERENCES


