# An Efficient Parallel Three-Level Preconditioner for Linear Elliptic Partial Differential Equations

by

Aixiang Yao

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE**

in

Computer Science



APPROVED:

Dr. Calvin J. Ribbens, Chairman

Dr. Layne T. Watson

Dr. Christopher Beattie

February, 1998
Blacksburg, Virginia

# An Efficient Parallel Three-Level Preconditioner for Linear Elliptic Partial Differential Equations

by

Aixiang Yao

Committee Chairman: Dr. Calvin J. Ribbens

**Computer Science**

## (ABSTRACT)

The primary motivation of this research is to develop and investigate parallel preconditioners for linear elliptic partial differential equations. Three preconditioners are studied: block-Jacobi preconditioner (BJ), a two-level tangential preconditioner (D0), and a three-level preconditioner (D1). Performance and scalability on a distributed memory parallel computer are considered. Communication cost and redundancy are explored as well.

After experiments and analysis, we find that the three-level preconditioner D1 is the most efficient and scalable parallel preconditioner, compared to BJ and D0. The D1 preconditioner reduces both the number of iterations and computational time substantially. A new hybrid preconditioner is suggested which may combine the best features of D0 and D1.

# ACKNOWLEDGEMENTS

I am greatly indebted to my advisor, Professor Calvin J. Ribbens for many stimulating discussions and his valuable advice throughout this work. Without his guidance, encouragement, and support, this paper would not have been possible. I would like to thank him and the other members of my committee, Dr. Watson and Dr. Beattie, for their guidance and assistance throughout my graduate studies.

Finally, I would like to thank my friends and my family for being there when I needed them.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# Introduction

As the development of science and technology continues its rapid race, the need for increased computational power for many applications is becoming more and more urgent. Progress in computer science and engineering helps to meet this need for computational power. The dramatic improvement in chip density together with an increase in clock speed will continue to reduce the feature size of integrated circuits. However, faster clock rates alone are insufficient to meet the needs of the largest problems, and it is extremely unlikely that clock rates will continue to improve at the present rate for much longer anyway (the "speed of light" problem). An obvious possible solution is parallelism. Eventually, many experts believe that parallelism will be present in all computers ([Fox95]) and that the long-term solution for the largest problems is massive parallelism.

Today we see parallelism in several large machines as many chips and printed circuit boards are replicated to build systems as arrays of computing nodes, each of which is a single chip CPU with additional memory modules. The Intel Paragon is an example of this kind of architecture. Moreover, new software and libraries dealing with parallel computing and interprocess communication are coming up like spring sprouts. On the Paragon alone, a sophisticated user may need to become familiar with NX, MPI [Gropp94],[Snir95], PVM [Geist94], HPF [Koelbel94], PIM [Cunha], ParaGraph, SPV, etc. Many research groups are active in this area as well. It seems likely that parallelism will become more and more important as a direction of development for computers in the next century. In fact, parallelism is already a widely used tool in scientific computing, especially in numerical linear algebra (see [Ortega88], [Dongarra91], [Gallivan90]). We refer the reader to the following for good general surveys of parallel computing: [Foster95], [Fox95], [Kuck96],[Toole95].

## Domain Decomposition for PDEs

An important source of large scientific computing problems is the numerical solution of elliptic partial differential equations (PDEs). Elliptic PDEs are often used to model the steady state behavior of some physical system. The reader is referred to [Birkhoff84] for an introduction to numerical methods for elliptic PDEs. A common approach for designing a parallel algorithm for the numerical solution of an elliptic PDE is "domain decomposition" (DD). Domain decomposition is basically a "divide and conquer" idea. Dividing the physical domain into regions called subdomains with interfaces between the subdomains, or distributing a huge amount of data among processors, or subdividing the solution of a large system into a set of smaller problems, are all examples of domain decomposition. Domain decomposition techniques are an intuitive organizing principle for a parallel PDE computation. However, the significance of domain decomposition extends beyond the readily apparent issues of discretization and geometry, and of modular software and hardware on distributed memory computers.

Early domain decomposition techniques were either direct or iterative, based on reductions of the global problem to a set of smaller problems. Many fast direct domain decomposition solvers have been developed and reported in the engineering literature. For a survey of research in DD methods, see [Smith96], [Chan94], and [Keyes95]. More recent developments have been based on an iterative approach which is potentially more efficient in both time and storage. Theoretically, domain decomposition methods can be viewed either as preconditioners for Krylov subspace methods for linear problems, or as preconditioners for the solvers of the linear systems arising from Newton's method for nonlinear problems. The focus of this research is on DD as a preconditioner for Krylov subspace iterative methods for linear problems.

An appropriate preconditioner needs to be chosen to make a DD computation more efficient. An ideal DD preconditioner would:

- achieve a high convergence rate (accelerate the convergence rate of the Krylov method);

- reduce the computation time;
- not require a prohibitive amount of memory storage;
- balance the work load well;
- allow high utilization and good system performance;
- achieve good scaled parallel speedup (depending on how many processors are involved and how the problem scales as the number of nodes is increased).

Obviously, there can be tradeoffs among the above considerations, and the application scientist is typically only interested in solving a given problem (much) faster or in solving a (much) bigger problem in "reasonable" time.

### Goals and organization of the thesis

The primary motivation for this research is to develop and test efficient parallel preconditioners for linear elliptic partial differential equations. Secondary goals are to get experience with real implementations of such algorithms on real parallel systems, and experience in doing systematic studies of the performance of these implementations. The type of DD preconditioner that we study is based on a preconditioner proposed by George Pitts. In his Ph.D. dissertation [Pitts95], Pitts proposed a "three-level" preconditioner based on three different grids: the original fine grid, a coarse grid, and a hybrid fine/coarse grid. Pitts was working in the context of linear systems arising from high order (HODIE [Lynch78]) finite difference methods. He implemented and tested his algorithm primarily on a sequential machine, although he does report some performance results on a parallel shared memory machine. In this research, we modify and extend Pitts' work for a parallel distributed memory machine, combine our preconditioned solver with standard second-order finite difference discretization, and consider scalability.

The major hardware and software components of our research are:

- Intel Paragon. We run our experiments on a parallel distributed memory machine, the Intel Paragon.
- ELLPACK. We store our matrix in ELLPACK storage format, distributing the entire matrix among the processing nodes involved. We also encorporate our code in the ELLPACK system, taking advantage of its parallel finite difference module.
- MPI. To achieve message passing in a portable way, MPI is used.
- PIM. The Parallel Iterative Methods package is a set of parallel implementations of several well-known iterative solvers, e.g., conjugate gradient, GMRES, etc. We test preconditioners using PIM's implementation of GMRES.

The organization of this thesis is as follows. Chapter 2 describes the hardware and software environment that we are in, including the Intel Paragon, ELLPACK, and iterative solvers. Chapter 3 discusses the basic concept of domain decomposition and preconditioners, as well as parallelization. Chapter 4 describes our experiments, results, and analysis. Chapter 5 summarizes our main results, and adds some comments and further research interests.

# Chapter 2

# Hardware and Software Environment

In this chapter, we describe the hardware and software environment that we use for our experiments, including the Intel Paragon, MPI, ELLPACK, an iterative linear equation solver, and the parallel version of that iterative solver. Our purpose is to investigate how good the Pitts three-level preconditioner is as a parallel algorithm implemented on a distributed memory machine. Also we explore the scalability of the three-level parallel preconditioner, compared to one-level (block Jacobi) and two-level (tangential) preconditioners. Details of these preconditioners are given in Chapter 3. We use parallel extensions to the ELLPACK problem solving environment. The two-dimensional problem domain is decomposed into subdomains either in one dimension (horizontal strips or vertical strips) or in two dimensions. For the linear system solution, we integrate the parallel iterative methods package (PIM) of da Cunha and Hopkins [Cunha94] into the ELLPACK computational framework. The PIM library provides flexibility, allowing the user to implement numerous methods in terms of the following basic linear algebra operations: matrix-vector product, preconditioner application, inner product and vector sum. These operations can be implemented in parallel using the MPI library.

## 2.1 Intel Paragon

We first take a brief look at the Intel Paragon on which we conducted our experiments. The Intel Paragon supercomputer is a distributed memory multicomputer. It consists of a large number of independent processors called nodes, each with its own memory and network interface, linked by communication channels. The nodes are physically interconnected in a

two-dimensional mesh. Logically, however, any node can communicate with any other node using worm-hole routing. A set of nodes can work concurrently on independent parts of a problem.

Each node on the Paragon has its own operating system which can be considered as permanently resident. Each node can run multiple processes, and each process can have multiple threads. Each process can be a stand-alone program, such as an editor, compiler, debugger, or shell, or can be a part of a parallel application. On each node, processes and threads time-share the node's processor by a scheduling mechanism.

A Paragon can be configured with thousands of nodes connected in a two-dimensional mesh (it is a scalable architecture). The nodes can be classified into three main categories: service nodes, I/O (input and output) nodes, and compute nodes.

- The interactive processes, such as user shell and editor, run on the service nodes. A user edits files, compiles and runs applications (including network applications) while logged in to one of the service nodes.
- All the input/output jobs are handled by I/O nodes.
- Compute nodes are used for compute-intensive parallel tasks. The Paragon at Virginia Tech, on which we ran our experiments, has 100 compute nodes, each with 32MB of RAM.

A parallel application consists of a group of related processes that work together on a single problem. Synchronization of processes and information sharing can be achieved by message passing, which is created and controlled by special system calls, or by calls to message passing libraries. There are a number of libraries available for message passing on the Paragon system:

- The native **NX** message passing library provided by Intel offers the best message passing performance for the Paragon system.

- The **MPI** (Message Passing Interface) library [Gropp94] is designed for high performance on both massively parallel (homogeneous) machines, such as the Paragon, and heterogeneous networks. Our work is based on MPI.
- **PVM** (Parallel Virtual Machine) [Geist94] is a software package that permits a heterogeneous collection of Unix computers hooked together by a network to be used as a single large parallel computer.

## 2.2 MPI

MPI (Message Passing Interface) [Gropp94] is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. MPI provides a standard interface to message passing procedures for parallel computers and networked workstations. To perform complicated communications among processors, programmers can use MPI routines instead of the message passing functions provided by any individual computer. In this way, program portability is achieved. This is the motivation for MPI. MPI has many useful features, such as point-to-point communication, collective operations, and application-oriented process topologies. Additionally, MPI offers many useful functions that allow the message passing to be performed easily. As an example, we list here some functions used in our implementation and experiments for this thesis:

- MPI_SEND. Sends a message from one processor to another processor.
- MPI_RECV. Receives a message.
- MPI_SENDRECV. Sends and receives messages. This function combines two functions together: MPI_SEND and MPI_RECV.
- MPI_GATHER. A specified "root" node gathers portions of an array from each processor and concatenates them into another array.
- MPI_ALLGATHER. Similar to MPI_GATHER except each processor gets a copy of the concatenated (gathered) array.

- MPI_ALLREDUCE. Gathers data from each processor and performs a "reduction" operation such as SUM, MAX, PRODUCT. All processors get a copy of the result.

## 2.3 ELLPACK

ELLPACK [Rice85] is a software system for solving linear elliptic boundary value problems which includes both a very high-level problem statement language and an extensible library of problem-solving modules. For this research, we used a version of ELLPACK that was extended to take advantage of a distributed memory parallel computer. This MPI-based extension includes a parallel 2-D finite difference discretization module.

The ELLPACK language is an extension of FORTRAN77, and ordinary FORTRAN77 statements can be mixed with ELLPACK statements. Elliptic equations, general 2-D domains, 3-D boxes, and boundary conditions can be stated directly in this language, and powerful executable statements are available which select predefined modules for each stage of the solution process. For a comprehensive description of ELLPACK, please refer to the book by Rice and Boisvert [Rice85]. Figure 2.1 shows the schematic diagram of the processing of an ELLPACK run ([Rice85], Figure 1.1).

Next, let us see how ELLPACK stores matrix data, and what the matrix data structure used in ELLPACK looks like. In ELLPACK, the matrix $A$ and vector $b$ of the linear system $Ax = b$ are generated by a discretization module. The nonzero equation coefficients and column indexes are stored in the two-dimensional real and integer arrays, COEF and IDCO, respectively. Both of these arrays are of dimension N by MAXNZ, where N is the number of linear equations and unknowns and MAXNZ is the maximum number of nonzeros per row.

The coefficients of a particular equation and its column indexes are stored in corre-

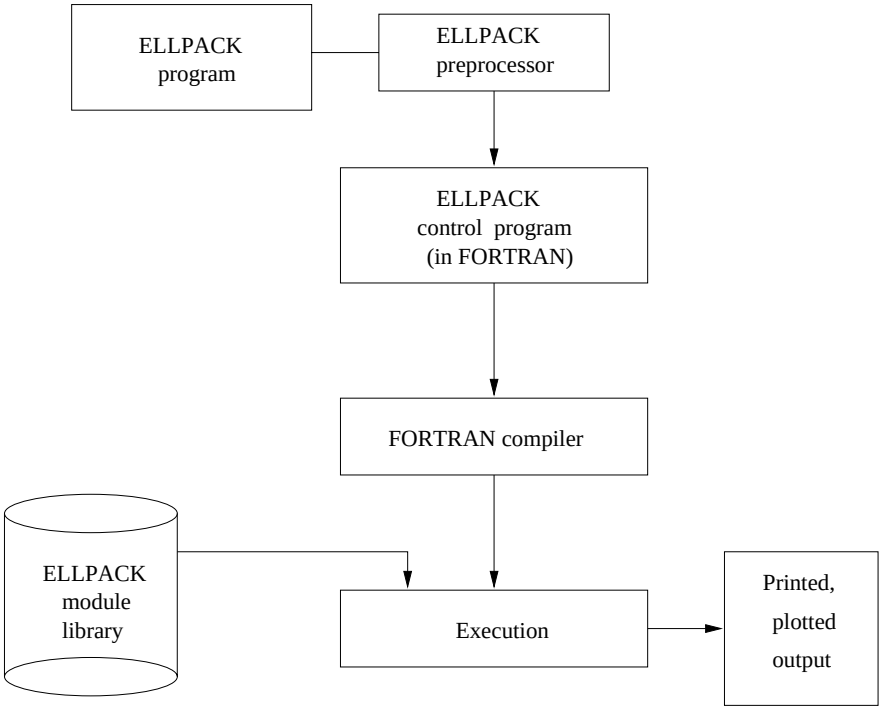



Figure 2.1: Schematic diagram of the processing of an ELLPACK run.

sponding rows of the matrices. For example, the coefficient matrix

$$A = \begin{pmatrix} 4. & -1. & -2. & 0. \\ -1. & 4. & 0. & -2. \\ -3. & 0. & 4. & -1. \\ 0. & 0. & -1. & 4. \end{pmatrix}$$

could be represented by

$$COEF = \begin{pmatrix} 4. & -1. & -2. \\ 4. & -1. & -2. \\ 4. & -3. & -1. \\ 4. & -1. & 0. \end{pmatrix}, \quad IDCO = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 1 & 4 \\ 4 & 3 & 0 \end{pmatrix}.$$

Notice that $COEF(I, J) = A(I, IDCO(I, J))$. The nonzero coefficients in a particular row are not necessarily left justified and may be in any order. Usually the diagonal element is

placed in column one of COEF, though this is not required.

The ELLPACK data structures are very efficient both in storage and computation time, for some types of sparse matrices. Each ELLPACK discretization module approximates the partial differential equation and boundary conditions by a finite system of linear equations, and the data (coefficient and indexing matrices) are stored into these data structures.

### Five-point star discretization

The most common finite difference discretization scheme for second order elliptic PDEs is often called *five-point star*. To approximate derivatives of the unknown $u$ at grid point $(x_i, y_j)$, in the process of discretization, only five values of $u(x, y)$ at grid points $(x_{i-1}, y_j)$, $(x_i, y_j)$, $(x_{i+1}, y_j)$, $(x_i, y_{j-1})$, $(x_i, y_{j+1})$ are involved. If we draw a digram of these five points, it looks like a star, as shown in the following:

| | | |
|---|---|---|
| | $U_{i,j+1}$ | |
| $U_{i-1,j}$ | $U_{i,j}$ | $U_{i+1,j}$ |
| | $U_{i,j-1}$ | |

In the above diagram, $U_{i,j}$ represents the approximation to $u(x, y)$ at grid point $(x_i, y_j)$. It is clear why this discretization method is called *five-point star*. If the four corner positions are added and nine points are involved, the scheme for discretization is called *nine-point stencil*. There are many other discretization stencils.

