High Performance Algorithms for Structural Analysis of Grid Stiffened Panels

by

Shaohong Qu

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. Zafer Gürdal

Dr. Christopher Beattie

September, 1997

Blacksburg, Virginia

High Performance Algorithms for Structural Analysis of Grid Stiffened Panels

by

Shaohong Qu

Committee Chairman: Dr. Calvin J. Ribbens

Computer Science

(ABSTRACT)

In this research, we apply modern high performance computing techniques to solve an engineering problem, structural analysis of grid stiffened panels. An existing engineering code, SPANDO, is studied and modified to execute more efficiently on high performance workstations and parallel computers. Two new SPANDO packages, a modified sequential SPANDO and parallel SPANDO, are developed. In developing the new sequential SPANDO, we use two existing high performance numerical packages: LAPACK and ARPACK to solve our linear algebra problems. Also, a new block-oriented algorithm for computing the matrix-vector multiplication $w = A^{-1}Bx$ is developed. The experimental results show that the new sequential SPANDO can save over 70% of memory size, and is at least 10 times faster than the original SPANDO. In parallel SPANDO, ScaLAPACK and BLACS are used. There are many factors that may affect the performance of parallel SPANDO. The parallel performance and the affects of these factors are discussed in this thesis.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude and appreciation to my advisor Calvin J. Ribbens for his valuable guidance and support throughout this research. Without his encouragement and patience, I could never have completed this work. I also wish to thank him for his carefully reading the whole thesis and correcting every technical and writing mistake.

Thanks go to my committee members, Dr. Gürdal and Dr. Beattie for their helpful suggestions and time spent reviewing this thesis. Thanks to my college and friend Burak Uzman for his kindly help.

Finally, I wish to thank my lovely wife, Yingjie Li, for her support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Computational scientists today face an increasingly complex world of science, engineering, and computation. Our understanding and modeling of the natural world is deeper and more complex every year. The problems that engineers want to model and solve only grow more detailed and complicated, as well. And the computational resources available—though offering greater and greater power—seem to come with a steep price: more and more complexity. In the face of these three waves of complexity it is extremely unlikely that one person will have sufficient expertise and experience in all three areas: natural science foundations, engineering analysis and design, and high performance computing. The need for interdisciplinary teams is obvious.

However, it is not always possible to put together an ideal team of computational scientists *a priori*. For example, it is very common for an engineer, who is an expert in analysis and design and has sufficient background in natural sciences and in computing, to implement reasonable computer models and solve modest problems. Unfortunately, when the problem size is scaled up and the computation moved to a high performance machine, the results are often extremely disappointing. Perhaps the code runs only slightly faster than on the engineer's own workstation, despite claims that the "supercomputer" is 100 times faster. Or perhaps the larger problem size is not even possible, due to memory limitations.

Modern high performance computers and algorithms are characterized by levels of complexity that are not present, or are not extremely important, in the typical desktop environment. Issues such as memory hierarchy, data distribution, and parallelism must be carefully dealt with, or performance on modern machines can suffer by several orders of magnitude. Even the engineer's desktop workstation is likely to contain two or more levels

of memory hierarchy; but for small problem sizes, using the memory hierarchy efficiently is not all that important. As the problem size grows, however, algorithms that fail to take memory hierarchy into account will run much slower. Similarly, if large parallel machines are to be used—especially distributed memory machines—complex issues of data distribution and communication must be addressed before any kind of reasonable performance can be achieved.

It is unreasonable to require an engineer to become expert in all the complexities of high performance computing (just as it is unreasonable to require a computer scientist to become expert in all the complexities of engineering analysis and design). By using sophisticated compilers and mathematical software libraries, the engineer can get some help in exploiting the computational resources without becoming an expert in them. However, for realistic applications, especially in a research setting, there is almost always considerable code that needs to be written from scratch.

The purpose of this research is to study the process of taking a typical engineering analysis code, and modifying it so that it runs efficiently on high performance machines. The focus is on identifying and quantifying the key algorithmic issues for efficient implementation of this type of application in both high performance workstation environments and a distributed memory parallel environment.

The engineering application code used as a test case for this research is the Stiffened Panel Design and Optimization (SPANDO) package [13], a code designed to do structural analysis of grid stiffened panels. The specific research goals with respect to SPANDO are these:

- Solve current problem sizes at least ten times faster on a single high performance workstation.
- For current problem sizes, investigate how much performance improvement is possible by moving to a parallel computer.
- Investigate how big a problem can be solved in "reasonable" time on a single work-

station and on the parallel machine.

- Describe the relative importance of the algorithmic issues considered.

The remainder of the thesis is organized as follows. Sections 1.1 and 1.2 give the engineering background to the problem, and an overview of the SPANDO package, respectively. In Chapter 2 the improvements for the sequential version of the code are introduced. Chapter 3 discusses changes made to parallelize SPANDO. The parallel performance is summarized and analyzed in Chapter 4. A summary and some remarks about future work are given in Chapter 5.

1.1 Background

To design an aircraft, one of the most important considerations is to reduce the weight. A large percentage of an aircraft's weight is from the structural frame; therefore, it is important to choose the lightest—but still adequate—material and structural configurations to minimize the frame weight. Panels that are comprised of both a skin and grid of stiffeners with a prescribed uniform pattern are promising structural components because they have superior response characteristics under combined loadings, and also, by using different stiffening arrangements, the response of these panels can be tailored to meet various design requirements [7]. Figure 1.1 shows an example of stiffened panel.

One approach in the analysis of grid stiffened structures uses the Lagrange Multiplier Method (LMM). The LMM approach computes the global stiffness properties for the entire panel by combining the stiffness properties of the skin and stiffeners. The Rayleigh-Ritz energy method is used to create a system of equations defining the total energy of the grid stiffened panel based on the total potential energy of the individual components of the panel, which are expressed in terms of displacement fields associated with the skin and stiffeners individualy. LMM is then used to minimize the total energy of the panel subject to displacement and slope continuity constraints between the skin and grid of stiffeners.

The details of deriving the algorithms are given by Grall and Gürdal [7]. Ultimately,

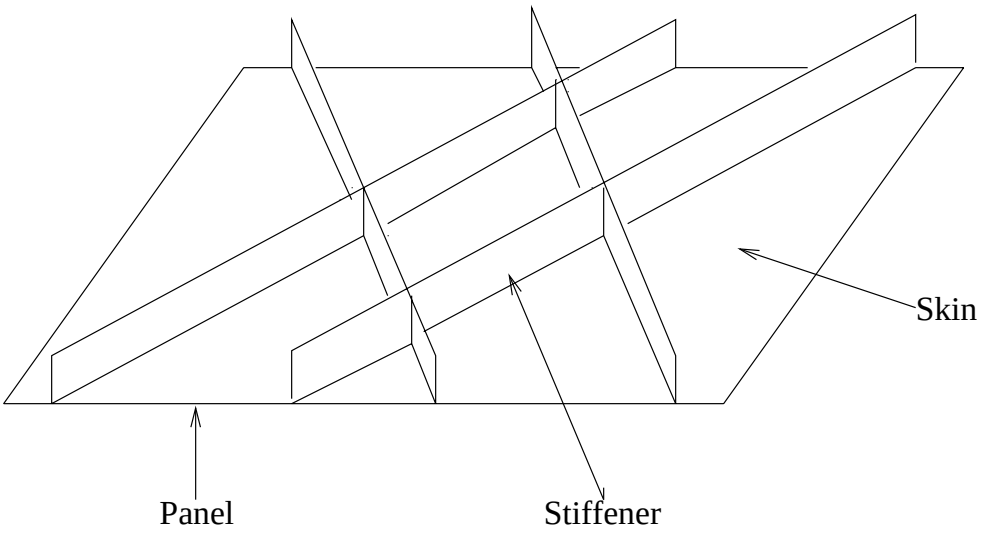

Figure 1.1: An example of stiffened panel.

the following generalized eigenvalue problem must be solved:

$$\begin{pmatrix} K_{11} & 0 & 0 & K_{14} & K_{15} \\ 0 & K_{22} & 0 & K_{24} & 0 \\ 0 & 0 & K_{33} & 0 & K_{35} \\ K_{41} & K_{42} & 0 & 0 & 0 \\ K_{51} & 0 & K_{53} & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ \mu \\ \nu \end{pmatrix} = \lambda \begin{pmatrix} M_{11} & 0 & 0 & 0 & 0 \\ 0 & M_{22} & 0 & 0 & 0 \\ 0 & 0 & M_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ \mu \\ \nu \end{pmatrix}, \tag{1.1}$$

where the unknowns are defined in Table 1.1 and the meanings of parameters (the dimension of the unknowns) are defined in Table 1.2. The application requires that we find the smallest real eigenvalue λ of system (1.1) and its associated eigenvector

The order of system (1.1) is given by

$$MN + IK + IK^{'}L + IJ + IJ.$$

By setting J=K (this sets the number of terms in the in-plane deflection series of the stiffener to be equal to the number of constraint points), it is possible to reduce the system

Table 1.1: The meaning of unknowns.

Unknown	Dimension	Meaning
A	M*N	Coefficients of skin out of plane deformation function.
B	I*K	Coefficients of stiffener out of plane deformation function.
C	I$K^{'}$L	Coefficients of shape function.
μ	IJ	Lagrange multipliers used for the skin-stiffeners interface continuity constraint.
ν	IJ	Lagrange multipliers used for the skin-stiffeners interface rotation constraint.

by eliminating μ and B. From (1.1) we have

$$\begin{aligned} \mu &= K_{24}^{-1}[\lambda M_{22}B - K_{22}B], \\ B &= -K_{42}^{-1}K_{41}A. \end{aligned}$$

The following reduced, or "condensed," system can then be derived:

$$\begin{pmatrix} K_{11}^{*} & 0 & K_{15} \\ 0 & K_{33} & K_{35} \\ K_{51} & K_{53} & 0 \end{pmatrix} \begin{pmatrix} A \\ C \\ \nu \end{pmatrix} = \lambda \begin{pmatrix} M_{11}^{*} & 0 & 0 \\ 0 & M_{33} & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ C \\ \nu \end{pmatrix}, \tag{1.2}$$

where

$$K_{11}^{*} = K_{11} + K_{14}K_{24}^{-1}K_{22}K_{42}^{-1}K_{41}, \tag{1.3}$$

$$M_{11}^{*} = M_{11} + K_{14}K_{24}^{-1}M_{22}K_{42}^{-1}K_{41}. \tag{1.4}$$

The order of the new system (1.2) is given by

$$MN + IK + IK^{'}L.$$

Equation (1.2) is the one actually solved by SPANDO.

Table 1.2: The meaning of indices.

Parameter	Meaning
M, N	The number of terms used in the skin deflection series (M in the x, N in the y direction).
I	The number of stiffeners.(NSTIF)
J	The number of terms in the in-plane deflection series of the stiffener.
K	The number of constraint points per stiffener.
$K^{'}L$	The number of terms used in the series defining the out-of-plane deflection series of the stiffeners.