Next let us give an example to see more clearly how the five-point star stencil works. We consider the following linear elliptic PDE:

$$au_{xx} + 2bu_{xy} + cu_{yy} + du_x + eu_y + f = g, \tag{2.1}$$

where $a, b, c, d, e$, and $f$ are constant coefficients, or coefficient functions, and $g$ is a given function. We illustrate the procedure by considering the case of a rectangular domain $R$. The unknowns $U_{i,j}$ are approximations to $u(x_i, y_j) = u_{i,j}$, $i = 1, \ldots, n,\ j = 1, \ldots, m$. In the

case of equally spaced grid points, the finite difference equation approximating Equation (2.1) at the point $(x_i, y_j)$ is based upon the following finite difference approximations:

$$u_{xx}(x_i, y_j) = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + O(h^2), \quad u_{yy}(x_i, y_j) = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} + O(k^2),$$

$$u_{xy}(x_i, y_j) = \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4hk} + O(hk),$$

$$u_x(x_i, y_j) = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2), \quad u_y(x_i, y_j) = \frac{u_{i,j+1} - u_{i,j-1}}{2k} + O(k^2),$$

where $h$ and $k$ are the grid spacings in the $x$ and $y$ directions, respectively. The above approximations are $O(h^2 + k^2)$ accurate. Writing the approximation of $u_{xx}$ into a stencil form, we have

$$u_{xx}(x_i, y_j) \approx \frac{1}{h^2} \begin{array}{|c|c|c|}\hline & & \\ \hline 1 & -2 & 1 \\ \hline & & \\ \hline \end{array} U,$$

a convolution of a mask with $U$ centered at the point $(x_i, y_j)$. The stencil forms for the other terms are similar to the above. These forms are put together, and then the approximation of Equation (2.1) can be written as

$$\left( \frac{a}{h^2} \begin{array}{|c|c|c|}\hline & & \\ \hline 1 & -2 & 1 \\ \hline & & \\ \hline \end{array} + \frac{2b}{4hk} \begin{array}{|c|c|c|}\hline -1 & & 1 \\ \hline & & \\ \hline 1 & & -1 \\ \hline \end{array} + \frac{c}{k^2} \begin{array}{|c|c|c|}\hline & 1 & \\ \hline & -2 & \\ \hline & 1 & \\ \hline \end{array} \right.$$

$$\left. + \frac{d}{2h} \begin{array}{|c|c|c|}\hline & & \\ \hline -1 & & 1 \\ \hline & & \\ \hline \end{array} + \frac{e}{2k} \begin{array}{|c|c|c|}\hline & 1 & \\ \hline & & \\ \hline & -1 & \\ \hline \end{array} + f \begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline \end{array} \right) U = g.$$

This schematic represents one equation, where everything is interpreted as being evaluated at a grid point $(x_i, y_j)$.

Or, equivalently,

$$\begin{array}{|c|c|c|}\hline \frac{-b}{2hk} & \frac{c}{k^2}+\frac{e}{2k} & \frac{b}{2hk} \\ \hline \frac{a}{h^2}-\frac{d}{2h} & f-\frac{2a}{h^2}-\frac{2c}{k^2} & \frac{a}{h^2}+\frac{d}{2h} \\ \hline \frac{b}{2hk} & \frac{c}{k^2}-\frac{e}{2k} & \frac{-b}{2hk} \\ \hline \end{array} U = g. \tag{2.2}$$

We focus on linear elliptic partial differential equations, and assume that $b \equiv 0$, and $ac > 0$. If $b = 0$, our stencil requires only five points. The approximation at $(x_i, y_j)$ takes the form

$$\begin{array}{|c|c|c|}\hline & \frac{c}{k^2}+\frac{e}{2k} & \\ \hline \frac{a}{h^2}-\frac{d}{2h} & f-\frac{2a}{h^2}-\frac{2c}{k^2} & \frac{a}{h^2}+\frac{d}{2h} \\ \hline & \frac{c}{k^2}-\frac{e}{2k} & \\ \hline \end{array} U = g, \tag{2.3}$$

interpreted as a convolution of a stencil (mask) with $U$ at $(i, j)$.

Notice that if we use a different stencil to approximate each term of the equation we end up with a different discretization stencil than five-point star. For high-order, more accurate discretization schemes, the reader can refer to HODIE [Lynch78], or HODIEX [Pitts95]. In these cases, higher accuracy can be expected. Also they involve more points for discretization and/or more evaluations of the right hand side $g$.

## 2.4 Iterative solver

We used GMRES (more exactly GMRES($k$)) as an iterative linear equation solver to conduct our experiments. The Parallel Iterative Methods package (PIM 2.1 [Cunha]) is a package designed for parallel linear equation solving. PIM is integrated into the parallel ELLPACK computational framework that we used.

### 2.4.1 PIM

The Parallel Iterative Methods package (PIM) is a collection of Fortran 77 routines to solve symmetric and nonsymmetric systems of linear equations on parallel computers using a variety of iterative methods. These methods include conjugate-gradient (CG), bi-conjugate-gradient (Bi-CG), conjugate-gradient squared (CGS), generalized minimal residual (GMRES), generalized conjugate residual (GCR), quasi-minimal residual (QMR), etc. Please refer to [Cunha] for details of PIM and also [Saad96] for an introductory survey of iterative methods. The PIM routines allow the use of user-supplied preconditioners. The user may choose to supply left-, right- or symmetric-preconditioning. Several stopping criteria are available in the PIM package.

Although many vendors supply their own linear algebra routines for use, the user may need to provide routines to compute the following three linear algebra operations:

1. Matrix-vector product $Mv$ and/or transpose-matrix-vector product $M^T v$.

2. Preconditioning step.

3. Vector norm and/or inner-products.

In this way, PIM can achieve portability across a variety of parallel machines and environments, and also give the user freedom to choose ways of matrix storage, access and partition. However we need to provide routines for 1) Matrix-vector product, 2) preconditioning, and 3) norm (inner product), as described. We will choose the preconditioner as defined in Chapter 3, use tools in the MPI library to define Matrix-vector product, and utilize the default tools in PIM for norms.

### 2.4.2 GMRES

The GMRES method [Saad86] is a robust method for solving nonsymmetric linear systems. The method uses the Arnoldi process to compute an orthonormal basis $\{v_1, v_2, \ldots, v_k\}$ of the Krylov subspace $\mathcal{K}(A, v_1)$. The solution of the system is taken as $x_0 + V_k y_k$ where $x_0$

is an initial guess, $V_k$ is a matrix whose columns are the orthonormal vectors $v_i$, and $y_k$ is the solution of the least-square problem $H_k y_k = \|r_0\|_2 e_1$. Here the upper Hessenberg matrix $H_k$ is generated during the Arnoldi process and $e_1 = (1, 0, \ldots, 0)^T$. This least-squares problem can be solved using a QR factorization of $H_k$.

For a description of the Arnoldi process, we refer the reader to Saad's book [Saad96, pp 147]. Arnoldi's procedure is an algorithm for building an orthogonal basis of the Krylov subspace $\mathcal{K}_m$ (see Figure 2.2).

Figure 2.2: Algorithm: Arnoldi

1. Choose a vector $v_1$ of norm 1
2. For $j = 1, 2, \ldots, m$ Do:
3. Compute $h_{ij} = (Av_j, v_i)$ for $i = 1, 2, \ldots, j$
4. Compute $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$
5. $h_{j+1,j} = \|w_j\|_2$
6. If $h_{j+1,j} = 0$ then Stop
7. $v_{j+1} = w_j / h_{j+1,j}$
8. EndDo

The GMRES iterates are constructed as

$$x_i = x_0 + y_1 v_1 + \ldots + y_i v_i,$$

where the coefficients $y_k$ have been chosen to minimize the residual norm $\|b - Ax_i\|$. The GMRES algorithm has the property that this residual norm can be computed without the iterate having been formed.

The problem with GMRES is that all of the previous iterates must be kept for the calculation of the next iterate, and that can increase storage requirements greatly when a large number of iterations are needed.

However, storage requirements are reduced by using a *restarted* version of GMRES. This means that after a certain number of iterations, say 10 or 20, the iteration is restarted by using the most recent approximation as the initial guess.

### 2.4.3 Parallel GMRES

The Parallel Iterative Method (PIM) package, designed to solve systems of linear equations on parallel computers, offers a number of iterative methods, including GMRES and RGMRES (restarted GMRES, or GMRES($m$) as it more commonly known). The purpose of GMRES($m$) is to solve the system $Q_1AQ_2x = Q_1b$ using the restarted GMRES method, where $Q_1$ is a left-preconditioner and $Q_2$ is a right-preconditioner.

Most of the computation in GMRES(m) can be done in parallel, as long as the user provides parallel versions of matrix-vector product, preconditioner application and vector-norm. If we distribute the global matrix among the processors, we can do matrix-vector product and vector-norm in parallel by computing partial matrix-vector products and vector-norms and then collecting the results together. However, the process of collecting and reducing data can be a potential bottleneck. Since the DD preconditioners are the focus of this research, we do not consider further the parallel performance of the rest of the GMRES(m) algorithm.

# Chapter 3

# Domain Decomposition and the Preconditioners

This chapter covers the idea of preconditioning and of domain decomposition (DD). We will describe three preconditioners: a block jacobi preconditioner (BJ), a two-level, or *tangential*, preconditioner (D0), and a three-level, or *hybrid coarse/fine mesh*, preconditioner (D1). Also we will discuss parallelization of the preconditioners.

## 3.1 Preconditioning

We consider a nonsingular linear system $Ax^* = b$. Sometimes it is not practical to compute the exact solution $x^*$ either because it is difficult to get the inverse matrix of $A$, $A^{-1}$, or because $A$ is ill-conditioned. In these cases, a better approach is to use preconditioners. Instead of solving the original equation, we solve a non-singular linear system equivalent to the original equation,

$$Q_1 A Q_2 y = Q_1 b, \quad y = Q_2^{-1} x, \tag{3.1}$$

for the general case, or solve a Hermitian positive definite (HPD) system

$$Q_1 \hat{A} \hat{A}^T Q_2 y = Q_1 b,$$

for a symmetric case with $\hat{A}\hat{A}^T = A$ and $Q_2 = Q_1^T$, where $Q_1$ and $Q_2$ are the *preconditioning matrices*.

**Definition 1.** Both $Q_1$ and $Q_2$ in (3.1) are **preconditioners**. $Q_1$ and $Q_2$ are called a *left-preconditioner* and *right-preconditioner*, respectively.

### Alternative definition

Some authors define preconditioner in a different form. Let $x$ be an approximate solution to the equation $Ax^* = b$, and $e = x^* - x$ be the error. Then

$$Ae = A(x^* - x) = b - Ax = r.$$

Assume we have an algorithm to approximate the error

$$e \approx B_1 r.$$

Thus the corrected new approximate solution is given by

$$x^{new} \leftarrow x + B_1 r. \tag{3.2}$$

**Definition 2.** The matrix operator $B_1$ (or sometimes $B_1^{-1}$) is referred to as the **preconditioner**, sometimes it is also called an **approximate solver** (more exactly defined as a **left-preconditioner**).

In order to accelerate the convergence rate, one can improve the method by adding a factor to the preconditioner

$$x^{new} \leftarrow x + \tau\, B_1(b - Ax).$$

The scalar $\tau$ is a parameter called **a damping factor** or **accelerator**, which may be chosen to improve convergence.

In fact, the two definitions for preconditioners are equivalent.

**Proposition:** Assume that the iteration defined by (3.2) converges. Then the preconditioner in Definition 2 is equivalent to a left-preconditioner in Definition 1.

**Proof of equivalence:** By assumption, the iteration defined by

$$x^{new} \leftarrow x + B_1(b - Ax)$$

converges. Hence, its limit (the true solution) is a fixed point satisfying

$$x = x + B_1(b - Ax).$$

This is equivalent to

$$B_1(b - Ax) = 0, \quad i.e., \quad B_1(Ax) = B_1 b.$$

Therefore the preconditioner $B_1$ in Definition 2 is exactly the same as the left-preconditioner $Q_1$ in Definition 1 if we choose $Q_1 = B_1$ and $Q_2 = I$, where $I$ represents the identity matrix.

Similarly, each left-preconditioner $Q_1$ in Definition 1 acts the same as a preconditioner in Definition 2 if we reverse the proof process.

### 3.1.1 Examples

**Example 1. Jacobi preconditioner.**

*Jacobi* iteration is defined for matrices that have nonzero diagonal elements. The method can be motivated by rewriting the nonsingular linear system $Ax = b$ as

$$(D + L + U)x = b,$$

where $D$ is the diagonal matrix consisting of diagonal elements of $A$, $L$ is the strictly lower triangle, and $U$ the strictly upper triangle of $A$. The above equation is equivalent to

$$Dx = -(L + U)x + b.$$

Suppose $x^{(k)}$ is an approximation to $x = A^{-1}b$. A natural way to generate a new approximation $x^{(k+1)}$ is to compute

$$Dx^{(k+1)} = -(L + U)x^{(k)} + b. \tag{3.3}$$

Applying $D^{-1}$ from the left (left-preconditioner) on both sides yields

$$x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}) = x^{(k)} + D^{-1}(b - Ax^{(k)}), \tag{3.4}$$

i.e.,

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, \ldots, n.$$

This defines the Jacobi iteration. Here the left-preconditioner $D^{-1}$ is used. That is why $D^{-1}$ is usually called the **Jacobi preconditioner**. Also according to Definition 2, the preconditioner in this example is $D^{-1}$.

**Example 2. Gauss-Seidel preconditioner.** The Gauss-Seidel method is motivated by revising the Jacobi method and using the most recently computed results of the iteration. This can be done by rewriting the linear system $Ax = b$ into

$$(D + L)x = -Ux + b \tag{3.5}$$

and generating a new approximation $x^{(k+1)}$ via the formula

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

From this, we get

$$x^{(k+1)} = D^{-1}(b - Lx^{(k+1)} - Ux^{(k)}),$$

which can be rewritten as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, \ldots, n.$$

This is the well-known Gauss-Seidel iterative algorithm.

Multipling $(D + L)^{-1}$ from the left on both sides of equation (3.4) yields

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

which can be rewritten as

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

Thus

$$x = x + (D + L)^{-1}(b - Ax).$$

It is clear by Definition 2 that $(D+L)^{-1}$ is used as a left-preconditioner. The preconditioner $(D + L)^{-1}$ is usually called the **Gauss-Seidel preconditioner**.

### 3.1.2 Left and right preconditioning

**Left preconditioning.** To apply a left preconditioner $Q_1$, we begin with the equation

$$Q_1 Ax = Q_1 b.$$

Given a vector $u$, an iterative method (e.g., CG, GMRES) typically requires us to compute the vector

$$Q_1 Au = v.$$

To this end, we need

- one matrix-vector product $y = Au$ and
- one system solution $Bv = y$, where $B \equiv Q_1^{-1}$ is given,

per iteration. $B \equiv Q_1^{-1}$ is used for the left preconditioning.

**Right preconditioning.** With right preconditioning, we start from the equation

$$AQ_2 y = b, \quad y = Q_2^{-1} x.$$

In this case, we compute

$$AQ_2 u = v.$$

With an iterative method, we require

- one solution of $u = Q_2^{-1} z$ for $z$ and
- one matrix-vector product $v = Az$

per iteration.

To solve (3.1) with an iterative method (e.g., CG, GMRES) requires one matrix-vector multiplication with $A$ and one system solution with $Q_1^{-1}$ ( or $Q_2^{-1}$) per iteration. For our purpose, we use $B = Q_1^{-1}$ for the left preconditioning, and $B = Q_2^{-1}$ for the right preconditioning, where $B \approx A$. We will refer to $B$ as the preconditioner.

The preconditioners BJ, D0, D1, which we are going to address later, may be applied on the left or right as long as the basic iterative method does not require the preconditioned matrix to be symmetric.

## 3.2 Domain decomposition

As described in Chapter 1, domain decomposition (DD) refers to a "divide and conquer" idea. However the term "domain decomposition" has slightly different meanings.

- Distribution of data among processors (distributed computing). Distribute data among processors in a distributed memory computer, or on a network of workstations. For example, partition the entire matrix for a large linear system into parts, store them among processors, distribute the computation among processors, and do the computation in parallel. These techniques are basically decomposition of data structures.

- Partition of physical domain (modeling). Divide the physical domain into subdomains with interfaces between the subdomains. The union of these subdomains is the entire physical domain. These subdomains can be either overlapped or nonoverlapped.

- Subdivision of the solution (methods and algorithm, more exactly preconditioning). Subdivide the solution of a large linear system into smaller problems, and solve these smaller problems in parallel with necessary interfaces and communication between processors. This use of the term "domain decomposition" refers specifically to preconditioning methods.

Early domain decomposition techniques were either direct or iterative, based on reductions of the global problem to a set of smaller problems. The more recent developments have been based on an iterative approach which is potentially more efficient. Theoretically, domain decomposition methods can be viewed either as preconditioners for Krylov subspace methods for linear problems, or as preconditioners for the solvers of the linear systems arising from nonlinear problems.

DD methods can be divided into two classes, overlapping domain methods and nonoverlapping domain methods. Overlapping methods are often referred to as **Schwarz methods**, after an early iterative domain decomposition technique proposed by H.A. Schwarz, in his

pioneering work [Schwarz1870]. Schwarz methods have been studied further by many authors (see [Smith94], [Chan94]). On the other hand, nonoverlapping methods are usually referred to as **substructuring** or **Schur complement methods**. Many Schur complement DD solvers have been described in the engineering literature (see [Kron53], [Przem63]).

## 3.3 Block Jacobi preconditioner (BJ)

The simplest preconditioner consists of just the diagonal elements of the matrix $A$:

$$\tilde{A}_{ij} = \begin{cases} a_{ij}, & \text{if } i = j \ ; \\ 0, & \text{otherwise} \ . \end{cases} \tag{3.6}$$

This is known as the (point) Jacobi preconditioner; it was derived in Section 3.1.

A natural generalization of Jacobi is to partition the equations and unknowns into blocks and update all the unknowns in a block simultaneously by solving a small linear system associated with just those unknowns. Rather than solving the subproblems exactly, it may make sense to solve them iteratively to some tolerance. If the index set of the unknowns is given by $S = \{1, \ldots, n\}$, and $S$ is partitioned into blocks as $S = \bigcup_i S_i$ with the sets $S_i$ mutually disjoint, then one can define a block Jacobi preconditioner as

$$\tilde{A}_{ij} = \begin{cases} a_{ij}, & \text{if } i \text{ and } j \text{ in the same index subset;} \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

The block Jacobi (**BJ**) preconditioner is now a block-diagonal matrix. A good description of block Jacobi iteration and the block Jacobi preconditioner can be found in [Saad96].

Often, natural choices for partitioning equations and the unknowns are the following (see [Barrett94]):

- In problems with multiple physical unknowns per node, blocks can be formed by grouping the equations and unknowns at each node.
- In problems representing a physical domain, a partitioning can be based on a natural decomposition of the physical domain.

- On parallel computers, it is natural to let the partitioning coincide with the distribution of unknowns over the processors.

In our experiments, we combine the second case with the third case: partitioning is based on a partition of the global physical domain into subdomains and a distribution of subdomains over the processors.

Jacobi preconditioners and block-Jacobi preconditioners need very little storage unless the block size becomes extremely large and the subproblems are solved directly; and they are easy to implement.

## 3.4 A two-level preconditioner (D0)

A two-level preconditioner that we study is the **tangential preconditioner**, which we call **D0** for convenience. In tangential preconditioning, two types (levels) of grids—a fine grid and a coarse grid—are involved. That is why it is called a two-level preconditioner. As we shall see, D0 is called tangential because part of the preconditioner is based on approximating the PDE only in the direction tangent to a subdomain interface.

In this section, we discuss the tangential idea underlying the D0 preconditioner. Then we investigate the matrix structure for the D0 preconditioner. These discussions can be applied to the next preconditioner (D1) that we are going to study. This preconditioner (D0) and the next (D1) can be easily described in terms of the physical domain and mesh. Thus let us look at an example first.

Suppose that the domain is the unit square and we have Dirichlet boundary conditions. Choose a uniform grid, with $n_x = n_y = 25$ grid lines in each directions. We decompose the domain into $4 \times 3$ subdomains ($N_x \times N_y$) as shown in Figure 3.1. These subdomains are physical subdomains, and can be viewed as a coarse grid over the original problem domain. Notice that $h_x = 1/(n_x - 1)$, $h_y = 1/(n_y - 1)$ for the fine grids, $H_x = 1/N_x$, $H_y = 1/N_y$ for the coarse grids, where $N_x$ is the number of subdomains in the $x$-direction, and $N_y$ is the number of subdomains in the $y$-direction. In this example, $h_x = h_y = 1/24$ for the fine

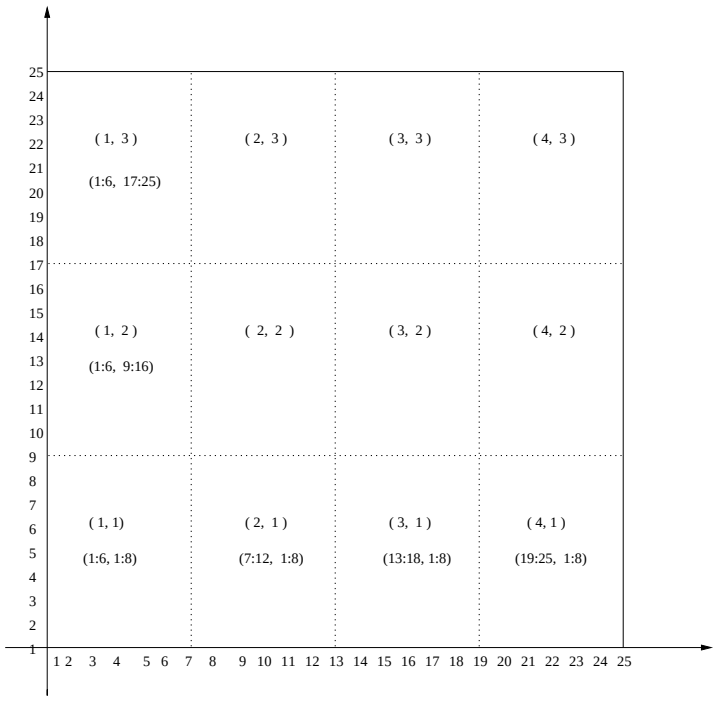



Figure 3.1: Physical subdomains.

grids, $H_x = 1/4, H_y = 1/3$ for the coarse grids.

A natural way to number interior grid points is from the left to the right, and from the bottom to the top. We usually call this the **natural numbering**. This is shown in Figure 3.2. There are three types of points: cross points, interface points, interior points.

**Cross points.** Cross points are those points at the corners of a subdomain but not on the natural boundary. They are in the intersection of two coarse grid lines.

**Interface points.** Interface points are those points on a coarse grid line (interface line) between two subdomains, which are not a cross point, nor on the natural boundary.

**Interior points.** Interior points are points in a subdomain but not on the boundary of the subdomain.

In order to better describe the logical structure of our preconditioners, we number the unknowns in a different way. 1) Start to number interior points first in a subdomain, from the left to the right, from the bottom to the top, then move to the next subdomain and

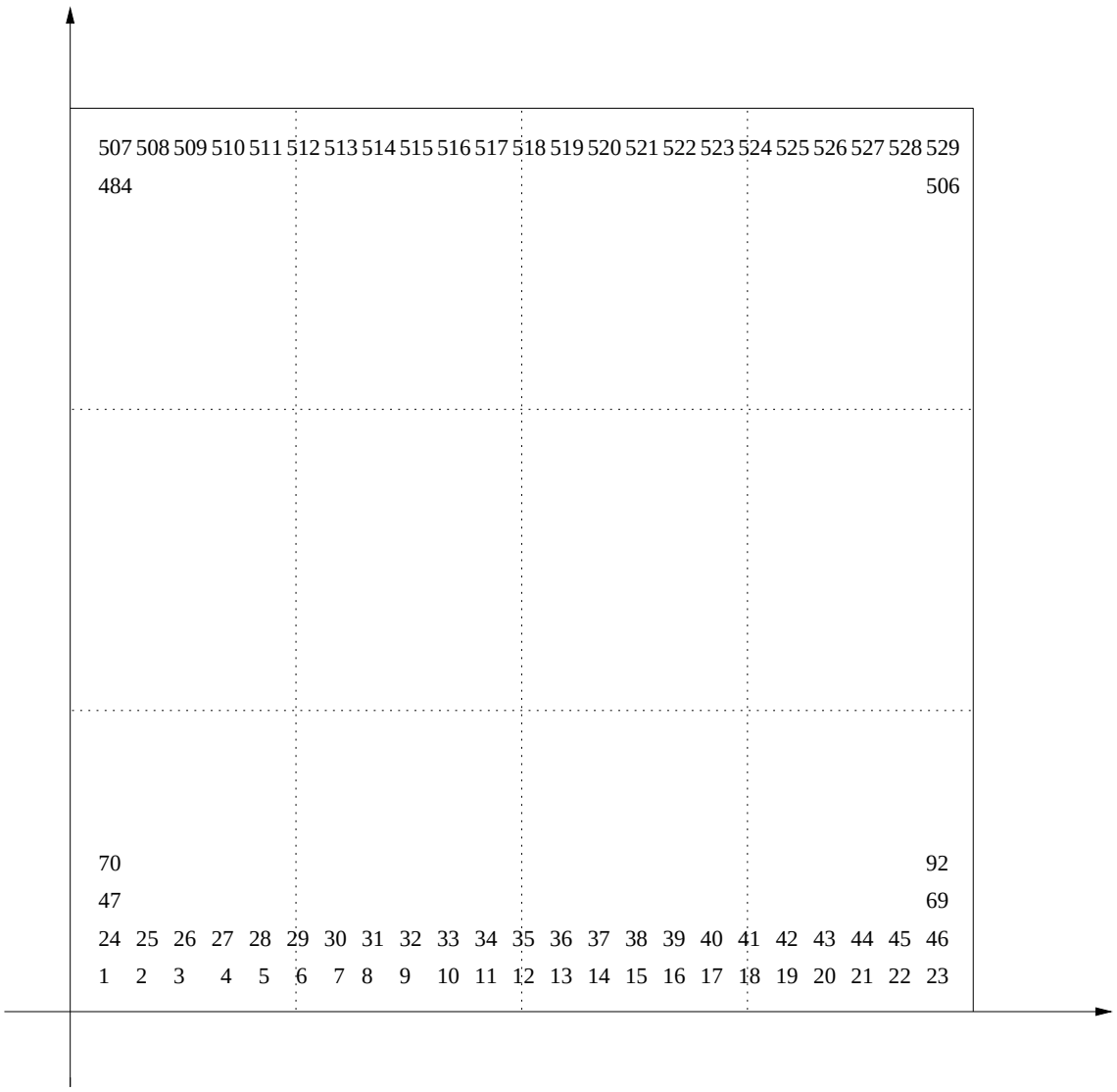



Figure 3.2: Natural numbering of unknowns.

number its interior points, and repeat until all the interior points are numbered. 2) Number interface points second. 3) Finally number cross points. We will call this numbering method **DD numbering**. Figure 3.3 displays the DD numbering for our example.

The global coefficient matrix after the DD numbering of equations and unknowns has the form

$$A = \begin{pmatrix} A_I & A_{IB} & 0 \\ A_{BI} & A_B & A_{BC} \\ 0 & A_{CB} & A_C \end{pmatrix}, \tag{3.8}$$

where $A_I$ is the matrix corresponding to the interior points, $A_B$ represents the interface (or "boundary") points, and $A_C$ corresponds to the cross points. The off-diagonal matrices $A_{IB}$ and $A_{BI}$ correspond to interactions between interior and interface points, while $A_{BC}$ and $A_{CB}$ represent interactions between interface and cross points. Note that the two zero blocks in (3.8) are due to the five-point star stencil. These blocks would be nonzero with nine-point stencil.

In the case of a one-dimensional domain decomposition (e.g., either $N_x$ or $N_y$ is one), there are no cross points. Thus, the coefficient matrix $A$ is reduced to a simpler form:

$$A = \begin{pmatrix} A_I & A_{IB} \\ A_{BI} & A_B \end{pmatrix}.$$

The D0 preconditioning matrix has the form

$$\tilde{A} = \begin{pmatrix} A_I & A_{IB} & 0 \\ 0 & \tilde{A}_B & \tilde{A}_{BC} \\ 0 & 0 & \tilde{A}_C \end{pmatrix}. \tag{3.9}$$

The block upper triangular matrix $\tilde{A}$ is an approximation of $A$, and both $A$ and $\tilde{A}$ contain the large block diagonal submatrix $A_I$.

In order for $\tilde{A}$ to be an efficient parallel preconditioner, we need to be able to compute

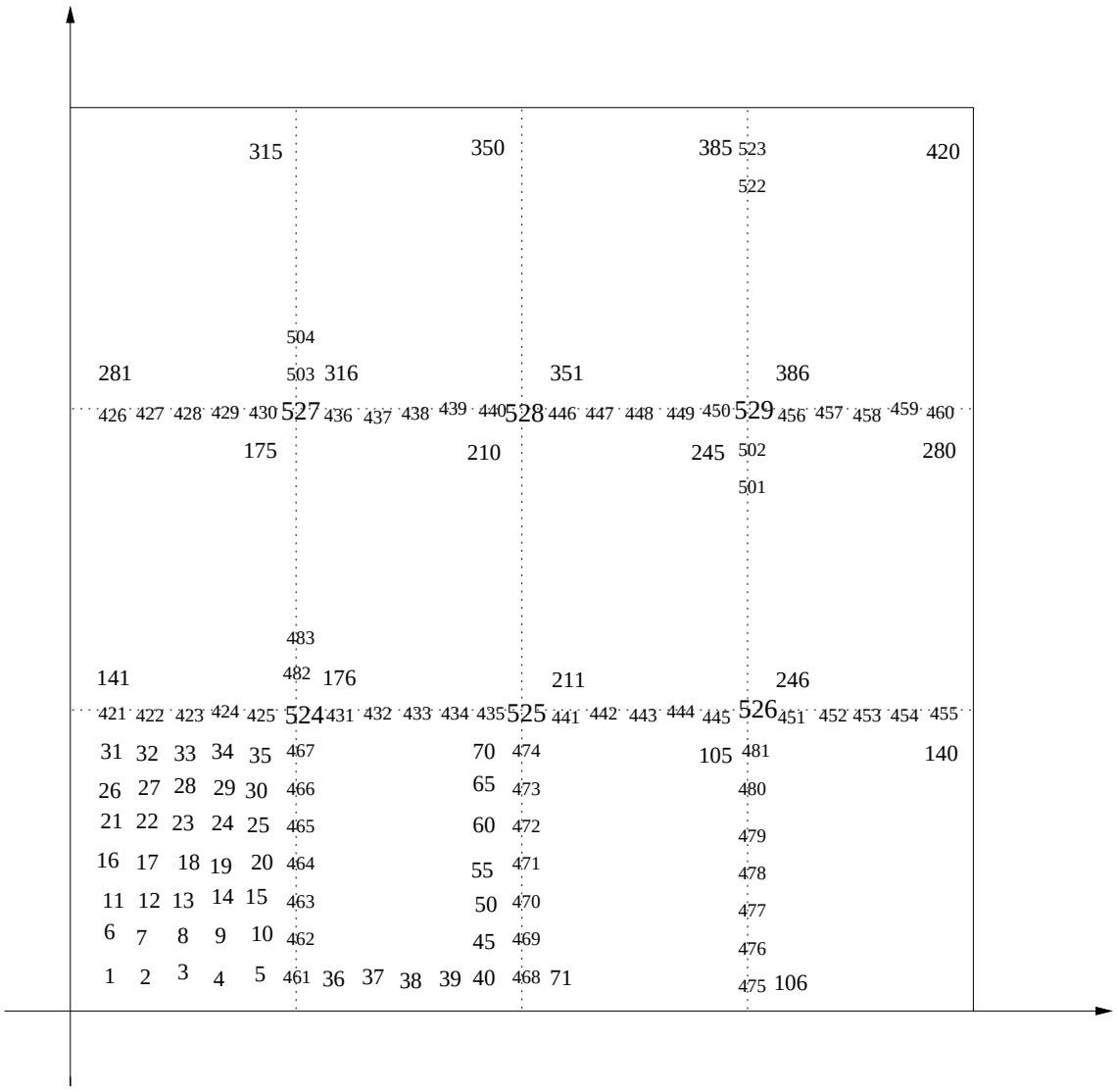



Figure 3.3: DD numbering for our preconditioners (D0 and D1).