1.2 SPANDO Package

In 1995, researchers in the Department of Engineering Science and Mechanics at Virginia Tech developed the Stiffened Panel Design and Optimization (SPANDO) package [13]. SPANDO employs LMM to predict the critical buckling loads and modal shapes of grid stiffened panels. It may also be coupled with Automated Design Synthesis (ADS), a numerical optimizer used to determine minimum weight or minimum cost designs subject to constraints on buckling of the panel assembly and material strength failure. Specifically, SPANDO calculates

1. The load distribution in the skin and stiffeners for a grid stiffened panel;
2. The critical buckling, the load resultants in the skin and stiffeners at the critical buckling load; and
3. The weight and cost of the entire panel.

There are nine main steps in a typical SPANDO analysis:

1. Input all the data necessary to describe the panel configuration.

2. Determine the stiffness properties of the skin and stiffeners. A smeared stiffness approach is used to calculate the global properties for the stiffened panel and forces in the skin and stiffeners.

3. Specified the location of constraint points.

4. The Rayleigh-Ritz energy solution is computed where the total energy of the grid stiffened panel is minimized subject to two continuity constraints that are enforced by LMM. The first constraint ensures continuity between the skin out-of-plane deformation and the stiffener in-plane deformation. The second constraint ensures that the skin and stiffeners remain at right angles to one another at all times.

5. The material strength for both the skin and stiffeners is computed.

6. A local failure analysis on the stiffeners is conducted.

7. The cost and weight of the panel are calculated.

8. Optimization is performed. This involves the minimization of an objective function (based on either weight or cost), using the ADS optimizer.

9. All output data are write to output files.

A detailed flow chart of the panel analysis and optimization procedure for SPANDO is shown in Figure 1.2 (this figure appears in [13]).

There are 15 subroutines in the SPANDO package. All of them are written in FORTRAN. Table 1.3 lists the routine names and their sizes. In Chapter 2, details of memory requirements and computation time for different problem sizes are discussed.

The original SPANDO package must be linked with the IMSL library. Three IMSL subroutines—DLINRG, DG3LRG, and DG8LRG—are used to compute matrix inverses and solve the generalized eigenvalue problem.

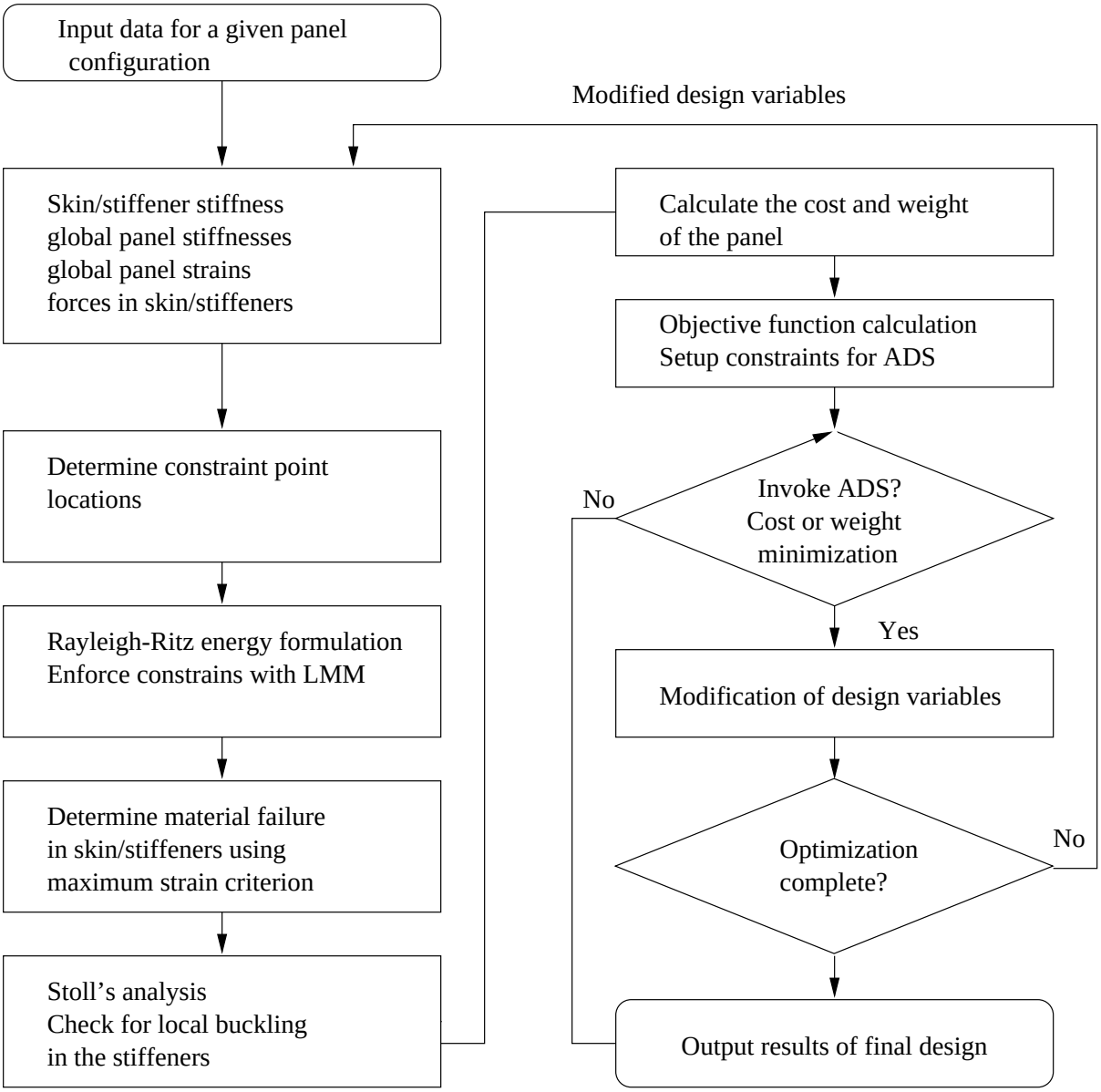

Figure 1.2: Flow chart of the panel analysis and optimization procedure used in SPANDO.

Table 1.3: SPANDO subroutine source code files.

Subprogram	Lines of code
ANLYZ	313
BREAK	313
COND	47
MMULT	67
OBJFUN	155
SIMPLE	212
STIF	704
STIF0	592
STIF1	184
STIF2	262
STIF3	349
STIF4	1258
STIF5	293
STIF6	160
STIF7	191

Chapter 2

Improvement of Sequential SPANDO

The first goal of this research is to improve the existing sequential SPANDO package so that modest problem sizes can be solved faster, and so that larger problem sizes can be solved. In this chapter, we analyze the original SPANDO code and then report on modifications to achieve a higher performance on a sequential computer. All results reported in this chapter are run on a Sun Sparc 20 workstation.

2.1 Analysis of Original SPANDO

2.1.1 Timing Information

To analyze SPANDO, first, the execution time used by each subroutine is measured. The input parameters for a standard test case are shown in Table 2.1.

Table 2.1: Parameters of the test file.

M	N	NSTIF	K	$K^{'}$
20	10	20	15	10

The size of the eigenvalue system is $MN + \text{NSTIF}(K + K^{'})$=700. Table 2.2 gives the time used by each subroutine. This time includes defining the matrices and one analysis (eigenvalue solve), but no optimization.

Table 2.2 shows that almost all the execution time is used by STIF4. In fact, the work in STIF4 is dominated by three computations:

1. Forming all sub-matrices $K_{11},\ K_{22},\ \ldots,\ K_{53},\ M_{11},\ M_{22},\ M_{33}$.

Table 2.2: Time (seconds) used by each subroutine.

STIF0	STIF1	STIF2	STIF3	STIF4	$\cdots$	Total
0.03	0.04	0.01	0.01	704.25	$\cdots$	705.73

2. Forming K_{11}^* and M_{11}^*, which are given in Equation (1.3), by computing two matrix inversions, K_{22}^{-1} and K_{44}^{-1}, six matrix-matrix multiplications, and two matrix additions:

$$\begin{aligned} \text{prod1} &= K_{14} \times K_{24}^{-1} \\ \text{prod2} &= K_{42}^{-1} \times K_{41} \\ \text{prod3} &= K_{22} \times \text{prod2} \\ \text{prod4} &= M_{22} \times \text{prod2} \\ \text{prod5} &= \text{prod1} \times \text{prod3} \\ \text{prod6} &= \text{prod1} \times \text{prod4} \\ K_{11}^* &= K_{11} + \text{prod5} \\ M_{11}^* &= M_{11} + \text{prod6} \end{aligned} \tag{2.1}$$

3. Solving the generalized eigenvalue system (1.2) by calling IMSL library function DG3LRG or DG8LRG. (DG3LRG only computes eigenvalues, DG8LRG computes both eigenvalues and eigenvectors.)

The time used for these three steps is 10, 249, and 445 seconds, respectively. Obviously, to reduce the execution time, it is necessary to find more efficient methods to do the matrix condensing procedure and to solve the generalized eigenvalue system.

2.1.2 Memory Information

The memory requirements of the original SPANDO package are dominated be the 22 arrays listed in Table 2.4. The sizes in the third column of Table 2.4 reflect the maximum parameter values allowed by the original code (see Table 2.3). The actual array dimensions

Table 2.3: Maximum values for original SPANDO parameters.

Parameter	Maximum	Meaning
K	25	Number of constraint points per stiffener.
$K^{'}$	20	Number of terms used in the series defining the out–of–plane deflection of the stiffeners.
M,N	30,30	MN is the number of terms used in the skin deflection series.
NSTIF	25	Number of stiffeners.

are

$$\text{MN=M} \times \text{N} = 900,$$

$$\text{NK=NSTIF} \times \text{K} = 625,$$

$$\text{NKP=NSTIF} \times K^{'} = 500,$$

$$\text{LDMAT=MN+NK+NKP=2025}$$

It requires 234 Mb of memory to run SPANDO. This is too large for a common computer to use this package. Furthermore, we would like to increase the values in Table 2.3. In the following sections we will describe steps taken to reduce these memory requirements.

2.2 BLAS Matrix-Matrix Multiply

As mentioned before, there are six matrix-matrix multiplies needed to calculate K_{11}^{*} and M_{11}^{*}. The subroutine MMULT in SPANDO does matrix-matrix multiplication in a straight forward way. However, much better performance can be achieved if the BLAS subroutine DGEMM is used.

The Basic Linear Algebra Subprograms (BLAS) [4, 5, 8] are high quality "building block" routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. Using matrix-matrix operations (in particular, matrix multiplication) tuned for

Table 2.4: Memory size for each array.