$\tilde{A}^{-1}v$ efficiently in parallel. To do this, we solve $\tilde{A}u = v$, i.e.,

$$\begin{pmatrix} A_I & A_{IB} & 0 \\ 0 & \tilde{A}_B & \tilde{A}_{BC} \\ 0 & 0 & \tilde{A}_C \end{pmatrix} \begin{pmatrix} u_I \\ u_B \\ u_C \end{pmatrix} = \begin{pmatrix} v_I \\ v_B \\ v_C \end{pmatrix}. \tag{3.10}$$

First we solve the small cross point system

$$\tilde{A}_C u_C = v_C. \tag{3.11}$$

Then we update the right hand side and solve the interface system

$$\tilde{A}_B u_B = v_B - \tilde{A}_{BC} u_C. \tag{3.12}$$

Finally, we update the right hand side and solve the interior system

$$A_I u_I = v_I - A_{IB} u_B. \tag{3.13}$$

As we shall see, the second and third steps above are the most expensive computationally, and also the easiest to do in parallel. It should be mentioned also that the whole preconditioning matrix is distributed in our parallel implementation, with each processor holding the portion of the matrix corresponding to its unknowns.

The basic idea for the D0 preconditioner is to use a coarse grid approximation of the PDE to define the matrix $\tilde{A}_C$ in (3.10), and a "tangential" approximation of the PDE to define $\tilde{A}_B$ and $\tilde{A}_{BC}$. In constructing $\tilde{A}_B$ and $\tilde{A}_{BC}$, D0 counts on only coupling along the edge of subdomains (along the interface points in the tangential direction) and not the components associated with the normal direction to the edge for the **interface points**. This can be described through equations. Suppose that we are to discretize the following equation

$$au_{xx} + cu_{yy} + du_x + eu_y + f = g, \tag{3.14}$$

at an interface point on a horizontal interface line. At this point, the tangential direction is the $x$-direction along the interface line, and its normal direction is the $y$-direction. Now

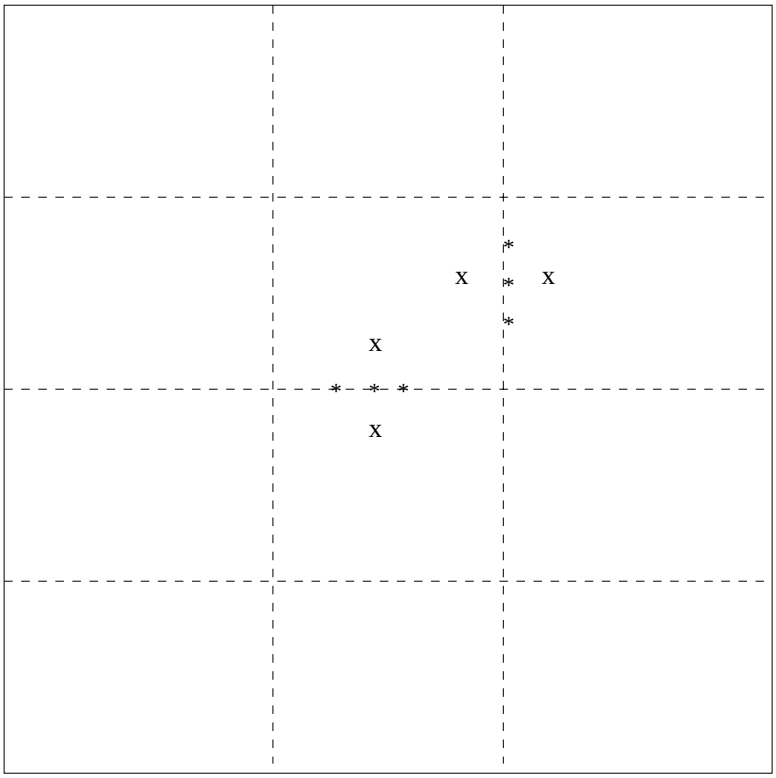



Figure 3.4: Coupling for interface points, tangential preconditioner (D0).

we keep the couplings along the tangential direction, and ignore the components associated with the normal direction. That is, instead of discretizing the original equation (3.14), we discretize the following equation

$$au_{xx} + du_x + f = g. \tag{3.15}$$

Note that the derivatives in the direction normal to the interface are simply ignored. As Figure 3.4 shows, the coupling components associated with the normal direction to the interface line are set to zero (ignored). The same thing is true for an interface point on a vertical interface line by exchanging the $x$-direction with the $y$-direction.

The idea of constructing an interface preconditioner using a tangential approximation

of the PDE is widely used. More references about tangential preconditioners can be found in [Keyes90], [Smith96], [Chan94].

Next, we study the structure of the preconditioning matrix in order to have a better understanding of its distribution across the processors.

### 3.4.1 The structure of the preconditioning matrix

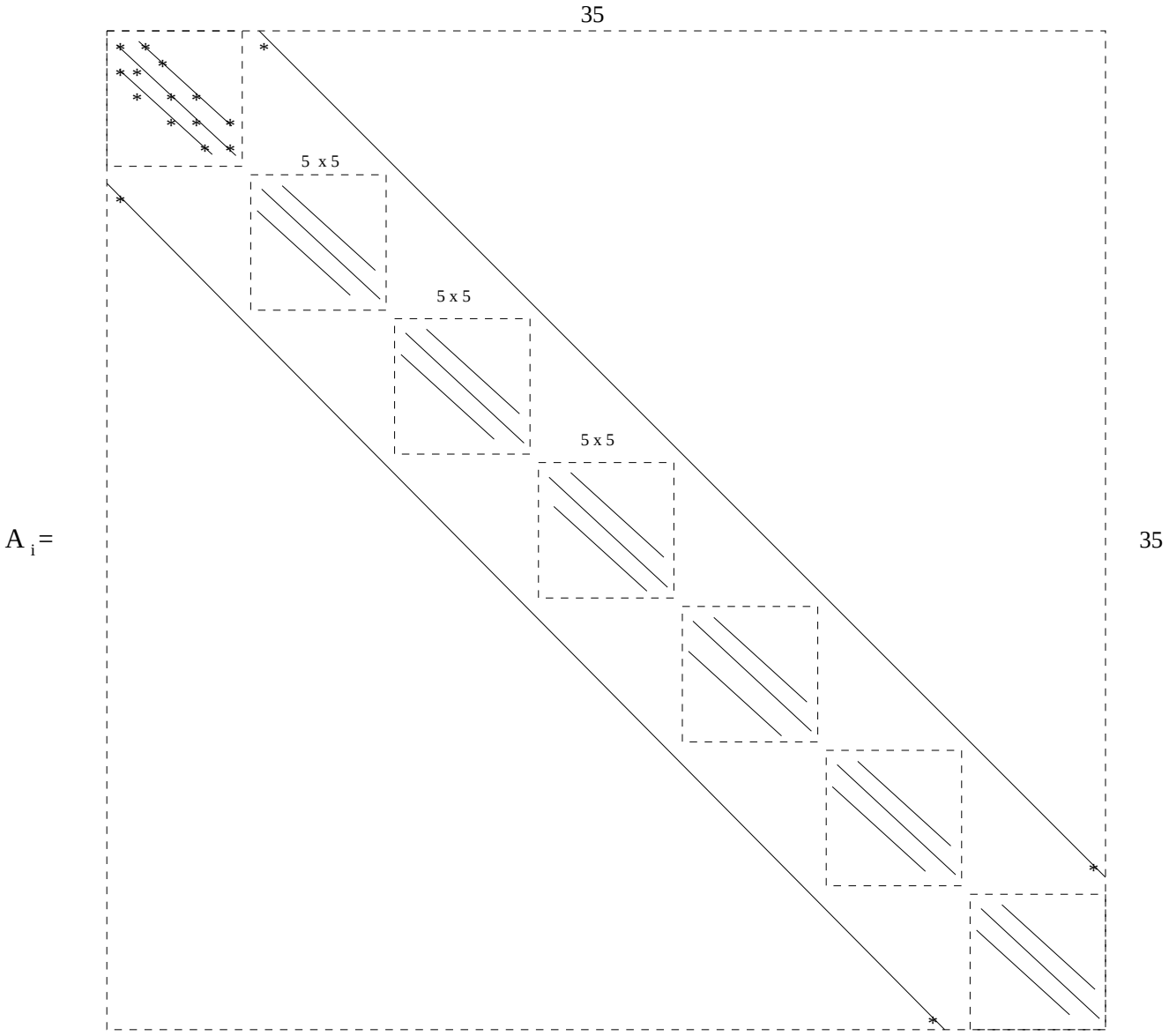
The preconditioning matrix $\tilde{A}$, an approximation of the matrix $A$, has the form

$$\tilde{A} = \begin{pmatrix} A_I & A_{IB} & 0 \\ 0 & \tilde{A}_B & \tilde{A}_{BC} \\ 0 & 0 & \tilde{A}_C \end{pmatrix}. \tag{3.16}$$

The large submatrix $A_I$ corresponding to interior points has a block diagonal structure,

$$A_I = \begin{pmatrix} A_1 & 0 & 0 & 0 \\ 0 & A_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & A_{12} \end{pmatrix}. \tag{3.17}$$

The submatrices $A_1, A_2, \ldots, A_{12}$ have identical structures, and they are block disjoint. This is very good for parallelization, since $A_I$ is the major part of the matrix $\tilde{A}$; for our example $A_I$ has dimension $420 \times 420$, compared to $\tilde{A}_B$ with dimension $103 \times 103$ and $\tilde{A}_C$ with dimension $6 \times 6$. Figure 3.5 displays the structure of each of the submatrices $A_i$ ($i = 1, 2, \ldots, 12$) of $A_I$.




Solid line indicates that the entries may be non-zero. All the other entries are zero.

Figure 3.5: Structure of the submatrix $A_i$.

The submatrix $\tilde{A}_B$ is also block diagonal. It has the form

$$\tilde{A}_B = \begin{pmatrix} A_{X1} & & & & & & & \\ & A_{X2} & & & & & & \\ & & A_{X3} & & & & & \\ & & & A_{X4} & & & & \\ & & & & A_{Y1} & & & \\ & & & & & A_{Y2} & & \\ & & & & & & A_{Y3} & \end{pmatrix}. \tag{3.18}$$

Submatrices $A_{Xi}$ $(i = 1, 2, 3, 4)$ of $\tilde{A}_B$ correspond to interface points on the horizontal interface lines (we call them $X$-interfaces) in the $i$-th column of subdomains, and submatrices $A_{Yj}$ $(j = 1, 2, 3)$ correspond to interface points on the vertical interface lines ($Y$-interfaces) in the $j$-th row of subdomains.

The structures of submatrices $A_{Xi}$ and $A_{Yj}$ are shown in Figure 3.6. All the $A_{Xi}$'s have the same structure; they are tri-diagonal, block disjointed. This is also true for all the $A_{Yj}$'s. Thus we can solve the linear systems for interface subdomains in parallel, although the degree of coarse grained parallelism is limited by the number of interfaces.

The submatrix for cross points has the form

$$\tilde{A}_C = \begin{pmatrix} x & x & & x & & \\ x & x & x & & x & \\ & x & x & & & x \\ x & & & x & x & \\ & x & & x & x & x \\ & & x & & x & x \end{pmatrix}. \tag{3.19}$$

The submatrices $A_{IB}$ and $\tilde{A}_{BC}$ are sparse matrices with only a few nonzeros in each row. They can be dealt with easily by tracing the neighbors and updating the right hand side.

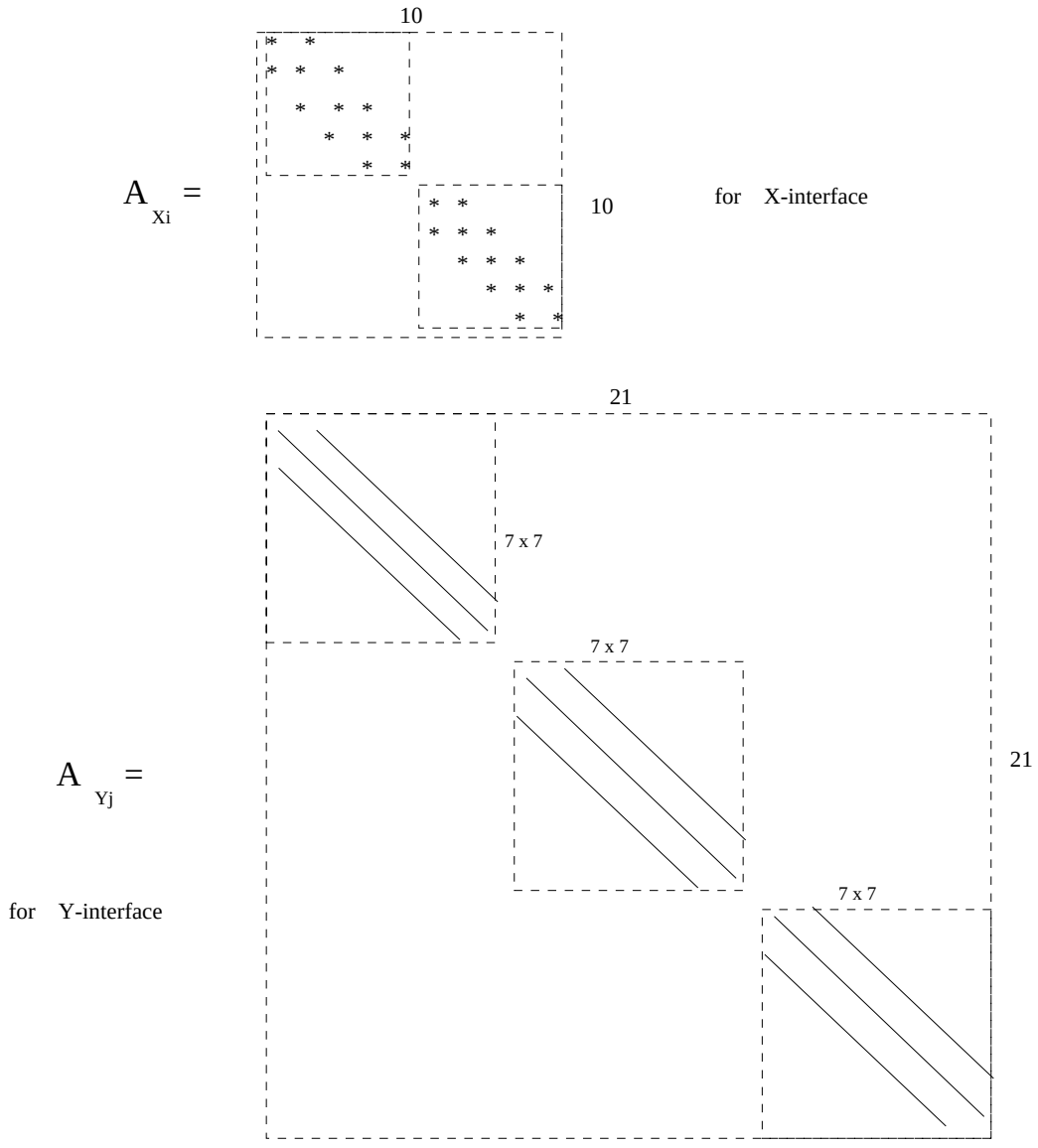



Figure 3.6: Structure of submatrices for $X$-interface and $Y$-interface points.