Name	**Dimension**	**Memory size (Mb)**
AMAT	LDMAT×LDMAT	32.8
AMAT14	MND×NKD	4.5
AMAT15	MND×NKD	4.5
AMAT22	NKD×NKD	3.1
AMAT24	NKD×NKD	3.1
AMAT33	NKPD×NKPD	2.0
AMAT35	NKPD×NKD	2.5
AMAT41	NKD×MND	4.5
AMAT42	NKD×NKD	3.1
AMAT51	NKD×MND	4.5
AMAT53	NKD×NKPD	2.5
BMAT	LDMAT×LDMAT	32.8
BMAT22	NKD×NKD	3.1
BMAT33	NKPD×NKPD	2.0
PROD1	MND×NKD	4.5
PROD2	NKD×MND	4.5
PROD3	NKD×MND	4.5
PROD4	NKD×MND	4.5
PROD5	MND×MND	6.5
PROD6	MND×MND	6.5
ECOPY	LDMAT×LDMAT	32.8
EVEC	LDMAT×LDMAT	65.6
Total		234

Table 2.5: Execution time (seconds) of Matrix-Matrix multiplication.

Problem size	MMULT	DGEMM	MMULT/DGEMM
450	64	4	16
920	371	31	12

a particular architecture can effectively mask the effects of the memory hierarchy (i.e., cache misses, virtual memory page misses, etc.), and permit matrix operations to be performed near the top floating point speed of the machine.

Table 2.5 compares the execution time of these two subroutines for four matrix-matrix multiplications, where the problem size in Table 2.5 is the size of matrix AMAT. We see that DGEMM is more than ten times faster than MMULT.

2.3 LAPACK Instead of IMSL for Linear Solves

The Linear Algebra PACKage (LAPACK) [1] is a collection of routines for linear system solving, least squares, and eigenproblems. LAPACK can attain high performance by using algorithms that do most of their work in calls to the BLAS. LAPACK seeks to make efficient use of the hierarchical memory by avoiding having to reload the cache too frequently. It casts linear algebra computations in terms of block-oriented, matrix-matrix operations through the use of the Level 3 BLAS. This approach results in maximizing the ratio of floating point operations to cache misses.

For the following four reasons we choose LAPACK as the linear solver instead of IMSL.

1. LAPACK can achieve better performance.
2. ARPACK (described in Section 2.4) depends on LAPACK.
3. LAPACK is freely available for a wide range of platforms.
4. ScaLAPACK, a parallel version of LAPACK, is also available.

Table 2.6: Influence of block size on matrix inversion.

Block size	8	16	32	64	128	256
time (sec)	0.186	0.182	0.180	0.178	0.178	0.182

Because LAPACK uses block algorithms—algorithms that operate on blocks or submatrices of the original matrix—the block size may affect the performance of the code. We test a simple case (matrix inversion) and find that the effect of block size is not significant for this application. Table 2.6 shows the execution times of computing the inverse of a 260×260 matrix using different block sizes.

In the matrix condensing procedure (computing K_{11}^* and M_{11}^*), two matrix inverses K_{42}^{-1} and K_{24}^{-1} are calculated. The time used for these two matrix inversions is given in Table 2.7, where the matrix size in the table is the size of matrix K_{42}. (K_{42} and K_{24} have the same size.)

Table 2.7: Time (seconds) of matrix inversions.

Matrix size	IMSL	LAPACK	IMSL/LAPACK
150	0.7	0.2	3.5
400	1.4	0.4	3.5

The data in Table 2.7 shows that to compute matrix inversion LAPACK is 3.5 times faster than IMSL.

2.4 ARPACK Instead of IMSL for Eigensolve

SPANDO only needs to find the maximum negative and minimum positive eigenvalues of the generalized eigenvalue system. The Original SPANDO uses IMSL subroutines DG3LRG to compute eigenvalues of the generalized eigenvalue system and DG8LRG to compute both eigenvalues and eigenvectors of the generalized eigenvalue system. Both

subroutines calculate all eigenvalues of the system. In subroutine STIF5, the maximum negative and minimum positive eigenvalues are then found from among all the eigenvalues. This procedure wastes computation time and memory—a large working space (ECOPY) is needed to solve the system and a large complex matrix (EVEC) is needed to store all the eigenvectors. In this section we discuss another approach—computing only a few necessary eigenvalues by using ARPACK.

2.4.1 ARPACK

ARPACK [11] is a software package developed at Rice University. It is designed to solve large scale eigenvalue problems. ARPACK is based on the Implicitly Restarted Arnoldi Method (IRAM). Details of IRAM can be found in [9]. The important features of ARPACK are as follows.

- A reverse communication interface allows the user to choose a matrix-vector product subroutine that takes advantage of matrix structure.
- ARPACK can be used to return only a few eigenvalues with a user specified criterion such as largest real part, largest absolute value, etc.
- A fixed pre-determined storage requirement suffices throughout the computation. Usually this is $nO(k) + O(k^2)$, where k is the number of eigenvalues to be computed and n is the order of the matrix.
- The numerical accuracy of the computed eigenvalues and vectors is user specified.

There are many sample driver routines in ARPACK that can be used as templates. A user only needs to modify the appropriate driver routine and write a problem specific matrix-vector multiplication subroutine. The following gives an example driver (DNDRV1) from ARPACK that shows how to use ARPACK to find a few eigenvalues and corresponding eigenvectors for the standard nonsymmetric eigenvalue problem $Ax = \lambda x$, where A is an $n \times n$ real nonsymmetric matrix.

```
c
c     %---------------------------------------------%
c     | M A I N   L O O P (Reverse communication loop) |
c     %---------------------------------------------%
c
 10   continue
c
c        %-------------------------------------------%
c        | Repeatedly call the routine DSAUPD and take |
c        | actions indicated by parameter IDO until    |
c        | either convergence is indicated or maxitr   |
c        | has been exceeded.                          |
c        %-------------------------------------------%
c
         call dnaupd ( ido, bmat, n, which, nev, tol, resid,
     &                 ncv, v, ldv, iparam, ipntr, workd, workl,
     &                 lworkl, info )
c
         if (ido .eq. -1 .or. ido .eq. 1) then
c
c           %--------------------------------------%
c           | Perform matrix vector multiplication |
c           |              y <--- OP*x             |
c           | The user should supply his/her own   |
c           | matrix vector multiplication routine |
c           | here that takes workd(ipntr(1)) as   |
c           | the input, and return the result to  |
c           | workd(ipntr(2)).                     |
c           %--------------------------------------%
c
            call av ( workd(ipntr(1)), workd(ipntr(2)))
c
c           %----------------------------------------%
c           | L O O P   B A C K to call DNAUPD again. |
c           %----------------------------------------%
c
            go to 10
c
         end if
c
c        %-----------------------------------------%
c        | No fatal errors occurred.               |
c        | Post-Process using DSEUPD.              |
c        |                                         |
c        | Computed eigenvalues may be extracted.  |
c        |                                         |
c        | Eigenvectors may be also computed now if  |
c        | desired.  (indicated by rvec = .true.)  |
c        |                                         |
c        | The routine DSEUPD now called to do this  |
c        | post processing (Other modes may require  |
c        | more complicated post processing than     |
c        | mode1.)                                 |
c        |                                         |
```

```
c        %-----------------------------------------%
c
         rvec = .true.
c
         call dneupd ( rvec, 'All', select, d, v, ldv, sigma,
     &         bmat, n, which, nev, tol, resid, ncv, v, ldv,
     &         iparam, ipntr, workd, workl, lworkl, ierr )
c
c        %-----------------------------------------------%
c        | Eigenvalues are returned in the first column |
c        | of the two dimensional array D and the       |
c        | corresponding eigenvectors are returned in   |
c        | the first NCONV (=IPARAM(5)) columns of the  |
c        | two dimensional array V if requested.        |
c        | Otherwise, an orthogonal basis for the       |
c        | invariant subspace corresponding to the      |
c        | eigenvalues in D is returned in V.           |
c        %-----------------------------------------------%
c
```

Example: The reverse communication interface and post processing for eigenvectors.

Table 2.8 describes some parameters used in ARPACK subroutines DNAUPD and DNEUPD.

Table 2.8: Parameters used in DNAUPD and DNEUPD.

Parameter	Description
ido	Reverse communication flag.
nev	The number of requested eigenvalues to compute.
ncv	The number of Lanczos basis vectors.
bmat	Indicates whether the problem is standard bmat='I' or generalized (bmat='G').
which	Specifies which eigenvalues of A are to be computed
tol	Specifies the relative accuracy to which eigenvalues are to be computed.
iparam	Specifies the computational mode, number of IRAM iterations, the implicit shift strategy, and outputs various informational parameters upon completion of IRAM.

2.4.2 Using ARPACK in SPANDO

SPANDO must compute the maximum negative and minimum positive eigenvalues of the generalized eigenvalue system

$$Ax = \lambda Bx.$$

ARPACK is appropriate to use in SPANDO for two major reasons: (1) ARPACK computes a few user specified eigenvalues of large scale problems; (2) ARPACK provides a reverse communication interface so that the user can take advantage of matrix structure.

Since there always exist maximum negative and minimum positive eigenvalues for a real engineering problem, instead of solving the generalized eigenvalue problem, it is possible to use ARPACK to solve an equivalent standard eigenvalue problem

$$\frac{1}{\lambda}x = A^{-1}Bx,$$

or

$$\mu x = A^{-1}Bx, \tag{2.2}$$

where $\mu = \frac{1}{\lambda}$. ARPACK is used to compute the smallest and largest real eigenvalues μ of Equation (2.2).

Our main eigenvalue-solving subroutine is a modification of the ARPACK sample driver DNDRV1, which computes the eigenvalues of largest magnitude (modulus) of a nonsymmetric standard eigenvalue problem. Given these eigenvalues, it is easy to find the real eigenvalues of smallest magnitude of the original generalized problem.

In each call to our matrix-vector multiplication subroutine, a vector $w = A^{-1}Bv$ is computed by the following two steps:

1. Compute matrix-vector product $u = Bv$.

2. Solve linear system $Aw = u$.

Each step requires $2n^2$ flops, where n is the dimension of matrices A and B, and assuming an LU factorization of A is computed initially. The LU factorization rerequires $\frac{2}{3}n^3$ flops.

So a total of

$$\frac{2}{3}n^3 + 4kn^2 \tag{2.3}$$

flops are needed for matrix-vector multiplication (k is the number of iterations, which is usually less than 150 for this application).

Table 2.9 compares the running time by using ARPACK and IMSL to solve the eigenvalue system for our three test problems. In these experiments the number of eigenvalues computed by ARPACK is 8. The numerical accuracy (residual tolerance) is set to approximately machine epsilon, 10^{-16}. Table 2.9 shows that using ARPACK reduces the running time by almost a factor of 10. Because ARPACK only computes a few eigenvalues, it does not need working space ECOPY and the complex array EVEC for storing all eigenvectors. Hence, 98.4 Mb memory space has been saved as well (see Table 2.4).

Table 2.9: Time (seconds) for solving eigenvalue problem.

Problem size	IMSL	ARPACK	IMSL/ARPACK
240	15.5	2.4	6.5
450	85.2	8.9	9.6
700	445.4	47.3	9.4

In ARPACK, the number of eigenvalues (m) to be computed is a parameter specified by the user. For different problems a different value of m can be used. When we use a small m ($m = 8$), the computation time is small. But because we compute eigenvalues that have largest magnitude, it is possible that all the m computed eigenvalues are complex. When this happens, a larger m is required to find the maximum and minimum real eigenvalues. In our tests, $m = 8$ is good enough for all test cases considered.

Table 2.10 gives the computation times when m is changed (the problem size is 450). When m is increased 4 times, the computation time only increases by 33%.

Table 2.10: Running time effected by number of eigenvalues.

Number of eigenvalues	computation time (sec)	%
4	7.43	100%
8	8.17	110%
12	8.95	121%
16	9.85	133%

2.5 Modified Condensing Procedure

In this section, the condensing procedure that reduces the size of the eigenvalue system from system (1.1) to system (1.2) is considered.

The original SPANDO package follows procedure (2.1) to compute A_{11}^* and B_{11}^*. This procedure requires two explicit matrix inversions and six matrix-matrix product operations. The two matrix inversion operations use

$$2 \times (\frac{2}{3}\mathrm{NK}^3 + 2\mathrm{NK}^3) = \frac{16}{3}\mathrm{NK}^3 \quad \text{flops.} \tag{2.4}$$

The six matrix-matrix product operations need

$$8\mathrm{MN} \times \mathrm{NK}^2 + 4\mathrm{MN}^2 \times \mathrm{NK} \quad \text{flops.} \tag{2.5}$$

Six large arrays, prod1, prod2, ..., prod6, are needed to store the intermediate results.

Instead of following the above procedure, the new procedure given below can be employed to reduce the computation time and save memory space.

1. Solve linear system $K_{42}K_{41}^{'} = K_{41}$, store $K_{41}^{'}$ in K_{41}.
2. Solve linear system $K_{24}K_{22}^{'} = K_{22}$, store $K_{22}^{'}$ in K_{22}.
3. Solve linear system $K_{24}M_{22}^{'} = M_{22}$, store $M_{22}^{'}$ in M_{22}.
4. Compute matrix-matrix multiplication prod2 $= K_{22}K_{41}$.

5. Compute matrix-matrix multiplication $K_{11}^* = K_{11} + K_{14} \times \text{prod2}$, store K_{11}^* in K_{11}

6. Compute matrix-matrix multiplication $\text{prod2} = M_{22}K_{41}$.

7. Compute matrix-matrix multiplication $M_{11}^* = M_{11} + K_{14} \times \text{prod2}$, store M_{11}^* in M_{11}

Matrices prod1, prod3, ..., prod6 are not needed in the new procedure; this saves 26.5 Mb memory (see Table 2.4). The new approach solves three linear systems and calculates four matrix-matrix multiplications. Solving three linear systems require

$$\frac{4}{3}\text{NK}^3 + 2\text{MN} \times \text{NK}^2 + 4\text{NK}^3 \text{ flops.}$$

The four matrix-matrix multiplications need

$$4\text{MN}^2 \times \text{NK} + 4\text{MN} \times \text{NK}^2 \text{ flops.}$$

Compared with the original procedure, the new procedure saves

$$2\text{MN} \times \text{NK}^2 \text{ flops.} \tag{2.6}$$

This savings in work is not nearly as significant as the savings in memory requirements due to the modified condensing procedure.

2.6 Block Algorithm

In this section the sparsity of the matrices is considered when solving the eigenvalue system

$$\mu x = A^{-1}Bx. \tag{2.7}$$

In this application, the arrays A and B (arrays AMAT and BMAT) are sparse; Figure 2.1 gives the structures of AMAT and BMAT. We can see from Figure 2.1 that AMAT has three large zero blocks and BMAT has seven zero blocks.

In Section 2.4.2 , we discussed using ARPACK to solve the eigenvalue problem. Recall that ARPACK is based on IRAM, which is an iterative method. ARPACK provides a

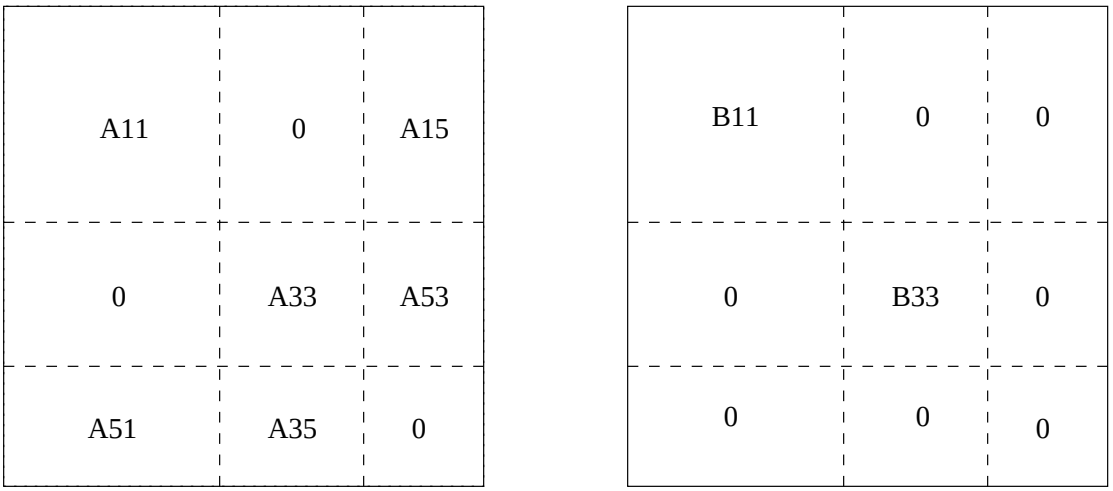

Figure 2.1: The structure of AMAT and BMAT.

reverse communication interface so that users can call their own matrix-vector product subroutine in each iteration. In our problem, the subroutine is used to compute

$$w = A^{-1}Bv, \tag{2.8}$$

where v is the input vector and w is the output vector.

To compute w, the following steps are taken:

- Factor $A = LU$ (once only, before the first call to ARPACK),
- compute $x = Bv$,
- solve lower triangular linear system $Ly = x$,
- solve upper triangular linear system $Uw = y$.

Since matrices A (AMAT) and B (BMAT) are structured (Figure 2.1), a more efficient computation strategy can be derived. First, rewrite

$$v = \begin{pmatrix} v_1(n_1) \\ v_2(n_2) \\ v_3(n_3) \end{pmatrix}, \qquad x = \begin{pmatrix} x_1(n_1) \\ x_2(n_2) \\ x_3(n_3) \end{pmatrix},$$

and

$$y = \begin{pmatrix} y_1(n_1) \\ y_2(n_2) \\ y_3(n_3) \end{pmatrix}, \qquad w = \begin{pmatrix} w_1(n_1) \\ w_2(n_2) \\ w_3(n_3) \end{pmatrix},$$

where n_1=MN, n_2=NKP, n_3=NK. Then, $x = Bv$ becomes

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} B_{11}v_1 \\ B_{33}v_2 \\ 0 \end{pmatrix}. \tag{2.9}$$

Secondly, the blocked LU decomposition of A is calculated before the first call to ARPACK. If no pivoting is needed then an LU factorization of A can be written as

$$\begin{pmatrix} A_{11} & 0 & A_{15} \\ 0 & A_{33} & A_{35} \\ A_{51} & A_{53} & 0 \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ 0 & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & 0 & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}, \tag{2.10}$$

where L_{ij} are lower triangular matrices and U_{ij} are upper triangular matrices. Unknown triangular matrices L_{ij} and U_{ij} satisfy the following equations:

$$\begin{aligned} L_{11}U_{11} &= A_{11} \\ L_{22}U_{22} &= A_{33} \\ L_{11}U_{13} &= A_{15} \\ L_{22}U_{23} &= A_{35} \\ L_{31}U_{11} &= A_{51} \\ L_{32}U_{22} &= A_{53} \\ L_{33}U_{33} &= -L_{31}U_{13} - L_{32}U_{23} \end{aligned}$$

L_{11}, U_{11}, L_{22}, U_{22} can be formed by factoring A_{11} and A_{33}. (Notice that we can still do

pivoting within these blocks.) U_{13}, U_{23}, L_{31}, and L_{32} can be expressed as

$$\begin{aligned} U_{13} &= L_{11}^{-1} A_{15} \\ U_{23} &= L_{22}^{-1} A_{35} \\ L_{31} &= A_{51} U_{11}^{-1} \\ L_{32} &= A_{53} U_{22}^{-1} \end{aligned} \tag{2.11}$$

and hence, can be computed by solving a series of triangular systems. Finally, L_{33} and U_{33} can be formed by computing and factoring $-L_{31}U_{13} - L_{32}U_{23}$.

The block structure can also be exploited in solving the linear systems $Ly = x$ and $Uw = y$. From the lower triangular system

$$\begin{pmatrix} L_{11} & 0 & 0 \\ 0 & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ 0 \end{pmatrix},$$

y can be computed as

$$\begin{aligned} y_1 &= L_{11}^{-1} x_1 \\ y_2 &= L_{22}^{-1} x_2 \\ y_3 &= L_{33}^{-1}(-L_{31} y_1 - L_{32} y_2) \end{aligned} \tag{2.12}$$

From the upper triangular system

$$\begin{pmatrix} U_{11} & 0 & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix},$$

w can be computed as

$$\begin{aligned} w_3 &= U_{33}^{-1} y_3 \\ w_2 &= U_{22}^{-1}(y_2 - U_{23} w_3) \\ w_1 &= U_{11}^{-1}(y_1 - U_{13} w_3) \end{aligned} \tag{2.13}$$

Combining Equations (2.12) and (2.13), w can be written in the following form

$$\begin{aligned} w_3 &= U_{33}^{-1}L_{33}^{-1}(-A_{51}U_{11}^{-1}L_{11}^{-1}x_1 - A_{53}U_{22}^{-1}L_{22}^{-1}x_2) \\ w_2 &= U_{22}^{-1}L_{22}^{-1}(x_2 - A_{35}w_3) \\ w_1 &= U_{11}^{-1}L_{11}^{-1}(x_1 - A_{15}w_3) \end{aligned} \tag{2.14}$$

Blocked matrix-vector multiplication algorithm (2.14) can be used to compute the matrix-vector multiplication (2.8). The detailed computation steps as implemented are shown in the following:

1. LU decomposition $A_{11} = L_{11}U_{11}$, it needs $\frac{2}{3}n_1^3$ flops.
2. LU decomposition $A_{33} = L_{22}U_{22}$, it needs $\frac{2}{3}n_2^3$ flops.
3. Forming $\bar{A} = -A_{51}A_{11}^{-1}A_{15} - A_{53}A_{33}^{-1}A_{35}$, it needs $2(n_1^2n_3 + n_1n_3^2 + n_2^2n_3 + n_2n_3^2)$ flops.
4. LU decomposition $\bar{A} = L_{33}U_{33}$, it needs $\frac{2}{3}n_3^3$ flops.
5. Matrix-vector product $x_1 = B_{11}v_1$, it needs $2n_1^2$ flops.
6. Matrix-vector product $x_2 = B_{33}v_2$, it needs $2n_2^2$ flops.
7. Solve linear system $L_{11}U_{11}y_1 = x_1$, it needs $2n_1^2$ flops.
8. Solve linear system $L_{22}U_{22}y_2 = x_2$, it needs $2n_2^2$ flops.
9. Matrix-vector product $y_3 = -A_{51}y_1 - A_{53}y_2$, it needs $2(n_1n_3 + n_2n_3)$ flops.
10. Solve linear system $L_{33}U_{33}w_3 = y_3$, it needs $2n_3^2$ flops.
11. Matrix-vector product $y_2 = x_2 - A_{35}w_3$, it needs $2n_2n_3$ flops.
12. Matrix-vector product $y_1 = x_1 - A_{15}w_3$, it needs $2n_1n_3$ flops.
13. Solve linear system $L_{22}U_{22}w_2 = y_2$, it needs $2n_2^2$ flops.
14. Solve linear system $L_{11}U_{11}w_1 = y_1$, it needs $2n_1^2$ flops.