## 3.5 A three-level preconditioner (D1)

Recall that the equations for the interface points in preconditioner D0 are finite difference approximations to the PDE that simply ignore derivatives in the direction normal to the interface. This may truncate much information and introduce large errors. To reduce these errors, a three-level preconditioner was proposed by Pitts [Pitts95]. For convenience, we will call this preconditioner **D1**. Essentially D1 is a modification of D0. D1 has the same equations for both cross points and interior points as D0; only the equations for the interface points are different.

To describe the idea of D1, let us consider an example. Suppose that we are to discretize the PDE at an interface point $(x_0, y_0)$ on a horizontal interface line. Instead of ignoring derivatives in the direction normal to the interface, D1 uses the following finite difference approximations:

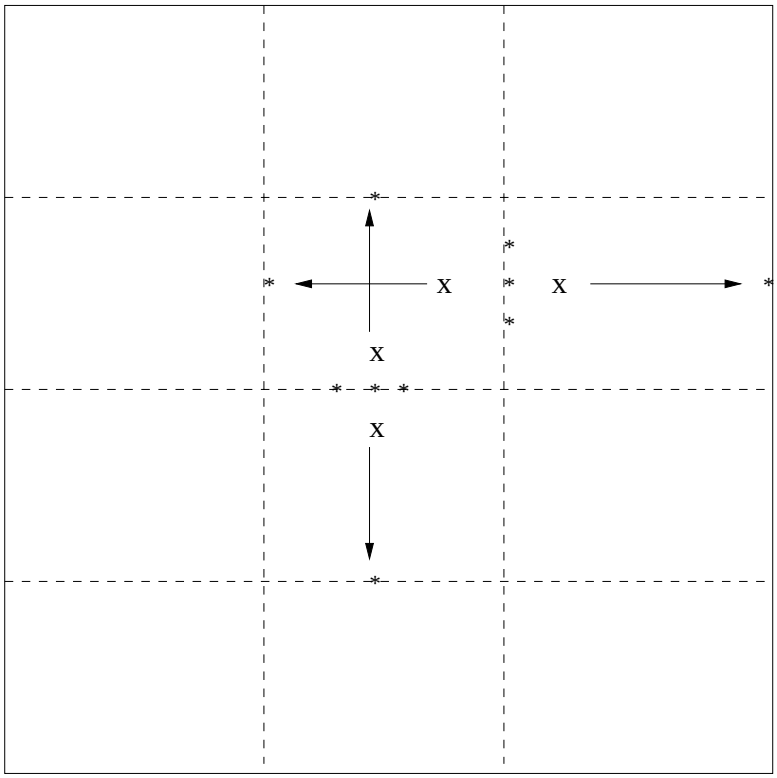
$$u_x(x_0,y_0) \approx \frac{u(x_0+h_x,y_0)-u(x_0-h_x,y_0)}{2h_x}, u_y(x_0,y_0) \approx \frac{u(x_0,y_0+H_y)-u(x_0,y_0-H_y)}{2H_y},$$

$$u_{xx}(x_0,y_0) \approx \frac{u(x_0+h_x,y_0)-2u(x_0,y_0)+u(x_0-h_x,y_0)}{h_x^2},$$

$$u_{yy}(x_0,y_0) \approx \frac{u(x_0,y_0+H_y)-2u(x_0,y_0)+u(x_0,y_0-H_y)}{H_y^2}.$$

As shown in Figure 3.7, the couplings normal to the edge, which were ignored in the tangential D0 preconditioner, are now replaced by coarse-grid interface points in D1. Notice that there are three types of grids involved. The interior points use the fine grid with fine mesh $h_x$ and $h_y$. The cross points use a coarse grid with coarse mesh $H_x$ and $H_y$. However, the interface points use a hybrid coarse/fine mesh with $h_x$ and $H_y$ combined. That is why we call this preconditioner a three-level preconditioner.

All the discussions for D0 can be applied to D1 except those for the interface points. Equations (3.8) — (3.13) are still valid for D1: The global coefficient matrix $A$ after the




x -- component ignored in D0, but replaced by course grid in D1.

Figure 3.7: Coupling for interface points in the preconditioner D1.

DD numbering has the form (3.8); the D1 preconditioning matrix $\tilde{A}$ has the form (3.9), i.e.,

$$\tilde{A} = \begin{pmatrix} A_I & A_{IB} & 0 \\ 0 & \tilde{A}_B & \tilde{A}_{BC} \\ 0 & 0 & \tilde{A}_C \end{pmatrix}; \tag{3.20}$$

we use Equations (3.10) — (3.13) to solve $\tilde{A}u = v$, in order for $\tilde{A}$ to be an efficient parallel preconditioner. For the structure of the preconditioning matrix $\tilde{A}$, the large submatrix $A_I$, block diagonal submatrix $\tilde{A}_B$, and the submatrix $\tilde{A}_C$ for cross points have the same structures as in (3.17), (3.18), and (3.19) respectively, except submatrices $A_{Xi}$ and $A_{Yj}$ in $\tilde{A}_B$. Submatrices $A_{Xi}$ $(i = 1, 2, 3, 4)$ correspond to interface points on the horizontal interface lines; $A_{Yj}$ $(j = 1, 2, 3)$ represent interface points on the vertical interface line. The structures of submatrices $A_{Xi}$ and $A_{Yj}$ are displayed in Figure 3.8. Notice that there is row coupling between the diagonal blocks of the $A_{Xi}$ and $A_{Yj}$ matrices.

In order to have a better understanding of our preconditioners, we outline the parallel preconditioner D1. There are mainly two aspects: one for setting up systems, and one for solving the systems (recall that one processor is responsible for one subdomain).

Set up systems. Each processor does the following:

1. Factor the matrix corresponding to my interior unknowns.
2. Form and factor vertical and horizontal interface matrices corresponding to my subdomain row and column.
3. Form and factor cross point matrix.

(Note: Steps 2 and 3 are "redundant". Step 1 is perfectly parallel.)

Solve the systems. Each processor does the following:

1. Gather right hand side $v_C$ of cross point system (each processor "owns" the crosspoint at its southwest corner, if any).

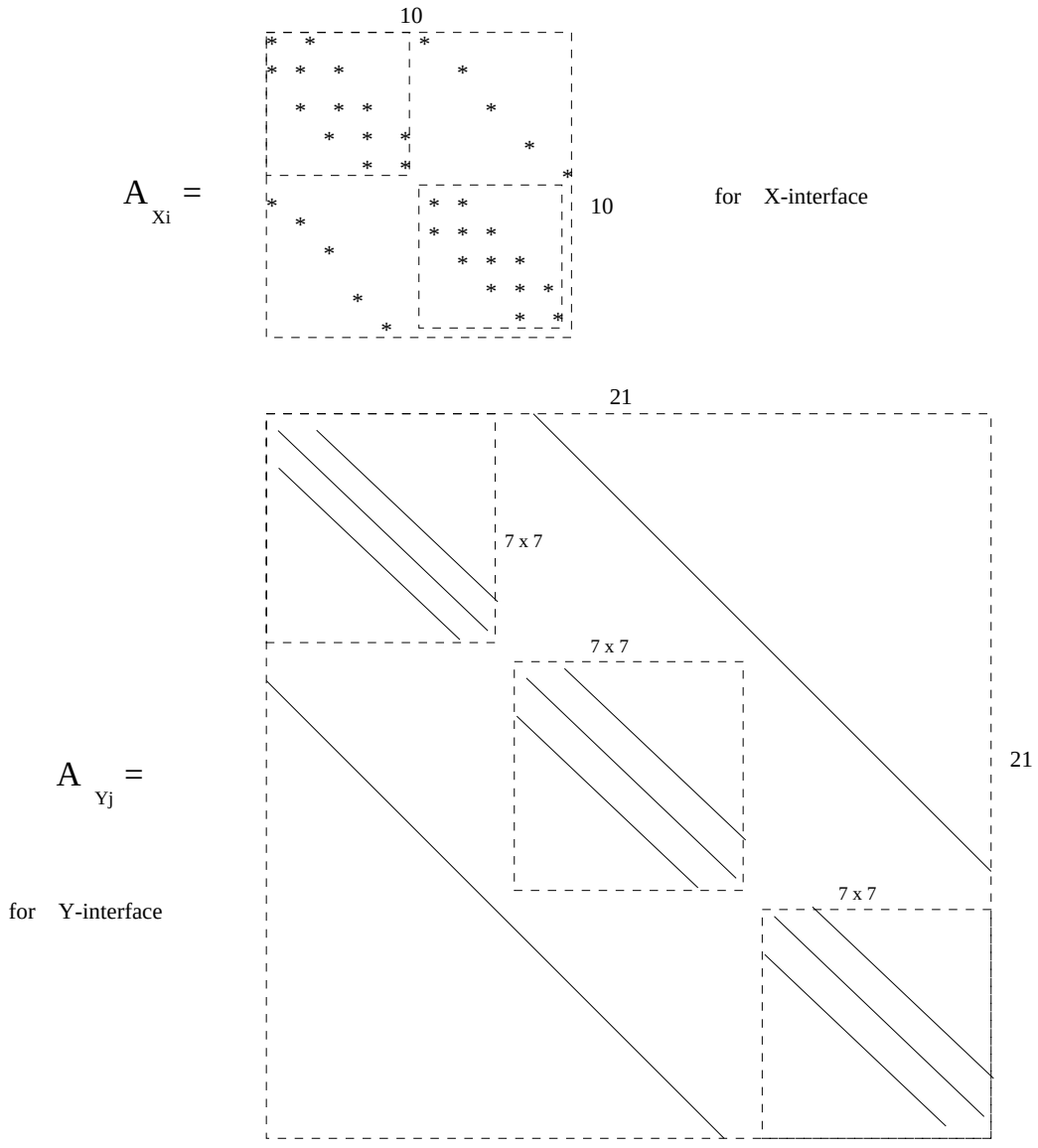



Figure 3.8: Structure of submatrices for $X$-interface and $Y$-interface points.

2. Solve $\tilde{A}_C u_C = v_C$ (redundant).

3. Update my portion of interface points (i.e., interface points on my south and west boundary), $v_B := v_B - \tilde{A}_{BC} u_C$.

4. Gather portion of $v_B$ corresponding to my row and column of subdomain. It is convenient to use MPI communicators.

5. Solve for all the interface unknowns (in $u_B$) corresponding to my row (column) of processors (i.e., $A_{Xi}$ and $A_{Yj}$ if I am processor $(i, j)$). Notice that this is a redundant part not in whole, but in row or column.

6. Update part of right hand side $v_I$ corresponding to my interior unknowns, $v_I := v_I - \tilde{A}_{IB} u_B$.

7. Solve for all the interior unknowns, the new $u_I$.

Since D1 is a modification of D0, the main difference in the implementation between D1 and D0 is dealing with interface points. D0 has much simpler matrix structures in the submatrices corresponding to interface points than D1, as we have seen in this section and the previous section. Step 2 in the setup phase, and Steps 4 — 5 in the solution phase are slightly different from the above in implementing D0.

# Chapter 4

# Numerical Experiments

## 4.1 Introduction

This chapter contains numerical experiment results that display how well our parallel preconditioner D1 performs, compared to the other two preconditioners BJ and D0. The scalability of the preconditioners and different views about performance are discussed. In addition, to answer the question of "how to achieve a better preconditioner?", the communication costs and amount of redundant computation for preconditioner D1 are explored.

### 4.1.1 The problem

The main problem we consider is Problem 12 from the ELLPACK PDE population [Rice85, Appendix A],

$$u_{xx} + u_{yy} + [1 + sin(\alpha x)]u_x - cos(\alpha y)u = f, \tag{4.1}$$

on the unit square, with Dirichlet boundary conditions. The function $f$ is chosen so that the true solution is

$$u(x, y) = cos\beta y + sin\beta(x - y). \tag{4.2}$$

For the parameters, $\alpha$ adjusts oscillation in the PDE coefficients, and $\beta$ adjusts oscillation in the solution. In this experiment, we set $\alpha = 10.0$ and $\beta = \pi$.

The following is the code for the problem in the parallel ELLPACK language:

```
*
*****************************
* PDE population: problem  12 *
*****************************
*
```

```
opt.  x subdomains = NDMSX
      y subdomains = NDMSY
      time

global.
      common / cprob / alpha, beta

eq.    uxx + uyy + (1.+sin(alpha*x))ux - cos(alpha*y)u = f(x,y)

bound.  u = true(x,y) on  x = 0.0
                      on  x = 1.0
                      on  y = 0.0
                      on  y = 1.0

grid. 129 x-points
      129 y-points

fort.
      alpha = 10.0
      beta  = pi

dis.  5 point star

*
* No preconditioning
*
sol.  pimgmres (eps=1.0e-6)
out.  max(error,20,20)
*
* Block Jacobi preconditioner on right
*
procedure. set block jacobi preconditioner
sol.  pimgmres (maxit=200, ipre=2, preconr = q7bjmn, eps = 1.0d-6)
out.  max(error,20,20)

*
* 7d0 (dd) preconditioner on right
*
procedure. set tangential preconditioner
sol.  pimgmres (maxit=200, ipre=2, preconr=q7d0mn, eps=1.0e-6)
out.  max(error,20,20)

*
* 7d1 (dd) preconditioner on right
*
procedure. set d1 preconditioner
sol.  pimgmres (maxit=200, ipre=2, preconr=q7d1mn, eps=1.0e-6)
out.  max(error,20,20)


sub.
      function true(x,y)
      common / cprob / alpha, beta
      true = cos(beta*y)+sin(beta*(x-y))
```

```
      return
      end
      function f(x,y)
      common / cprob / alpha, beta
      bxmy = beta*(x-y)
      ux = beta*cos(bxmy)
      b2 = (beta)*(beta)
      uxx = -b2*sin(bxmy)
      u = cos(beta*y)+sin(bxmy)
      uyy = -b2*u
      f = uxx + uyy + (1.+sin(alpha*x))*ux
     +    - cos(alpha*y)*u
      return
      end
end.
```

### 4.1.2 Experimental details

The global physical domain is decomposed into $N_x \times N_y$ subdomains. We assume that the number of processors involved in the computation equals the number of subdomains, i.e., $N_{procs} = N_x \times N_y$. We make this assumption based on the following reasons. First, it makes thing easy to implement. Second, it may enhance system utilization. Third, it balances the workload because every node is almost equally loaded and equally busy.

There are many variables which can be studied: number of processors, dimensions of subdomains $(N_x, N_y)$, discretization (grid) size $(n_x, n_y)$, stopping criteria, etc. One-dimensional domain decompositions (also called strips), such as $N_x = 1$, or $N_y = 1$, are special cases of two-dimensional domain decompositions.

We use the following as a guideline in our experiments:

- Group physical subdomains in rows or columns for interface subdomains.
- Use MPI as a tool to deal with message passing among processors, using communicators to handle the communication for interface subdomains.
- Integrate PIM into the ELLPACK computational framework.

We run our experiments on the Intel Paragon, with 64-bit arithmetic.

Table 4.1: Performance on Problem 12, with 16 processors.

| | | Preconditioners | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | NONE | | BJ | | D0 | | D1 | | D0/D1 |
| subdomain | 1/h | iter | total | iter | total | iter | total | iter | total | |
| 1 x 16 | 64 | 510 | 2.55 | 92 | 1.94 | 136 | 2.85 | 9 | 1.02 | 2.79 |
| 1 x 16 | 96 | 979 | 7.20 | 121 | 6.79 | 160 | 8.40 | 10 | 3.26 | 2.58 |
| 1 x 16 | 128 | 1556 | 16.25 | 144 | 16.00 | 177 | 17.94 | 11 | 7.21 | 2.49 |
| 1 x 16 | 192 | 3006 | 56.68 | 184 | 62.17 | 195 | 66.75 | 13 | 36.56 | 1.83 |
| 4 x 4 | 64 | 510 | 2.84 | 63 | 0.80 | 33 | 0.59 | 15 | 0.36 | 1.64 |
| 4 x 4 | 96 | 979 | 7.69 | 71 | 1.85 | 37 | 1.20 | 17 | 0.78 | 1.54 |
| 4 x 4 | 128 | 1556 | 18.49 | 84 | 4.00 | 39 | 2.22 | 18 | 1.45 | 1.53 |
| 4 x 4 | 192 | 3006 | 60.07 | 111 | 12.69 | 40 | 6.19 | 19 | 4.28 | 1.45 |

## 4.2 Performance of the preconditioners

In this section, we compare the results with the three preconditioners as well as the result without any preconditioner.

The experiment is based on the problem described in Section 4.1.1. We use GMRES method in PIM with the following parameters: ibasis=10 (this is the restart parameter), right preconditioning with BJ, D0 and D1. We test two cases, 1-D strips and 2-D squares. In both cases, we let the dimension of the matrix (the linear system) vary. Here we choose the convergence threshold $\epsilon$ fixed to be $10^{-6}$. The experimental results are shown in Table 4.1. The last column is the ratio of the total time of D0 over D1.

There are several observations that can be made on the results in Table 4.1:

- The importance of using a preconditioner is clear. In all cases the number of iterations is reduced substantially when a preconditioner is used. However, for the strip decomposition, only the D1 preconditioner actually reduces the total time. Interestingly, D0 does not even outperform BJ for the strip decomposition. For the 2-D decompositions, all three preconditioners produce a substantial reduction in time over the unpreconditioned case.

Table 4.2: Performance on Problem 2, with 16 processors.

| | | Preconditioners | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | NONE | | BJ | | D0 | | D1 | | D0/D1 |
| subdomain | 1/h | iter | total | iter | total | iter | total | iter | total | |
| 1 x 16 | 64 | 723 | 3.41 | 119 | 2.60 | 19 | 0.60 | 9 | 1.36 | 0.44 |
| 1 x 16 | 96 | 1540 | 10.83 | 158 | 8.44 | 18 | 1.70 | 10 | 3.65 | 0.47 |
| 1 x 16 | 128 | 2697 | 27.79 | 184 | 20.18 | 18 | 4.44 | 11 | 7.57 | 0.59 |
| 1 x 16 | 192 | 5421 | 101.36 | 242 | 77.69 | 17 | 20.89 | 13 | 46.40 | 0.45 |
| 4 x 4 | 64 | 723 | 3.98 | 67 | 0.86 | 30 | 0.50 | 16 | 0.38 | 1.32 |
| 4 x 4 | 96 | 1543 | 12.08 | 84 | 2.21 | 35 | 1.16 | 17 | 0.78 | 1.45 |
| 4 x 4 | 128 | 2696 | 31.58 | 100 | 4.73 | 38 | 2.32 | 18 | 1.42 | 1.63 |
| 4 x 4 | 192 | 5421 | 107.50 | 157 | 17.44 | 44 | 6.69 | 19 | 4.30 | 1.56 |

- D1 performs significantly better than either BJ or D0 in all cases. This is due to the fact that D1 requires many fewer iterations to achieve convergence. Note also that the number of iterations for D1 grows very slowly as $1/h$ grows.

- The advantage of D1 over D0 does shrink slightly as $1/h$ grows. This makes sense because the extra cost of the interface solves required with D1 grows with the fine grid dimension.

To analyze how the performance is affected by problem difficulty, we tested another problem, Problem 2 from the ELLPACK PDE population:

$$u_{xx} + (1+y^2)u_{yy} - u_x - (1+y^2)u_y = f \quad \text{on} \quad \Omega = [0,1] \times [0,1] \tag{4.3}$$

with Dirichlet boundary conditions, where $f$ is chosen such that the true solution is

$$u(x,y) = 0.135(e^{x+y} + (x^2 - x)^2 \log(1+y^2)). \tag{4.4}$$

This problem does not have the oscillation in the PDE coefficients or true solution that Problem 12 has. The results are shown in Table 4.2. From these two examples (see Tables 4.1 and 4.2), we may say the following:

- For strip decompositions, the harder problem sees a bigger advantage for D1 over D0. In fact, D0 outperforms D1 on Problem 2, with the number of iterations nearly constant with respect to $1/h$.

- For 2-D decompositions, there is a consistent advantage of about 50% for D1 over D0.

### Reducing bandwidth in interface systems

Before turning to parallel performance issues, we point out a simple, but important connection between the ordering of equations and unknowns and bandwidth reduction. If we choose the proper way of numbering unknowns, the bandwidth of the linear system can be reduced and solving time is thus reduced. For example, suppose we have an interface system with $n_y$ unknowns per processor in the $y$-direction and $N_x$ subdomains in each subdomain row. There are two natural methods to number the unknowns in the interface system:

**Method 1:** If we number the unknowns in the leftmost column first from the bottom to top, then the 2nd column from bottom to top, and so on, this results in a matrix with bandwidth $2(n_y + 1) + 1$.

**Method 2:** If we number the first row of the interface unknowns at the bottom first from the left to the right, then the second row from the left to the right, and so on, we end up with a matrix for the interface system with a bandwidth $2(N_x + 1) + 1$.

If $n_y$ is larger than $N_x$, we should choose method 2 to number the unknowns in the interface system. In that way, we can reduce the bandwidth of the matrix, and thus reduce the solving time. In general, method 2 is better than method 1 because $n_y$ is usually larger than $N_x$, and $n_x$ is larger than $N_y$. Table 4.3 shows the results before and after this renumbering for the D1 preconditioner. The computation time has improved after the renumbering. Notice that the change is more important for the strip (1D) decomposition because the interface system is bigger in this case.

Table 4.3: Before and after the renumbering for D1.

| | | Before renumbering (Method 1) | | After renumbering (Method 2) | |
|---|---|---|---|---|---|
| subdomain | 1/h | iteration | total | iteration | total |
| 1 x 16 | 64 | 9 | 2.05 | 9 | 1.02 |
| 1 x 16 | 96 | 10 | 5.24 | 10 | 3.26 |
| 1 x 16 | 128 | 11 | 11.41 | 11 | 7.21 |
| 1 x 16 | 192 | 13 | 44.03 | 13 | 36.56 |
| 4 x 4 | 64 | 15 | 0.38 | 15 | 0.36 |
| 4 x 4 | 96 | 17 | 0.81 | 17 | 0.78 |
| 4 x 4 | 128 | 18 | 1.56 | 18 | 1.45 |
| 4 x 4 | 192 | 19 | 4.47 | 19 | 4.28 |

It is important to notice that this idea can be used for the systems of interior points as well. If we choose a proper way to number the unknowns in the interior systems, we can reduce the bandwidth of the matrices and reduce the solving time for the interior systems. Again, the improvement would only be noticeable for subdomains with many more grid lines in one direction than the other (e.g., strip decomposition).

## 4.3 Parallel performance of the preconditioners

In this section, we report and compare parallel performance and scalability of the methods, i.e., the three preconditioners BJ, D0, and D1, using three measures of scalability described by Embree and Ribbens [Embree97]. We do not consider traditional fixed problem size speedup, since as the number of processors changes, the preconditioner changes (we assume that the number of processors equals the number of subdomains). Hence, as the number of processors changes, you are solving different preconditioned linear systems. Therefore, it is not that useful to compute traditional speedup, which compares performance on $p$ processors to performance on 1 processor, for a fixed problem. Furthermore, the scaled problem size metrics are more appropriate than the fixed size metrics when the number of processors varies over a wide range, e.g., 1 up to 64. All data reported in this

section is based on Problem 12.

### 4.3.1 Computational rate

According to Embree and Ribbens [Embree97] and Choi, et al. [Choi94], one common way to summarize and evaluate scalable parallel performance is based on MFLOP rates (millions of floating point operations per second). Suppose we have a 2-D decomposition with $N_x = N_y$. And suppose the "local" problem size is fixed, i.e., each processor is responsible for $n_{local} \times n_{local}$ unknowns. Obviously, as the number of processors is increased, the total MFLOP rate should increase. Furthermore, for a fixed number of processors, the total MFLOP rate will increase with local problem size because the per-processor computational work grows faster as a function of $n_{local}$ than inter-processor communication. However, there is some asymptotic limit to MFLOP rate for each fixed number of processors. Certainly, this asymptotic limit is less than the peak theoretical speed of the machine, for example. If we plot MFLOP rate vs. $n_{local}$ (see Figures 4.1 and 4.2), there are two questions to ask: "How quickly does each curve reach its asymptotic limit?" and "How close is that asymptotic limit to the peak achievable speed of the machine?" If an algorithm achieves a computational rate that is a reasonable fraction of the peak machine speed, and if it does so for a wide range of local problem sizes and numbers of processors, then it is considered scalable according to this view.

As shown in Figure 4.1, the MFLOP rate of D1 does increase as the number of processors increases as expected. (To compute MFLOP rates we used a modified version of a program written by Mark Embree [Embree97] which does a careful operation count for GMRES($m$).) The highest rates achieved for a given number of processors are typical for preconditioned iterative methods on elliptic problems (see [Embree97]). For example, with 36 processors the peak observed rate is approximately 300 MFLOPS, or approximately 17% of the peak theoretical speed of the machine (a single compute node is capable of 50 MFLOPS). However, we do see from Figure 4.1 that relatively large local problem sizes are required before the machine is efficiently used. For example, with 64 processors, even with a local problem

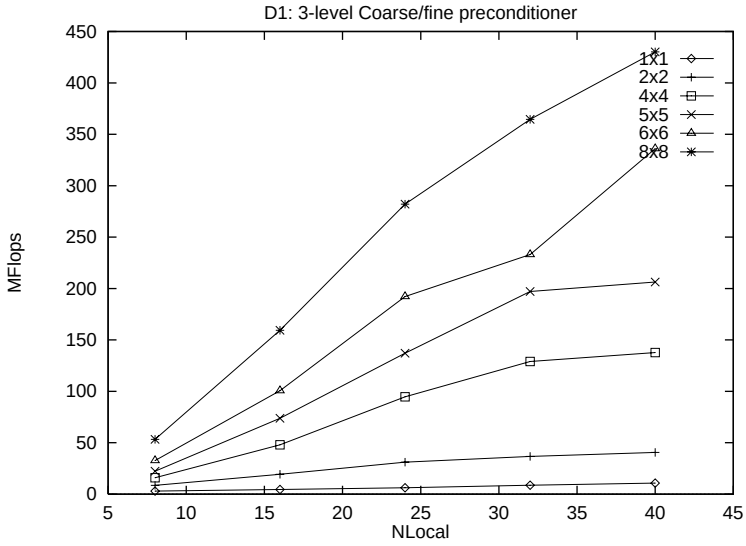



Figure 4.1: D1: Mflops vs. local problem size.

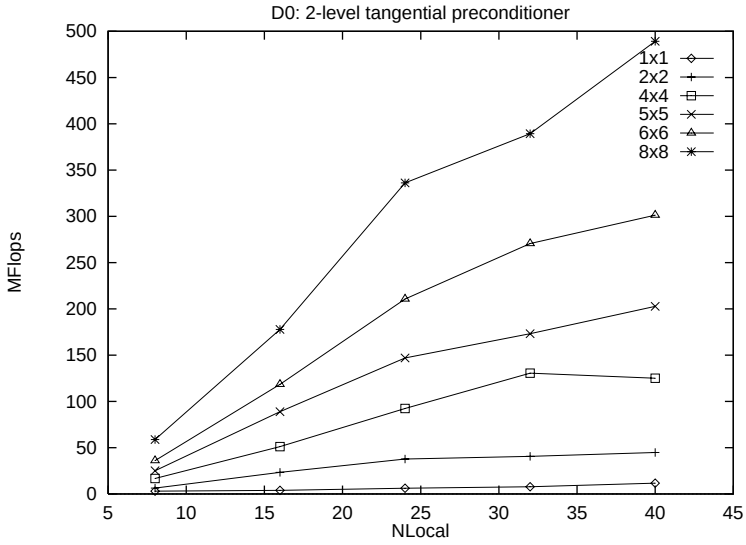



Figure 4.2: D0: Mflops vs. local problem size.

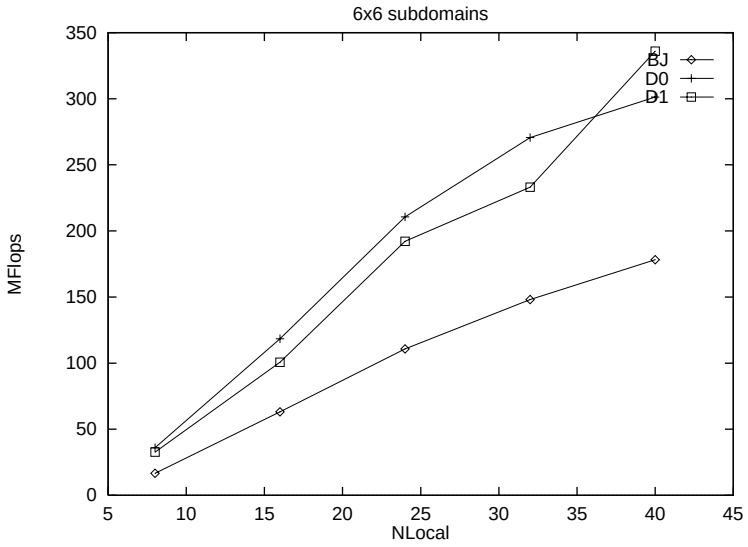



Figure 4.3: Computational rate.

size of $40 \times 40$ we may not have achieved the asymptotic limit to performance.

For comparison purpose, we also plot MFLOP rate vs. local problem size for the two-level (D0) preconditioner (see Figure 4.2). The results are similar to Figure 4.1.

Finally, in Figure 4.3 we plot the data for all three preconditioners for the $6 \times 6$ subdomain case. We see that all three methods perform poorly when the local problem size is small, but that D1 and D0 achieve a significantly higher computational rate for large problem sizes.

### 4.3.2 Isogranularity

The isogranularity view of scalability (see View 3 in [Embree97]) is to fix local problem size (fixed local "granularity") and to measure the computational rate as the number of processors increases. If the computational rate is a linear function of the number of processors (in other words, its plot is a straight line) the algorithm is considered scalable. The ideal case is that the plot of computational rate vs. the number of processors is a straight line with slope equal to the peak per-processor MFLOP rate. To measure scalability from this view point, we fix local problem size to be $40 \times 40$. Results for D1, D0, and BJ are shown

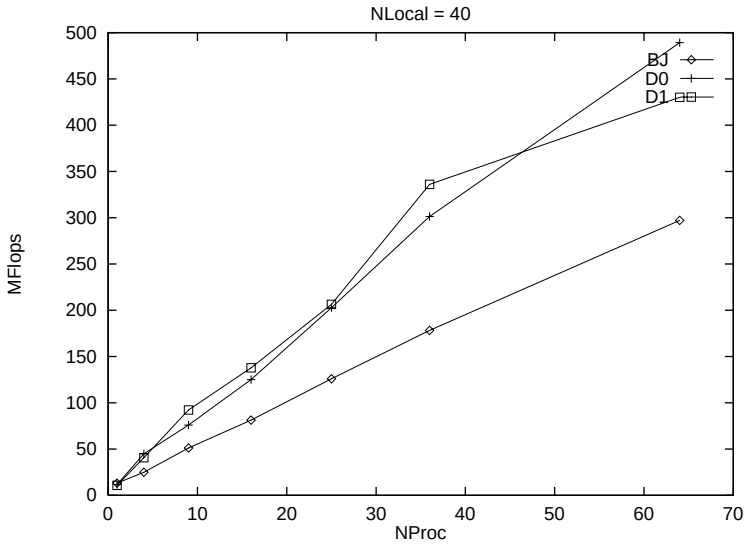



Figure 4.4: Isogranularity.

in Figure 4.4. Each curve in Figure 4.4 represents computational rate for the fixed local granularity. From Figure 4.4, it is easy to see that all three curves are roughly linear, with slopes considerably lower than the peak per-processor MFLOP rate (but consistent with the results in Section 4.3.1). There does not appear to be a significant difference between D0 and D1 based on the isogranularity view of scalability. However, computational rate is not the most important metric when comparing two different algorithms, as we discuss in the next section.