The first four operations are computed only one time. The remaining operations are needed in each iteration. The total number of flops is

$$\begin{aligned} &\tfrac{2}{3}(n_1^3+n_2^3+n_3^3)+2n_3(n_1^2+n_2^2+n_1n_3+n_2n_3) \\ &+2k(3n_1^2+3n_2^2+n_3^2+2n_1n_3+2n_2n_3), \end{aligned} \tag{2.15}$$

where k is the number of iterations. The number of flops of non-block matrix-vector multiplication (Equation (2.3)) is given by

$$\tfrac{2}{3}n^3+4kn^2=\tfrac{2}{3}(n_1+n_2+n_3)^3+4k(n_1+n_2+n_3)^2. \tag{2.16}$$

When we choose the values of parameters as in Section 2.1.2, then n_1=900, n_2=500, and n_3=625. With k=50 iterations, the operation counts in expressions (2.15) and (2.16) are 3.68 and 6.36 gflops, respectively. The blocked matrix-vector product saves 42% of floating point operations. Table 2.11 compares the execution time of non-blocked and blocked algorithms on two problem sizes.

Table 2.11: Time (seconds) to solve the eigenvalue system.

Problem size	non-block	block	non-block/block
450	9.8	4.5	2.2
920	77.0	32.9	2.3

Another advantage of using blocked matrix-vector multiplication is that the large arrays AMAT and BMAT are no longer needed. Instead, the small matrices AMAT11, BMAT11, COPY15, and COPY35 are used. For the parameters used in the original SPANDO package, the block oriented algorithm saves memory size of

$$65.6-2\times 1.3-4.5-2.5=56 \text{ Mb}.$$

2.7 Summary of Results from New Sequential SPANDO

In this section we summarize all the improvements made to sequential SPANDO by reporting improvements in overall time and space requirements.

In the new SPANDO, instead of using IMSL, ARPACK is employed to solve the generalized eigenvalue system. The nonzero structure of sparse matrices AMAT and BMAT is exploited. To improve matrix-matrix multiplication performance, the BLAS (Basic Linear Algebra Subprograms) subroutine DGEMM is called. For the same parameters given in Section 2.1.2, the new SPANDO saves $98.4 + 26.5 + 56 = 180.9$ Mb memory, which is 76% of original SPANDO. It is now possible to compute a bigger problem by using the new SPANDO. Table 2.12 gives two examples to show that the new SPANDO can solve bigger problems by using the same memory size (234 Mb).

Table 2.12: Problem sizes that can be solved by new SPANDO.

	M	N	NSTIF	K	$K^{'}$
Original	30	30	25	25	20
Example 1	40	30	30	45	45
Example 2	30	30	40	40	30

Table 2.13 summarizes the comparison between execution time of the original SPANDO and the new SPANDO. We see that the new SPANDO reduces execution time by well over a factor of 10, and that the improvement grows with problem size.

Table 2.14 summarizes the improvement from each change discussed in this chapter. The problem size used is 920×920. From Table 2.14 we can see that the three major improvements are using BLAS, using ARPACK, and the block algorithm.

Table 2.13: SPANDO execution time

Problem size	Old package (sec)	New package (sec)	Old/NEW
350	72	7	10.3
450	153	12	12.8
920	1370	64	21.4

Table 2.14: The results of each improvement.

Improvement	Total time (sec)	Old/New
Original	1,370	1.0
Using BLAS	1,025	1.3
Using LAPACK	1,017	1.3
Using ARPACK	114	12.1
Block algorithm	64	21.4

Chapter 3

Parallel SPANDO

The modified sequential SPANDO can be used to calculate bigger problems, but the required computation time and memory size still increases very quickly when the problem size increases. Furthermore, to find the optimal solution of a problem, SPANDO may need to solve hundreds of eigenvalue problems. So, for solving big problems, or modest problems with optimization, a parallel SPANDO package is clearly of interest.

Table 3.1 lists the memory size and execution time in seconds for solving two different size problems by using the modified sequential SPANDO package. These results show that

Table 3.1: New sequential SPANDO performance.

Size	Memory (Mb)	M-forming	M-condensing	E-solving	Total time
920	17	14	10	15	43
1420	34	39	37	48	128

memory requirements and execution time increase much faster than problem size. Also, almost all of the execution time is used in: (1) forming the matrices, (2) the condensing procedure, and (3) solving the generalized eigenvalue system. Parallel computing technology can be used in these three parts to decrease execution time and per-processor (i.e., "local") memory size requirements.

The goal of this chapter is to introduce the parallel SPANDO package. We first introduce two libraries used in the parallel SPANDO code: a communication library (BLACS) and a parallel linear algebra library (ScaLAPACK). The process of parallelizing SPANDO is then described. We give a few performance results here to illustrate the improvements made.

More complete performance results are reported in Chapter 4.

3.1 The Communication Library: BLACS

The Basic Linear Algebra Communication Subprograms (BLACS) [3] is a software package developed at the University of Tennessee, Knoxville. The purpose of BLACS is to create a linear algebra oriented message passing interface that is implemented efficiently and uniformly across a large range of distributed memory platforms.

At present, four different BLACS implementations are available. These four implementations are based on four different message passing layers: MPI, MPL, NX, and PVM. MPI and PVM are used in most systems, while MPL is used in IBM's SP series, and NX is for Intel's supercomputer series (iPSC2, DELTA, and Paragon). Our research work is on an Intel Paragon with BLACS based on NX.

The major reasons we choose BLACS as the parallel SPANDO communication library is that we use a parallel linear algebra library (ScaLAPACK; see next section) which is based on BLACS. Also, code modifications required to change platforms are minimal because the BLACS are available on a wide variety of parallel computers.

There are two important features of BLACS we exploit:

Two-dimensional array based communication. Many communication packages, like MPI and PVM, are based primarily on one dimensional arrays. BLACS, however, makes it much more convenient to operate on 2D arrays. For our application, it is more natural to express most communication operations in terms of matrices.

Scoped operations. If there are N_p processes involved in a parallel task, these processes are often presented as a 1D array, labeled 0,1,..., $N_p - 1$. For solving linear algebra problems, it is often more convenient to map this 1-D array of N_p processes into a logical two dimensional grid. A process can now be referenced by its coordinates within the grid. Using a 2D grid, we have three natural scopes: all processes, a row

of processes, and a column of processes. BLACS provides broadcast operations based on these three scopes.

Next, we describe two broadcast subroutines that are used extensively in parallel SPANDO. If a process P_{ij} intends to broadcast an integer M by N matrix A to all other processes (or to the processes that are in the same row or column), the subroutine

IGEBS2D(ICONTXT, SCOPE, TOP, M, N, A, LDA)

is called. If A is a real (or double precision) matrix, then subroutine RGEBS2D (or DGEBS2D) is called. The difference between them is the first character of the subroutine name. I, R, D stand for integer, real, and double precision, respectively. The parameters of the subroutine are as follows:

- ICONTXT: (Integer) The BLACS context handle. In the BLACS, each logical process grid is enclosed in a context. A context may be thought of as a message passing universe.
- SCOPE: (Character) Scope of processes to participate in operation. Limited to 'ROW', 'COLUMN', or 'ALL'.
- TOP: (Character) Network topology to be emulated during communication. In broadcasts, the BLACS provide both pipelining and non-pipelining topologies. The default TOP option (which is invoked by setting TOP=' ') will attempt to use the topology that minimizes the cost of one call to a combine operation.
- M: (Integer) The number of matrix rows to be broadcast.
- N: (Integer) The number of matrix columns to be broadcast.
- A: (Type array) A pointer to the beginning of the (sub)array to be broadcast.
- LDA: (Integer) The leading dimension of the matrix A.

For a process to receive a broadcast matrix, the subroutine

vGEBR2D(ICONTXT, SCOPE, TOP, M, N, A, LDA,I,J)

is called, where v expresses the type of A. The parameters I and J are the process coordinates of the source of the broadcast.

In parallel SPANDO, the above two subroutines (broadcast send and broadcast receive) are used in the input subroutine STIF0. When parallel code runs on multiprocessors, it is not always possible for each processor to read input data by itself. Instead, only one processor, say processor P_{00} reads input data from disk, and then broadcasts the data to all other processors. Because a SPANDO input file has many 2D array, BLACS is convenient and efficient for broadcasting this input data. There is another subroutine (the matrix-vector multiplication subroutine used in ARPACK) that explicitly uses BLACS broadcast to transfer data among processors; we will discuss it later in this chapter.

3.2 The Linear Algebra Library: ScaLAPACK

The Scalable Linear Algebra PACKage (ScaLAPACK) [2] is the parallel version of LAPACK [1]. The purpose of ScaLAPACK is to achieve high performance in a portable way over a large class of distributed-memory MIMD computers. ScaLAPACK is written in Fortran 77 in a Single Program Multiple Data (SPMD) style. The BLACS is used for interprocessor communication.

At the present time, ScaLAPACK is portable across the following distributed-memory architectures: Intel series (NX), IBM SP series (MPL), Thinking Machines CM-5 (CMMD), and clusters of workstations via PVM and MPI.

To understand how ScaLAPACK's computational routines are used in parallel SPANDO, we must first understand how the data is distributed over the processors. Then, in Section 3.2.2 we list the ScaLAPACK routines used.

3.2.1 Data Distribution

The way the data is distributed over the processors is of fundamental importance to the performance of parallel software on distributed memory machines. The data in SPANDO consists of several large matrices. Two often used algorithms for distributing matrices are block-partitioned algorithm and cyclic-partitioned algorithm. The block-partitioned algorithm takes a block of data and assigns the data to some processor. By using block-partitioned algorithm, the frequency of data transferred between processors is reduced and the local node performance is maximized. The cyclic-partitioned algorithm assigns contiguous data entries to successive different processors. This algorithm helps to maintain load balance when the computational load is not homogeneous. ScaLAPACK adopts the block-cyclic distribution algorithm, a hybrid of block– and cyclic–partitioning, to optimize its performance.

As we discussed in Section 3.1, the 1-D array of N_p processors can be mapped into a logical two dimensional processor grid. Let P be the number of rows in the grid and Q be the number of columns, where P×Q=$N_g \leq N_p$. The block-cyclic distribution procedure is described in the following.

1. An M by N matrix A is first decomposed into MB by NB blocks A_{ij}. The blocks in the last processor row may have fewer than MB rows, and the blocks in the last processor column may have fewer than NB columns.