### 4.3.3 Time to solution

Computational rate and isogranularity are useful measures of scalability for a particular method, but it can be misleading to compare different iterative methods using only these measures. Obviously, a method could have a high computational rate but not be efficient because it takes too long to solve the problem, e.g., it takes too many iterations. It is almost always more important to measure the "bottom-line" time to solve the problem than to measure the computational rate alone. In this section, we focus our attention on the time to the solution as in [Embree97] (view 5 of scalability).

Table 4.4: Time to solution on Problem 12

| | Preconditioners | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | NONE | | BJ | | D0 | | D1 | | D0/D1 |
| subdomain | iter | total | iter | total | iter | total | iter | total | |
| 1 x 1 | 123 | 0.90 | 1 | 0.25 | 1 | 0.36 | 1 | 0.35 | 1.03 |
| 2 x 2 | 324 | 3.40 | 30 | 1.34 | 15 | 0.90 | 9 | 0.69 | 1.30 |
| 3 x 3 | 620 | 7.10 | 44 | 2.21 | 26 | 1.51 | 13 | 1.05 | 1.44 |
| 4 x 4 | 1004 | 11.97 | 60 | 2.98 | 26 | 1.70 | 13 | 1.26 | 1.35 |
| 5 x 5 | 2232 | 27.40 | 114 | 5.02 | 40 | 2.34 | 17 | 1.47 | 1.59 |
| 6 x 6 | 4672 | 58.65 | 204 | 8.53 | 56 | 3.09 | 24 | 2.03 | 1.52 |
| 7 x 7 | 6106 | 78.05 | 259 | 10.75 | 56 | 3.16 | 24 | 2.18 | 1.45 |
| 8 x 8 | 10690 | 137.04 | 401 | 16.39 | 75 | 4.30 | 31 | 2.72 | 1.58 |
| 9 x 9 | 13242 | 172.82 | 488 | 19.99 | 74 | 4.30 | 30 | 2.78 | 1.55 |

The problem considered is still Problem 12 of the PDE population with fixed granularity. Fixed granularity means the local problem size is fixed (and thus global problem size and solution accuracy grow with the number of processors). In other words, $H_x/h_x$ is fixed, where $H_x$ and $h_x$ are coarse-grid width and fine-grid width, respectively.

The best one could hope for in a scalable parallel PDE-solver is that the time and the number of iterations to solution would depend only on the local problem size. Thus the ideal case is that both the time and the number of iterations to solution should be independent of the number of processors as long as the granularity is fixed. We did our experiments on three different preconditioners: BJ, D0, and D1. The local size of the grid for each processor is fixed to be $32 \times 32$. The convergence threshold $\epsilon$ for GMRES varies as a function of global problem size for this experiment. We use $\epsilon = 10^{-5}$ for 1,2,3, and 4 subdomains in each direction, $\epsilon = 10^{-6}$ for $N_x = N_y = 5$, $\epsilon = 10^{-7}$ for $N_x = N_y = 6, 7$, and $\epsilon = 10^{-8}$ for $N_x = N_y = 8, 9$. Those value of $\epsilon$ are chosen so that the linear systems are solved with enough accuracy to achieve the expected level of accuracy in approximating the PDE.

The plot of time vs. the number of processors can be seen in Figure 4.5 with the corre-

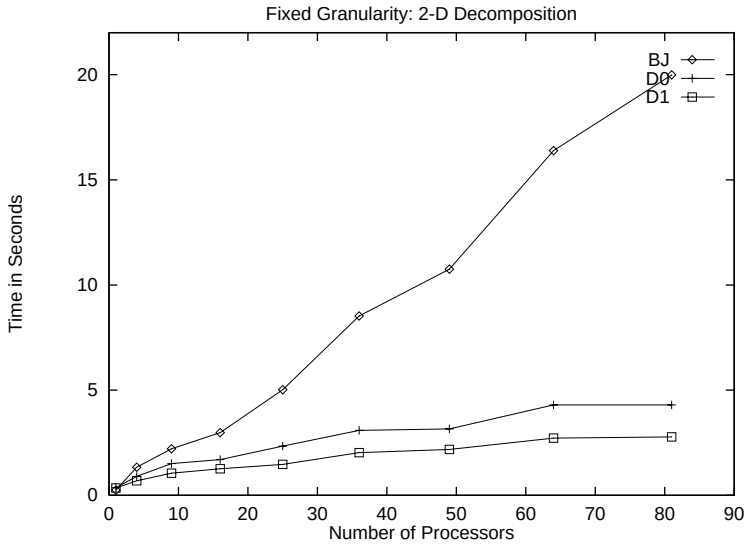



Figure 4.5: 2-D decomposition: time to solution (Problem 12).

sponding data listed in Table 4.4. By this measure of scalability, we conclude that

- D1 is the most scalable and efficient preconditioner. BJ is not competitive at all.
- D1's advantage is growing with the number of processors.
- The number of iterations grows more slowly for D1 than for D0 and BJ as the global problem size increases (see Table 4.4).
- Both the time to solution and the number of iterations for D1 grow very slowly as a function of the number of processors.

## 4.4 Towards better parallel performance

Although the 3-level D1 preconditioner performs well, in this section we consider the need and potential for improvement. To have good parallel performance, we need to consider several aspects about parallelization discussed in Chapter 3. In order to improve parallel performance with our preconditioner, we have the following questions to ask: "What is the cost for communications?", "Where is the bottleneck?" and "How can we make it faster?".

The following formula helps us to have a better understanding of parallelism and of how to improve the performance of our preconditioner.

$$parallel\ time \approx \frac{T_p}{p} + T_s + C,$$

where $T_p$ is the total time for doing parallel parts, $T_s$ is the total time for executing sequential sections, $C$ is for communication time, and $p$ is the number of processors.

To improve the parallel performance, we hope $T_p$ should be the major part, compared to $T_s$ and $C$, and hope to reduce $T_s$ and $C$ as much as possible. $T_s$ is due to "redundant" computation, when all the processors are executing the same thing sequentially.

### 4.4.1 Estimating the cost of communication

Recall that the main difference between D0 and D1 is the difference in dealing with the interface system. The purpose in this experiment is to measure the cost of the communication between the processors during the computation, i.e., to measure the communication time. To this end, we still choose Problem 12 with right-preconditioning. This time, we focus our attention only on 3-level preconditioner D1, with local granularity fixed to be $64 \times 64$, and 20 iterations. If we cut off communications for solving the interface system, communications for matrix-vector product, vector-norm, global sum, etc., we can see the difference between "the time with communication" and "the time without communication". The difference is the cost for communication or the penalty of communications. We measured the time in the three cases: with full communication (the normal case), no interface communication (cut off the interface communication only), and with no communication.

The data is shown in Table 4.5 (where total comm = total time - no commun, extra comm = total time - no interface commun). We see from Table 4.5 that the extra communication required in D1 (vs. D0) is relatively significant when compared with the total communication costs, varying from 20% to 75%. However, and more importantly, the total communication costs are not significant at all compared with the total time. Hence, the extra communication costs of D1 are not that significant.

Table 4.5: Measure of communication costs

| Preconditioner D1 | | | | |
|---|---|---|---|---|
| subdomain | total time | no interface commun | no commun | extra comm/total comm |
| 2 x 2 | 9.16 | 8.91 | 8.32 | 0.30 |
| 3 x 3 | 11.65 | 11.10 | 10.92 | 0.75 |
| 4 x 4 | 13.16 | 13.09 | 13.03 | 0.54 |
| 6 x 6 | 16.86 | 16.72 | 16.17 | 0.20 |
| 8 x 8 | 22.53 | 22.33 | 21.73 | 0.25 |

Table 4.6: Redundancy on interface factorization and solves

| subdomain | total time | redundancy | rate (percentage) |
|---|---|---|---|
| 2 x 2 | 5.47 | 0.30 | 5.5 |
| 3 x 3 | 14.81 | 1.09 | 7.4 |
| 4 x 4 | 29.00 | 2.65 | 9.1 |
| 5 x 5 | 37.26 | 5.07 | 13.6 |
| 6 x 6 | 71.54 | 9.67 | 13.5 |
| 7 x 7 | 92.90 | 13.66 | 14.7 |
| 8 x 8 | 147.30 | 21.57 | 14.6 |

### 4.4.2 Estimating the cost of redundant interface factorization and solve.

Another term that affects the parallel performance is the redundant work $T_s$ as we mentioned above. In order to estimate this factor, we measured the time (cost) of the redundant factorization and solving for the interface system. Table 4.6 gives us some idea how much redundant work there is. The last column in Table 4.6 is the redundant rate, defined as redundant time divided by the total time. We see that almost 15% of the computation time is spent doing redundant work as the number of subdomains grows.

### 4.4.3 A hybrid preconditioner

As we discussed in the previous subsections, the communication costs for the D1 preconditioner are not significant. However, the redundant work in the interface solvers takes

a significant portion of the total time that we cannot ignore. To reduce the redundant factorization and solving time and to reduce communication costs in solving the interface system, we propose a hybrid preconditioner of D0 and D1. This hybrid preconditioner, we call it D01, has a D0 "sweep", followed by a block-Jacobi iteration on the D1 interface system.

The advantage of the preconditioner D01 and the goals that we expect this preconditioner to achieve are as follows:

- It has D1-like convergence.
- It achieves D0-like parallelism.
- It reduces the solving time for the interface systems.

The idea of the D01 preconditioner is this: First all the processors in the same row (or in the same column) do the D0 "sweep" — solving the interface systems in D0's way. Then these processors do a "D1-like" iteration — using these solutions from D0 as initial guess to update the right hand side, reducing D1 interface system to block diagonal, which can easily be solved in parallel.

Let us recall the structures of submatrices for the interface systems of D0 and D1. After using the solutions from a D0-sweep as initial guess to update the right hand side, the matrices of $A_{X_i}$'s and $A_{Y_j}$'s for D1 are reduced from D1-like (see Figure 4.7) to D0-like block-diagonal matrices (see Figure 4.6). In this way, the solving time of the interface system is greatly reduced. More importantly, we can solve the interface systems with good parallelism as D0 does. Thus, this reduces the redundancy and it can achieve D0-like parallelism. The key question is how much the convergence will be hurt by this method, which amounts to a single iteration of block-Jacobi on the D1 interface system, with a good initial guess suplied by D0.

The following is the pseudo-code to implement the interface part for this hybrid preconditioner D01:

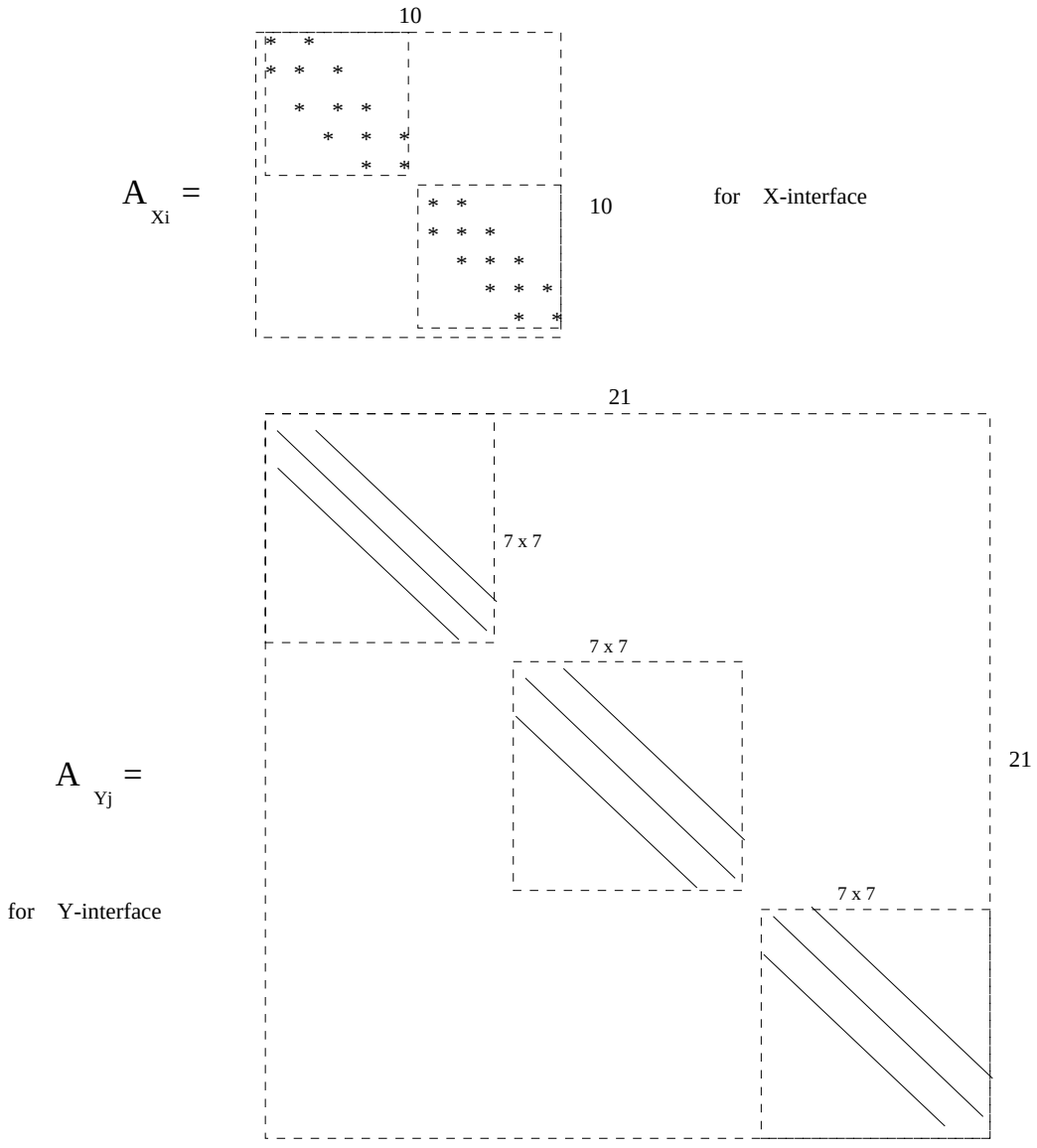



Figure 4.6: Matrix structure for D0's interface systems.

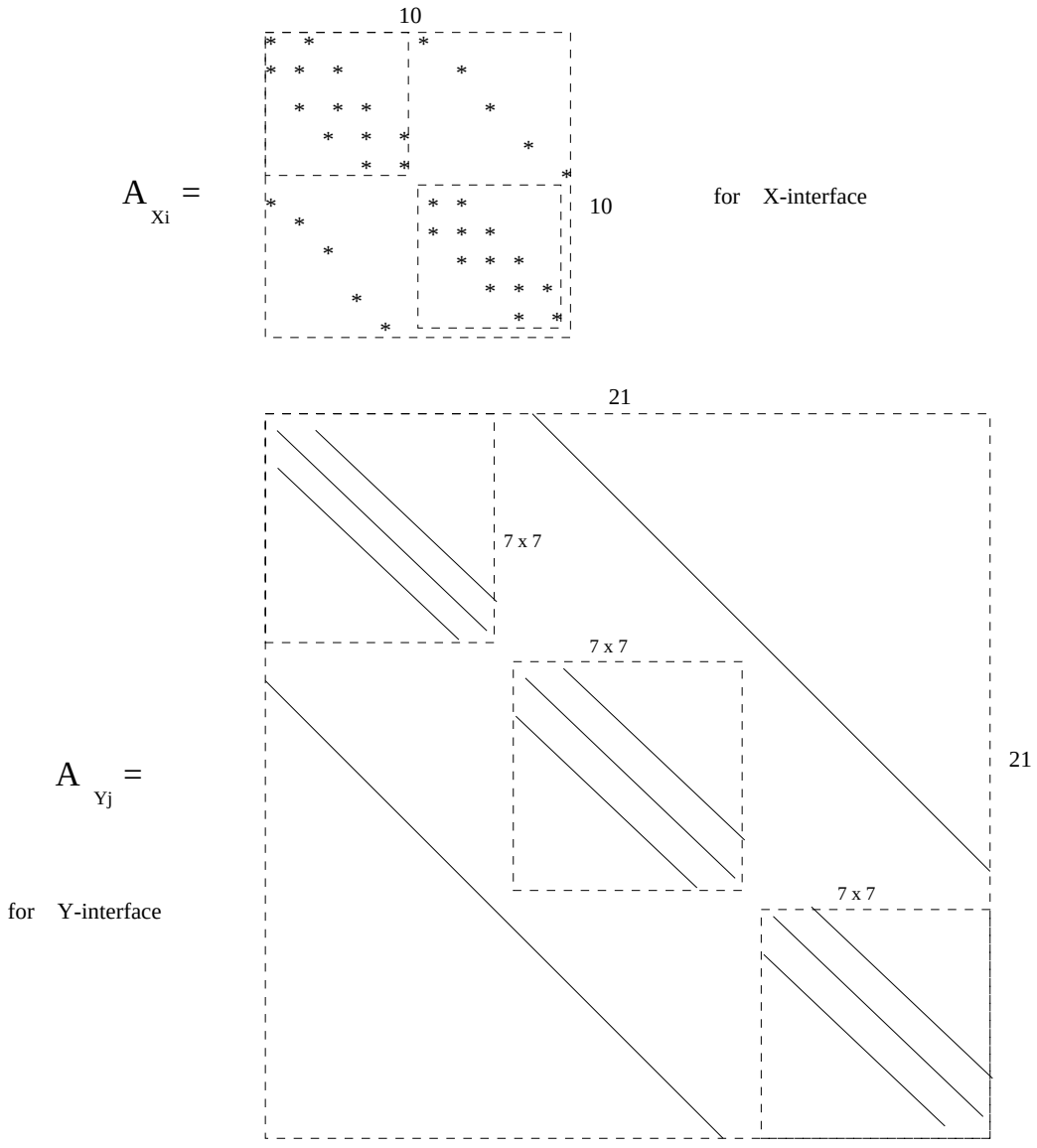



Figure 4.7: Matrix structure for D1's interface systems.

1. Form and factor the interface system once

```
Form and factor D0 interface system once (block diagonal).
Form and factor D1 interface system once
 (only have to factor diagonal blocks).
```

2. Solve the interface system iteratively

```
Solve D0 interface system in parallel.
Use the solution as initial guess to update the right hand side
  of D1 interface system .
Solve D1 interface system in parallel just as D0 does.
```

# Chapter 5

# Conclusions and Future Studies

## 5.1 Summary and conclusions

In brief words, we conclude the following: our three-level preconditioner D1 is the most efficient and scalable preconditioner on the test problem, compared to the other two preconditioners, BJ and D0. D1 substantially reduces both the number of iterations and computational time. The total communication cost of D1 is not significant. However, the redundant work of D1 on interface systems takes a significant portion of the total time when the interface system is big. We have proposed an idea to improve our preconditioner D1 to achieve a better performance.

To have a good parallelization and to implement a DD algorithm more efficiently, there are some practical matters about domain decomposition to consider:

- Partition the physical domain properly. The assumption that the number of subdomains equals the number of processors is reasonable. Use proper coarse grid size.
- Balance data evenly. Distribute data or matrix among the processors. The ideal case is nearly even divided.
- Use preconditioners. To find exact solution is more expensive in most cases. A good way is to incorporate an inexact solver into the existing framework, that is, to use preconditioners.
- Do computations in parallel in each iterative step. Do the tasks in parallel as much as possible. PIM (the parallel iterative methods package) is a very useful and convenient tool to achieve parallelization.

- Solve the coarse-grid problem. There are three choices: 1. keep the data in place and solve it in parallel with data exchanges; 2. gather the data in one node, solve there and broadcast the result; 3. gather the data to all nodes and solve it on all of them in parallel (redundantly). We used the third choice, and the resulting overhead was not serious. However for many more processors (and subdomains), the first alternative might be more efficient.

- Reduce message passing times, overhead, as much as possible. Keep the data locally (in each node). If the problem is not big, a very significant portion of the total time will be spent on message passing among nodes. So reduce the number of times of message passing (overhead) as much as possible.

## 5.2 Future directions

This thesis describes a concrete example of distributing data among processors, doing computation in parallel, and it shows how to construct a preconditioner and solver for distributed memory parallel computers or networks. It also uncovers many areas for future studies. The following aspects need to be considered:

1. Implementation of D01 idea, possibly generalizing to several Jacobi iterations.

2. Different stencils. We may use $3 \times 3$ stencil or $4 \times 4$ stencil for discretization instead of five-star stencil. More generally, discretization stencils on both sides of the linear system, such as HODIE(Rice), or HODIEA(Pitts) can be used.

3. More general equations. For Higher-order PDEs or systems of equations, domain decomposition and our efficient preconditioner idea can be applied.

4. General boundary conditions. In the research we have assumed Dirichlet-boundary conditions. For non-Dirichlet boundary conditions, the implementation becomes more complicated, which provides more challenging work.

5. General (non-rectangle) domains. ELLPACK includes a finite difference discretization module for non-rectangular domains. But how to define the subdomains and the hybrid coarse/fine grid is not obvious in this case.

6. 3-D domain decomposition. Apply the same idea to three-dimensional domain decomposition discretization.

# REFERENCES


[BARRETT94] Richard Barrett, et al., *TEMPLATES for the solution of linear systems: building blocks for iterative methods*. SIAM, Philadelphia 1994.

[BIRKHOFF84] G. Birkhoff, and R. Lynch, *Numerical Solution of Elliptic Problems*, SIAM, Philadelphia, 1984.

[BJØRSTAD89] P. E. Bjørstad, and O. B. Widlund, To overlap or not to overlap: A note on a domain decomposition method for elliptic problems, *SIAM J. Sci. Stat. Comput.* 10(5), pp.1053–1061, 1989.

[CHAN92] T. F. Chan, and D. Goovaerts, On the relationship between overlapping and nonoverlapping domain decomposition methods, *SIAM J. Matrix Anal. Appl.* 13(2), pp.663, 1992.

[CHAN94] Tony F. Chan, and Tarek P. Mathew, Domain Decomposition Algorithms, *Acta Numerica* (1994), pp.61-143.

[CHOI94] J. Choi et al., The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines, *Tech. Rep. TM–12470, Oak Ridge National Laboratory*, Oak Ridge, TN, 1994.

[CUNHA] Rudnei Dias da Cunha, and Tim Hopkins, *PIM 2.0, The Parallel Iterative Methods package for Systems of Linear Equations, User's Guide. PIM 2.1, User's Guide*, Universidade Federal do Rio Grande do Sul Brasil, 1997.

[DONGARRA79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK User's Guide*, SIAM Philadelphia, 1979.

[DONGARRA91] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.

[EMBREE97] M. Embree, and C. J. Ribbens, *On the Scalability of Parallel Krylov Subspace Methods*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.

[FOSTER95] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[FOX95] Geoffrey C. Fox, Basic Issues and Current Status of Parallel Computing – 1995, *NPAC Report SCCS–736*, 1995.

[FREUND92] Roland Freund, Gene H. Golub, and Noel Nachtigal, *Survey on Krylov subspace methods: Iterative Solution of Linear Systems*, Acta Numerica, 1992, pp.57–100, Cambridge University Press, Cambridge, 1992.

[GALLIVAN90] K. A. Gallivan, and others, *Parallel Algorithms for Matrix Computations*, SIAM, Philadelphia, 1990.

[GEIST94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machina*, MIT Press, Cambridge, MA, 1994.

[GROPP89] W. D. Gropp and D. E. Keyes, Domain Decomposition on Parallel Computers, *Research Report YALEU/DCS/RR–723*, Aug. 1989.

[GROPP94] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, the MIT Press, Cambridge, MA 1994.

[KEYES90] D. E. Keyes, and W. D. Gropp, Domain decomposition techniques for the parallel solution of nonsymmetric systems of elliptic boundary value problems. *Applied Numerical Mathematics*, 6(1989/90), pp.281–301.

[KEYES92] David E. Keyes, Domain Decomposition: A Bridge between Nature and Parallel Computers, *ICASE technical report*, ASME Winter Annual Meeting, November 8–13, 1992.

[KEYES95] D. Keyes, Y. Saad, and D. Truhlar, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*, SIAM, Philadelphia, 1995.

[KOELBEL94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.

[KRON53] G. Kron, A set of principles to interconnect the solutions of physical systems, *J. Appl. Phys.* 24(8), pp.965.

[KUCK96] David J. Kuck, What Is Good Parallel Performance? And How Do We Get It? *IEEE Computational Science & Engineering*, Spring 1996.

[LYNCH78] R. E. Lynch, and J. R. Rice, High accuracy finite difference approximation to solutions of elliptic partial differential equations, *Proc. Natl. Acad. Sci. USA* Vol. 75, No. 6, pp.2541–2544, June 1978. Applied Mathematical Sciences.

[ORTEGA88] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1988.

[PITTS95] George G. Pitts, Domain Decomposition and High Order Discretization of Elliptic Partial Differential Equations, *Ph.D. Dissertation, Virginia Tech, 1995.*

[PRZEM63] J. S. Przemieniecki, Matrix structural analysis of substructures, *Amer. Inst. Aero. Astro. J.* 1(1), pp.138–147.

[RICE85] John R. Rice, and Ronald F. Boisvert, *Solving Elliptic Problem Using ELLPACK,* Springer-Verlag, New York, 1985.

[SAAD86] Y. Saad, and M. H. Shultz, GMRES: a generalized minimum residual algorithm, *SIAM J. Sci. Stat. Comput.*, 7(1986), pp.856–869.

[SAAD96] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, ITP (An International Thomson Publishing Company), 1996.

[SCHWARZ1870] H. A. Schwarz, *Gesammelte Mathematische Abhandlungen*, volume 2, pp.133-143. Springer Verlag, 1890. First published in Vierteljahrsschrift der Naturforschenden Gesellschaft in Zurich, volume 15, 1870, pp.272–286. (earliest known iterative DD technique).

[SMITH96] Barry F. Smith, Petter E. Bjørstad, and William D. Gropp, *Domain Decomposition: parallel multilevel methods for elliptic PDEs*, Cambridge University Press, 1996.

[SNIR95] M. Snir, et al., *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1995.

[TOOLE95] John C. Toole, High Performance, High Confidence, High Capability, *IEEE Computational Science & Engineering*, Fall 1995.

# VITA


**College Address**

Dept. of Computer Science
Virginia Tech.
Blacksburg, VA 24061
*ayao@csgrad.cs.vt.edu*

**Home Address**

919 Camelot Dr., #40
Salem, VA 24153
(540)345-4168


**NAME:** Aixiang (I Song) Yao

**EDUCATION:** **M.S. in Computer Science** Spring 1998.

Virginia Polytechnic Institute and State University (Virginia Tech.)

Blacksburg, VA

GPA : Overall - 3.75/4.0; major - 3.8/4.0

**Ph.D. in Mathematical Physics**, Spring 1995

Virginia Polytechnic Institute and State University (Virginia Tech.)

Blacksburg, VA

GPA : Overall - 3.76/4.0; major - 3.82/4.0

**B.S. in Mathematics** Spring 1982, Wuhan University, China

GPA : Overall - 3.7/4.0; major - 3.8/4.0

**PROJECTS:** Department of Computer Science, Virginia Tech

**Parallel Computing**

- developing applications for distributed and parallel systems using MPI (message passing interface) and PIM (parallel iterative methods)

**Networks and the World Wide Web**

-co-designed and implemented an interface on the Web, which could be

used to search an ORACLE database. Java, CGI techniques are used.

-co-authored **WWW:Beyond the Basics**

(http://ei.cs.vt.edu/∼wwwbtb/book/)

**CC++ Project**

-used object-oriented techniques to test distributed system performance

-coded the program using CC++

**Digital Image Processing**

-Designed and implemented C programs to process digital images

-Investigated edge detection and pattern recognition algorithms

-Proposed a new technique for edge detection

**Assembler Project**

-Built an assembler for S68 machine including dynamic symbol table, parser, lexical analysis, and code generation.

**Database Project**

-used SQL to design and implement a relational database for a department. It supports queries across student, enrollment, and courses

**SKILLS:** **Computer Languages**

Visual C++, C, Java, HTML, Pascal, FORTRAN, CGI, ORACLE, Motif/OSF, Shell Script, MPI, Lex, Yacc, TEX, LATEX, Funnelweb

**Operating Systems**

OSF/1 3.2, SunOS 4.1.3/5.4, Solaris 2.3, Ultrix 4.4, AIX, MS-DOS 6.22, Microsoft Windows 3.11, Windows 95/NT, Macintosh, Intel Paragon OSF/1

**Other**

Web Server Adminstration, Netscape, OpenWindow, Matlab, Mathematica, Excel, ParaGraph, SPV (System Performance Visualization)

**EMPLOYMENT EXPERIENCE:**

*REFERENCES*

**Software Engineer**, Spring 1997-Present.

Data Systems Consulting, Inc., Roanoke, VA

**Research Assistant**, 1995-1997.

Dept. of Computer Science, Virginia Tech, Blacksburg, VA

-Developing applications and software for distributed and parallel systems using C, FORTRAN, MPI(Message Passing Interface)

**Teaching Assistant & Research Assistant**, 1990-95

Dept. of Math, Virginia Tech

-Served as consultant for all undergraduate courses in mathematics

-Research on applied math, kinetic theory, and spectral analysis

**Instructor**, 1988-Spring 1990

China Institute of Atomic Energy, Beijing, China

-Taught undergraduate courses in mathematics

**PERSONAL INTERESTS:**

Tennis, Volleyball, Light Music, Comedies, Hiking.

**PUBLICATIONS:**


1. Aixiang Yao, *Kinetic Theory and Global Existence in $L^1$ for a Dense Square-Well Fluid* Ph.D. Dissertation, Virginia Polytechnic Institute and State University, May (1995).

2. W. Greenberg, P. Lei, R. Liu and A. Yao, *Solution of Kinetic Equations for Dense Gases*, Proceedings of the Nineteenth International Symposium on Rarefied Gas Dynamics, I. Harvey, ed., (1994).

3. Aixiang Yao and Mingzhu Yang, *On the Index of Eigenvalues of Transport Operators in General Inhomogeneous Media* , Science in China (Scientia Sinica) **35**, no.7, 870-876 (1992).

4. Shenghua Wang and Aixiang Yao, *Solution to Parametric Equations with Generalized Boundary Condition in Transport Theory* , Acta Math. Sci., English Ed., **12**(1992),

no.4, 435-442. (Mathematical Review 94 h : 82056, American Mathematical Society, 1994).

5. Aixiang Yao and Mingzhu Yang, *On the Index of Eigenvalues of Transport Operator for Inhomogeneous Spheres* , (in Chinese), Acta Math. Sci. **12**(1992), no.2, 20-21.

6. Mingzhu Yang and Aixiang Yao, *On the Spectrum of the Transport Operator for a Bounded Convex Body*, Acta Math. Sci., English Ed., **10** (1990), no.4, 402-411. (Mathematical Review 92h: 47102, American Mathematical Society, 1992).

7. Aixiang Yao, *Neumann Series Solution for the Integral Boltzmann Transport Equation*, Acta Math. Sci., English Ed., **6**(1986), no.3, 279-285. (Mathematical Review 89a: 82064, American Mathematical Society, 1989).

8. Aixiang Yao and Mingzhu Yang, *Neumann Series Solution for General Stationary Transport Equation*, (in Chinese), J. Engineering Mathematics **4**(1987), no.3, 27-32.