2. Each block A_{ij} is mapped to the processor in the (i mod P, j mod Q) position of the grid. Thus every processor owns a collection of blocks, which are locally and contiguously stored in a two dimensional array.

An example matrix A(7,7) is distributed as follows. Let MB=NB=2, P=Q=2. First, A is logically partitioned into 2 by 2 blocks

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}

Then, this block-partitioned matrix is mapped to the processor grid:

a_{11}	a_{12}	a_{15}	a_{16}	a_{13}	a_{14}	a_{17}
a_{21}	a_{22}	a_{25}	a_{26}	a_{23}	a_{24}	a_{27}
a_{51}	a_{52}	a_{55}	a_{56}	a_{53}	a_{54}	a_{57}
a_{61}	a_{62}	a_{65}	a_{66}	a_{63}	a_{64}	a_{67}
a_{31}	a_{32}	a_{35}	a_{36}	a_{33}	a_{34}	a_{37}
a_{41}	a_{42}	a_{45}	a_{46}	a_{43}	a_{44}	a_{47}
a_{71}	a_{72}	a_{75}	a_{76}	a_{73}	a_{74}	a_{77}

The distributed matrix A can be described by the following parameters:

1. CTXT: the BLACS context in which this matrix is defined.
2. M: the number of rows in the distributed matrix A.
3. N: the number of columns in the distributed matrix A.
4. MB: the row block size.
5. NB: the column block size.
6. RSRC: the process row over which the first row of A is distributed.
7. CSRC: the process column over which the first column of A is distributed.

8. LLD: the leading dimension of the local array storing the local blocks.

All these parameters are encapsulated in an integer array called an *array descriptor*. It is the user's responsibility to distribute the matrix and create the matrix descriptor before using ScaLAPACK computational routines.

3.2.2 ScaLAPACK Computational Routines Used in Parallel SPANDO

Six ScaLAPACK computational routines are called in parallel SPANDO. They are

1. PDGETRF: Computes an LU factorization of a real general distributed matrix, using partial pivoting with row interchanges.

2. PDGETRS: Solves a real general system of distributed linear equations AX=B, using the LU factorization computed by PDGETRF.

3. PDGEMV: Performs matrix-vector operations y := α Ax + βy, where α and β are scalars, x and y are distributed vectors and A is an distributed matrix.

4. PDGEMM: Performs matrix-matrix operations C := α A $\times$ B + βC, where α and β are scalars, and A, B and C are real general distributed matrices.

5. PDLANGE: Computes the one norm, or the Frobenius norm, or the infinity norm, or finds the element of largest absolute value of a distributed matrix.

6. PDLASCL: multiplies a M by N real distributed matrix by a real scalar.

3.3 SPANDO Parallelization Procedure

Table 3.1 shows the bottlenecks of the sequential SPANDO code are forming the matrices, condensing the system, and solving the generalized eigenvalue system. In this section, we discuss in detail the strategy used in parallelizing each of these three parts.

3.3.1 Generating the Distributed Matrices

Suppose there are $P \times Q = N_p$ processors used in the computation, where P is the number of rows in the processor grid, and Q is the number of columns in the processor grid. As we mentioned in Section 3.1, processor P_{00} reads input data and broadcasts it to all other processors. Each processor now generates its own parts of the distributed matrices $A_{11}, A_{24}, \cdots, B_{33}$ by following three steps:

1. Declare local arrays: If a matrix A(M,N) is to be distributed over the processor grid, every processor in the grid should declare array A(locM,locN), where

$$\text{locM} = \text{MB}((M-1)/(P \times \text{MB})) + \text{mod}((M-1), \text{MB}) + 1,$$
$$\text{locN} = \text{NB}((N-1)/(Q \times \text{NB})) + \text{mod}((N-1), \text{NB}) + 1,$$

   and MB and NB are the number of rows and columns in each block.

2. Create array descriptors for each matrix by using ScaLAPACK subroutine DESCINIT.

3. Generate local matrix A's entries with the following code:

```
DO J=1,N
  compute processor column for matrix column J
  IF(My processor is in this column) THEN
    DO I=1,M
      compute processor row for matrix row I
      IF(My processor is in this row) THEN
        compute local position locX(I), locY(J)
        calculate A(I,J) and store in A(locX(I),locY(J))
      END IF
    END DO
  END IF
END DO
```

The global entry A(I,J) is stored in processor $P_{p,q}$, where p and q are computed by

$$p(I) = \text{mod}((I-1)/\text{MB}, P),$$
$$q(J) = \text{mod}((J-1)/\text{NB}, Q).$$

The local coordinates locX and locY are given by

$$\text{locX}(I) = \text{MB} * ((I-1)/(P * \text{MB})) + \text{mod}((I-1), \text{MB}) + 1,$$

$$\text{locY}(J) = \text{NB} * ((J-1)/(Q * \text{NB})) + \text{mod}((J-1), \text{NB}) + 1.$$

When we increase the number of processors, the required local memory size and the times used for generating matrices both improve. Complete performance results are shown in Chapter 4. Table 3.2 gives a typical result when the problem size is 2460.

Table 3.2: Time used for creating matrices.

Processors	Memory Size	Matrix Creation Time
4	31.2 Mb	35 sec
8	20.9 Mb	23 sec
16	12.8 Mb	13 sec

On the Intel Paragon at Virginia Tech there are 100 processors, each with 32 Mb of local memory. Because the matrices are distributed over the processors, the more processors used, the less local memory is required. So, it is possible to compute a bigger problem by using multiprocessors. For different number of processors, Table 3.3 lists the biggest problem sizes that can be computed by using at most 32 Mb of local memory. When the number of processors increases by a factor of four, the biggest problem size increases by a factor of two.

Table 3.3: The biggest global problem size requiring 32 Mb of local memory.

Number of processors	1	4	16	64
Problem size ($\times 10^3$)	1.2	2.4	4.8	9.6

There are approximately 200 Mb of virtual memory available to each local processor (a limit caused by swap space limits). Virtual memory can be used to solve much bigger problems. But because the speed of virtual memory page swapping is very slow, the user will suffer an unpredictably long computational time.

3.3.2 Condensing the Eigenvalue System

In Section 2.5 we discussed an improved procedure for reducing the generalized eigenvalue system from Equation (1.1) to (1.2). It is straightforward to parallelize this condensing procedure. Instead of calling LAPACK computational routines, the corresponding ScaLAPACK computational routines are called to compute LU decompositions, triangular solves, and matrix-matrix multiplications. Table 3.4 shows the time used in parallel SPANDO's condensing procedure for a typical example. The problem size tested is 2460. More results are given and analyzed in Chapter 4.

Table 3.4: The time (seconds) used for parallel condensing procedure.

Processors	Linear solves	M-M multiplications	Total condensing time
8	36	16	52
16	23	8	31

Unlike LAPACK, where the computational routines use an optimal block size automatically, in ScaLAPACK the block size must be provided by the user. In the above test, the block size is MB=NB=32. The effect of block size on performance will be discussed in Section 4.1.

3.3.3 Solving the Generalized Eigenvalue System

Sequential ARPACK is still used to solve the eigenvalue system in the parallel SPANDO package. In our application, most of the computational work for solving the eigenvalue system is in the matrix-vector multiplication $w = A^{-1}Bv$. For example, for a test problem of size 920, 97% of the sequential time needed to solve the eigenvalue problem is spent in the matrix-vector multiplication. Recall that the matrix-vector subroutine is provided by the user. So without changing any other ARPACK routine, a parallel matrix-vector subroutine should achieve a good parallel result for solving the generalized eigenvalue system. In fact, there is a parallel version of ARPACK, called PARPACK [12]; but because our problem sizes

are not big enough, the parallel version of ARPACK is not used in parallel SPANDO code. (In fact, using PARPACK gives "slowdown," rather than speedup for this application.)

To solve the generalized eigenvalue problem over $P \times Q$ processors, the following steps are taken:

1. All matrices A and B (A_{11}, A_{33}, ...) are distributed over all processors.

2. In iteration k, all processors redundantly compute vector v_k by calling ARPACK routine DNAUPD. Each processor has a complete copy of v_k.

3. Processors in the first column of the processor grid rewrite v_k to ScaLAPACK format—permuting v_k over the first column of processors in order to use ScaLAPACK computational routines. There is no communication in this step.

4. ScaLAPACK computational routines are used to compute system (2.14). The result is the vector w_k distributed across the first column of processors.

5. Processors in the first column broadcast their local vector w_k to all other processors.

6. All processors assemble a complete copy of w and then return w to the main iteration loop.

An example of the time used for solving the generalized eigenvalue system is shown in Table 3.5. The problem size is 2460. We find the parallel eigensolver speeds up well. More extensive performance results are given and analyzed in the next chapter.

Table 3.5: Time (seconds) used for solving the generalized eigenvalue system.

Processors	LU decomp	M-V mult	ARPACK routine	broadcast	Total
4	73	67	1	5	146
8	41	28	1	4	74
16	23	28	2	4	57

Note that the ARPACK computation could be done only in the first processor column, so that communication of w_k to all processors is not needed. We only need to broadcast to the first processor column. We have not tested this improvement, but the experimental results show that the communication time in our implementation is not significant.

Chapter 4

Parallel SPANDO Performance Analysis

In this chapter, we first describe several factors that may affect the performance of parallel SPANDO. Then, in Section 4.2 we present experimental results and analyze these factors. Finally, in Section 4.3 the overall performance of parallel SPANDO is described in terms of fixed problem size speedup and scalability.

4.1 Factors Affecting the Performance of Parallel SPANDO

The following factors may affect the performance of parallel SPANDO:

Number of computed eigenvalues. When using ARPACK, the user must specify how many eigenvalues should be computed. The number of computed eigenvalues will almost certainly affect the execution time.

Iteration tolerance. When using ARPACK, the user may choose a proper iteration tolerance. By default, ARPACK uses machine epsilon as iteration tolerance. Usually, a bigger iteration tolerance will degrade the precision of the results, but it also reduces the iteration time, thus reducing the computational time.

Block size. In ScaLAPACK, matrices are distributed over processors. The block-cyclic algorithm is used for distributing data; thus, the block size may affect the performance of ScaLAPACK, and hence of parallel SPANDO.

Shape of processor grid. We logically map the 1D processor array to a 2D processor grid. For the same number of processors, we may use different grid shapes. For example, 4 processors may be mapped to 1 by 4, or 2 by 2, or 4 by 1 grids. This factor

changes the data distribution among processors. So, it may affect the performance of parallel SPANDO.

Structure of AMAT. When testing the performance of our system, by "size of the problem" we mean the size of the generalized eigenvalue system: the size of the matrix AMAT, given by LDMAT=MND+NKD+NKPD. The parameters MND, NKD, and NKPD determine the shape (structure) of AMAT and BMAT. When the shape of AMAT is changed, the computation time is changed. In our tests, we only use the input files from the research group of Engineering Science and Mechanics Department of Virginia Tech. Hence, we did not study the affect of this factor since it would have required us to have available more problem instances.

4.2 Experimental Results and Analysis of Factors

In this section, we report on experiments designed to study the effect of the factors described above. All experiments are executed on an Intel Paragon. Table 4.1 describes the characteristics of the Intel Paragon at Virginia Tech.

Table 4.1: Characteristics of the Intel Paragon.

Processor	i860 XP
Clock speed (MHz)	50
Compute nodes	100
Memory per node (Mb)	32
BLAS	Version 5.0
BLACS	NX BLACS 1.1
Communication Software	NX
Fortran compiler	if77
Fortran flag	-O4
Precision	double (64-bit)

In the following we study the effect of every individual factor; that is, we fix all factors except the factor that is studied. All experiments are run on 16 processors. The problem

parameters used in the test case are given in Table 4.2.

Table 4.2: Parameters used in test file.

M	N	NSTIF	K	K'	Total size
30	30	26	30	30	2460

The Effect of Number of Computed Eigenvalues

Five values for number of eigenvalues computed are considered, with block size BS equals 64, a 4×4 processor grid, and iteration tolerance 10^{-6}. Figure 4.1 shows that the the execution time increases linearly, but slowly, with eigenvalue number. We can choose

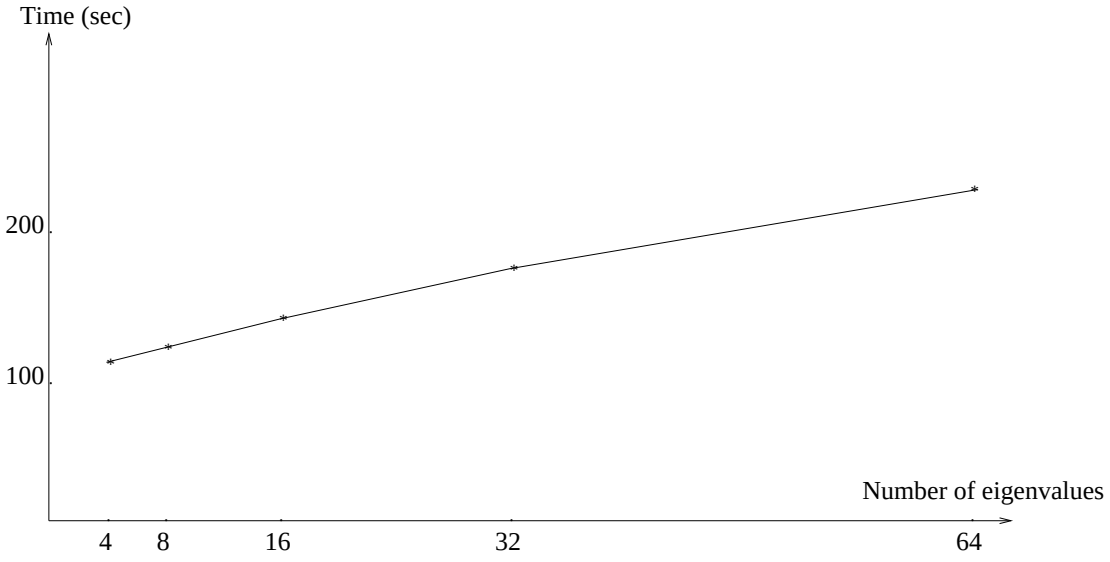

Figure 4.1: Running times for different number of eigenvalues.

a reasonably big number of eigenvalues to ensure that the biggest negative eigenvalue and the smallest positive eigenvalue are included in this set.

Table 4.3: Running times for different iteration tolerance.

Tolerant number	10^{-16}	10^{-12}	10^{-10}	10^{-8}	10^{-6}
Number of M-V multi	130	110	106	97	93
Time (sec)	151	141	139	134	132

The Effect of Iteration Tolerance

Let the number of computed eigenvalues be 16 and all other conditions be the same as in the previous experiment. Table 4.3 shows the effect of iteration tolerance. The results show that execution time is not very sensitive to iteration tolerance.

The Effect of Block Size

In ScaLAPACK, most of the computation routines are performed in a blocked fashion using Level 3 BLAS. A smaller block size is helpful for load balance but increases the frequency of communication between processes and may cause the cache on each processor to be used less efficiently. In the following experiments, we fix the computed eigenvalue number (16) and use 10^{-6} as iteration tolerance. Figure 4.2 describes the effect of block size. The results show that the execution time is very sensitive to block size changes. Choosing 32 – 64 as block size produces good performance.

The Effect of Processor Grid

So far in this section, all experiments are executed over 16 processors which are mapped to a 4×4 grid. Now, the effect of processor grid is studied. We can map 16 processors to five different 2D grids as indicated in Table 4.4. The data in Table 4.4 suggests that using

Table 4.4: The effect of processor grid.

Processor grid	1×16	2×8	4×4	8×2	16×1
Time (sec)	255	183	134	135	134

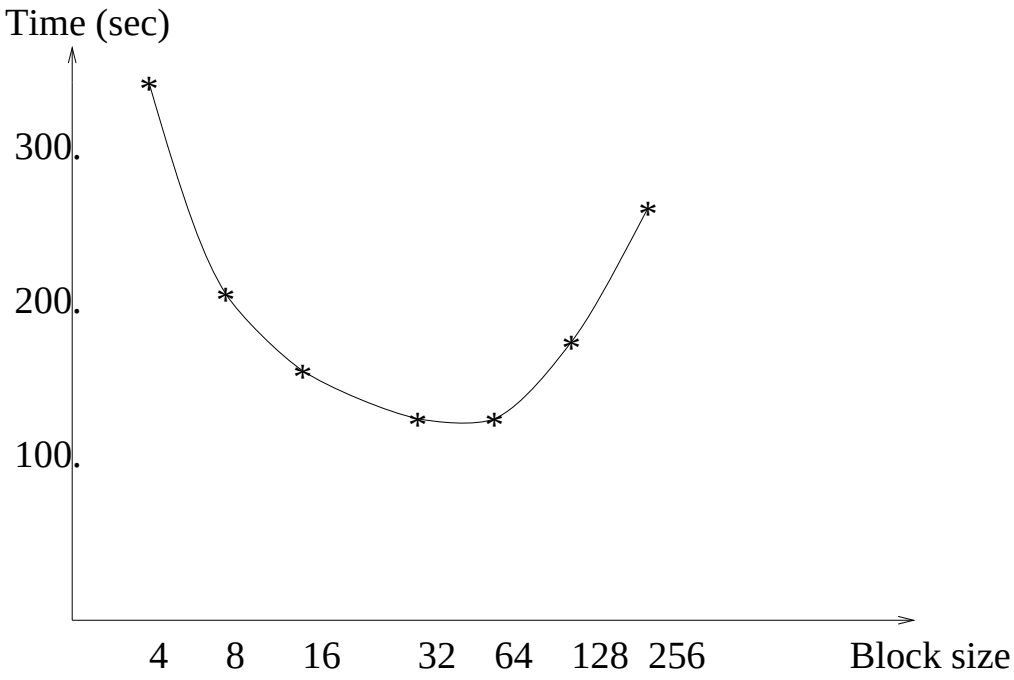

Figure 4.2: Running times for different block size.

square or 'thin' processor grids will give better performance. In fact, our experiments show that the LU decompositions and matrix-matrix multiplications perform better for square grids, but when we solve the generalized eigenvalue system, there are hundreds of matrix-vector multiplications. The 'thin' grid will achieve a better performance for computing these matrix-vector products.

4.3 Overall Parallel Performance

4.3.1 Fixed–Size Speedup

In this section, we study the performance of parallel SPANDO. Three input files are used in our experiments. Table 4.5 describes the parameters in these files. The experimental

Table 4.5: The parameters of the test cases.

Problem	Total size	M	N	NSTIF	K	KP
1	1,200	20	20	20	20	20
2	2,400	40	20	20	40	40
3	4,800	40	40	20	80	80

results are given in Tables 4.6—4.8. For each problem we report local memory requirements, execution time, and speedup (over the next smaller case) for three different numbers of processors. Two conclusions are obvious:

1. The problem size can be doubled when the number of processors increases four times.

2. For fixed problem size, the execution time is reduced by at least a factor of 1.5 when the number of processors is doubled, but the performance degrades when we use "too many" processors to solve a "small" problem.

The experimental results do not show a perfect speedup—the computation time is not cut in half when the number of processors is doubled. This is because the local matrix sizes are too small. In our experiments, the maximum local matrix size is 400×400 (e.g., for A_{11} in Equation (2.10)). When the number of processors is doubled, the maximum local matrix size is reduced to 200×400. ScaLAPACK can not achieve good performance with this matrix size. In fact, according to the ScaLAPACK Users' Guide [2], good parallel performance typically requires local matrix size greater than 1000×1000.

We can not test bigger problems because the local memory size of our Paragon is 32 Mb. Our results suggest that parallel SPANDO will perform better on larger problems on a parallel computer which has more local memory.

Ironically, one possible way to get larger local matrix sizes with the current machine is to *not* use the block-oriented algorithm described in Section 2.6. This approach does not take advantage of the sparse structure of AMAT; that is, in this algorithm, we compute the matrix-vector multiplication $y = A^{-1}Bx$ directly. In Chapter 2 we compared the non-block

Table 4.6: The experiment results for problem 1.

Processor number	Memory size	Execution time	Speed up
1 (1×1)	31 Mb	96 sec	1
2 (2×1)	19 Mb	64 sec	1.5
4 (2×2)	13 Mb	49 sec	2.0

Table 4.7: The experiment results for problem 2.

Processor number	Memory size	Execution time	Speed up
4 (2×2)	31 Mb	233 sec	1
8 (4×2)	20 Mb	152 sec	1.5
16 (4×4)	13 Mb	135 sec	1.7

Table 4.8: The experiment results for problem 3.

Processor number	Memory size	Execution time	Speed up
16 (4×4)	32 Mb	603 sec	1
32 (8×4)	20 Mb	360 sec	1.7
64 (8×8)	14 Mb	359 sec	1.7

algorithm and the block oriented algorithm, and the conclusion is that the block oriented algorithm saves over 40% of flops in our case. But since a big local matrix size can achieve a better parallel result, we also consider this non-block version of parallel SPANDO. The only difference between these two algorithms is in solving the generalized eigenvalue system. Table 4.9 compares the parallel performance for solving the generalized eigenvalue system. The results show that for almost all our experiments the block oriented algorithm is still better than the non-block algorithm, although the advantage shrinks as the number of processors grows.

Table 4.9: The time used for solving generalized eigenvalue system.

Problem size	Processors	Non-block	Block
1200×1200	1	54 sec	39 sec
	2	45 sec	31 sec
	4	35 sec	27 sec
2400×2400	4	128 sec	116 sec
	8	101 sec	89 sec
	16	82 sec	88 sec
4800×4800	16	300 sec	276 sec
	32	217 sec	207 sec

4.3.2 Scalability

Three types of matrix operations in parallel SPANDO are considered in this section. These operations account for almost all of the work in the condensing procedure and the eigen-solve. Recall from Table 3.1 that these two steps correspond to about two-thirds of the sequential running time—the remaining time being spent in generating the matrices. We do not consider the scalability of matrix generation here because our scalability analysis requires floating point operation counts (flops) and these are difficult to estimate for the matrix generation step. However, matrix generation parallelizes very easily, requiring no communication, so its scalability should be excellent.

By considering the scalability of the three dominant linear algebra computations, we can estimate the scalability of parallel SPANDO as a whole. The three operations are:

1. LU decomposition. There are five LU decompositions in SPANDO: two of them come from the matrix condensing procedure, the other three come from solving the generalized eigenvalue system. The total number of floating-point operations in LU decomposition is
$$\frac{4}{3}NK^3 + \frac{2}{3}MN^3 + \frac{2}{3}NK^3 + \frac{2}{3}NKP^3.$$

2. Matrix-matrix operations. These operations include matrix-matrix multiplications and triangular matrix solves (computing C=A^{-1}B, where A, B, and C are matrices and an LU factorization of A is known). There are five triangular solves, with three coming from the matrix condensing procedure and two coming from solving the generalized eigenvalue system. Also, there are six matrix-matrix multiplications, four of them from the matrix condensing procedure and the other two from solving the generalized eigenvalue system. The total number of floating-point operations in matrix-matrix operations is given by
$$6MN \times NK^2 + 4NK^3 + 4MN^2 \times NK.$$

3. Matrix-vector operations. These operations include matrix-vector multiplication and simple triangular solves (y=A^{-1}x, where x and y are vectors, and A's LU decomposition is known). The total number of flops in matrix-vector operations is
$$2k(3MN^2 + 3NK^2 + NKP^2 + 2MN \times NKP + 2NK \times NKP),$$
where k is the number of ARPACK iterations.

We study the scalability of each operation. The measure used here is mflops per second. An algorithm is considered scalable if the mflop rate per processor remains relatively constant as the number of processors is increased, with local problem size held relatively

constant. If the computational rate per processor decreases, it is most likely due to increased communication costs. The input test files used here are the same as in Section 4.3.1. Tables 4.10 to 4.12 show the total number of flops and running time of each type of operation. Tables 4.13 to 4.15 give mflop rates per processor for fixed local memory size.

The following conclusions can be derived from the experimental results.

1. For each of the three matrix operations, the mflop rate increases with local problem size. And also, it is more scalable for a bigger local problem size.

2. For each of the three matrix operations, the mflop rate decreases with the number of processors.

3. For a fixed local problem size, when the number of processors is increased by a factor of 16, the mflop rate of LU decomposition and matrix-matrix operations is decreased by only a factor of 2; the mflop rate of matrix-vector operations is decreased by a factor of 5.

4. For a large local problem size, the LU decomposition performs better than matrix-matrix and matrix-vector operations, but for a modest or small local problem size, the matrix-matrix operations perform better than the other two.

5. From Table 4.10 to 4.12 we see that the most expensive of the three operations are the matrix-matrix and matrix-vector operations. So, overall parallel SPANDO scalability is more heavily influenced by the scalability of these two operations than by *LU* factorization. Roughly speaking, for a fixed local problem size, when the number of processors is increased by a factor of 16, the mflop rate for matrix-matrix and matrix-vector operations is decreased by a factor of 3.5. Overall, SPANDO's scalability will be slightly better (recall that matrix generation scales much better than this).

Table 4.10: Number of flops and running time of LU decomposition.

Processors	Memory size	Mflop of LU	Time for LU
1 (1×1)	31 Mb	213	7 sec
2 (2×1)	19 Mb	107	5 sec
4 (2×2)	13 Mb	53	4 sec
4 (2×2)	31 Mb	426	18 sec
8 (4×2)	20 Mb	213	14 sec
16 (4×4)	13 Mb	107	13 sec
16 (4×4)	32 Mb	853	45 sec
32 (8×4)	20 Mb	426	39 sec
64 (8×8)	14 Mb	213	34 sec

Table 4.11: Number of flops and running time of matrix-matrix operation.

Processors	Memory size	Mflop of M-M	Time for M-M
1 (1×1)	31 Mb	896	35 sec
2 (2×1)	19 Mb	448	20 sec
4 (2×2)	13 Mb	224	12 sec
4 (2×2)	31 Mb	1,792	91 sec
8 (4×2)	20 Mb	896	50 sec
16 (4×4)	13 Mb	448	40 sec
16 (4×4)	32 Mb	3,584	261 sec
32 (8×4)	20 Mb	1,792	163 sec
64 (8×8)	14 Mb	896	145 sec

Table 4.12: Number of flops and running time of matrix-vector operation.

Processors	Memory size	Mflop of M-V	Time for M-V
1 (1×1)	31 Mb	352	19 sec
2 (2×1)	19 Mb	176	17 sec
4 (2×2)	13 Mb	88	16 sec
4 (2×2)	31 Mb	454	52 sec
8 (4×2)	20 Mb	227	53 sec
16 (4×4)	13 Mb	114	57 sec
16 (4×4)	32 Mb	475	122 sec
32 (8×4)	20 Mb	238	107 sec
64 (8×8)	14 Mb	119	137 sec

Table 4.13: Mflop rate per processor with local problem size = 31 Mb.

Processors	LU decomposition	M-M operation	M-V operation
1	30.4	25.6	18.5
4	23.7	19.7	8.7
16	19.0	13.7	3.9

Table 4.14: Mflop rate per processor with local problem size = 20 Mb.

Processors	LU decomposition	M-M operation	M-V operation
2	21.4	22.4	10.4
8	15.2	17.9	4.3
32	10.9	10.8	2.2

Table 4.15: Mflop rate per processor with local problem size = 13 Mb.

Processors	LU decomposition	M-M operation	M-V operation
4	13.3	18.7	5.5
16	8.2	11.2	2.0
64	6.2	6.2	0.9

Chapter 5

Summary and Conclusion

In this research, we apply modern high performance computing techniques to solve an engineering problem, structural analysis of grid stiffened panels, with good success. Two new SPANDO packages, a modified sequential SPANDO and parallel SPANDO, are developed.

In developing the new sequential SPANDO, we investigate different numerical packages and find that LAPACK performs well for solving the required linear algebra problems: matrix inversion, LU decomposition, and triangular solve. As an eigenvalue solving package, ARPACK is very suitable for solving our generalized eigenvalue system because: (1) ARPACK computes a few user specified eigenvalues and corresponding eigenvectors; (2) ARPACK provides a reverse communication interface so that the user can take advantage of matrix structure.

In these two new SPANDO packages we use a new block-oriented algorithm to compute the matrix-vector multiplication $w = A^{-1}Bx$. This block-oriented algorithm is used in the ARPACK iterations. For saving memory requirements, a new condensing procedure for reducing the size of the eigenvalue system is introduced.

The experimental results show that the new sequential SPANDO achieves a significantly higher performance in terms of memory requirements and execution time. Compared with the original SPANDO, the new SPANDO saves over 70% of memory size, and is at least 10 times faster. Now, it is possible to solve a bigger problem by using a sequential computer, or to solve many small problems in more reasonable time.

Based on the new sequential SPANDO, a parallel version of SPANDO is also developed. The parallel version of SPANDO uses multiprocessors to solve a problem faster by working

together. In parallel SPANDO a widely used communication library (BLACS) is used to interchange data among processors. A parallel linear algebra library (ScaLAPACK) is used to compute linear algebra problems like LU decomposition, solving the triangular systems, etc. Because the matrices are distributed on different processors, the local memory requirement is decreased when the number of processors is increased. In fact, when the number of processors is increased by a factor of 4, the problem size can be doubled. So, parallel SPANDO can be used to solve bigger problems if enough processors are available.

There are many factors that may affect the performance of parallel SPANDO; experiments are designed to study the affects of these factors. The results show that:

1. The execution time increases linearly, but slowly, with number of computed eigenvalues.

2. The execution time is not very sensitive to iteration tolerance used in ARPACK.

3. To distribute data on processors, choosing 32–64 as block size produces good performance.

4. Using square or "thin" processor grid topologies achieves the best performance.

The overall parallel performance of SPANDO is also studied. The experimental results show that:

1. For a fixed global problem size, the execution time is reduced by at least a factor of 1.5 when the number of processors is doubled, for modest numbers of processors.

2. For a fixed local problem size, when the number of processors is increased by a factor of 16, the mflop rate per processor of matrix-matrix operations is decreased by a factor 2, and the mflop rate of matrix-vector operations is decreased by a factor of 5.

Parallel SPANDO can be used to solve current problem sizes very quickly, although it does not pay to use more than about 16 processors for these small problems. We also have good

evidence that parallel SPANDO would be able to solve much larger problems in reasonable time, but larger per-node memory sizes are required.

Our research results illustrate that modern high performance computing techniques are very important for solving engineering application problems. Developing new computing technology (e.g., parallel computing) and standard software packages (e.g., LAPACK, ARPACK, ScaLAPACK, etc.) is crucial to satisfying the requirements of large engineering applications. On the other hand, it also important for computer scientists, who trace and develop new computing techniques to work with engineers to solve real world problems.

REFERENCES

[1] E. Anderson, Z. Bai, ..., *LAPACK Users' Guide*, SIAM, Philadelphia, 1995.

[2] L. S. Blackford, ..., *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

[3] J. J. Dongarra, R. C. Whaley, *A User's Guide to the BLACS v1.1*, 1997.

[4] J. J. Dongarra, J. DuCroz, S. Hammarling, R. J. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Softw., Volume 14, 1988.

[5] J. J. Dongarra, J. DuCroz, S. Hammarling, I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., Volume 16, 1990.

[6] M. Embree, C. Ribbens, *On the Scalability of Parallel Krylov Subspace Methods* Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[7] B. Grall, Z. Güdal, *Structural Analysis an Design of Geodesically Stiffened Composite Panels with Variable Stiffener Distribution*, Technical Report, Virginia Tech, 1992.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Softw., Volume 5, 1979.

[9] R. Lehoucq, *Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration*, Ph.D Thesis, Rice U., 1995. Available from www.caam.rice.edu/software/ARPACK.

[10] R. B. Lehoucq, J. A. Scott, *An Evaluation of Software for Computing Eigenvalues of Sparse Nonsymmetric Matrices*, 1996.
Available from www.caam.rice.edu/software/ARPACK.

[11] R. B. Lehoucq, D. C. Sorensen, C. Yang, *ARPACK Users' Guide*, 1997.

[12] K. J. Maschhoff, D. C. Sorensen, *A Portable Implementation of ARPACK for Distributed Memory Parallel Architectures*, 1996. Available from
www.caam.rice.edu/software/ARPACK.

[13] G. Soremekun, Z. Gürdal, C. Kassapoglou, B. Owen, *SPANDO User's Manual*, final report for sikorsky aircraft contract, 1996.

[14] D. C. Sorensen, *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations* Rice University, 1995. Available from
www.caam.rice.edu/software/ARPACK.

VITA

Shaohong Qu was born in Tianjing, China, in 1962. He graduated from the University of Science and Technology of China (USTC), with a M.S. in mathematics in 1988. After graduation, he worked in the USTC for 6 years. He then came to Virginia Tech to continue his educations in mathematics and computer science in 1994. Shaohong Qu completed his studies and received his M.S. in computer science in Fall 1997