To the Graduate Council:

I am submitting herewith a thesis written by Theresa Hong Do entitled "Sequential and Data-Parallel Implementations of a Lanczos Algorithm for the Singular Value Decomposition." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael W. Berry, Major Professor

We have read this thesis
and recommend its acceptance:

Jack Dongarra

Jean R. S. Blair

Accepted for the Council:

Lew Minkel

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Th Do

Date 12/17/92

Sequential and Data-Parallel Implementations of a Lanczos Algorithm for the Singular Value Decomposition

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Theresa Hong Do

May 1993

Acknowledgements

I thank my thesis advisor, Dr. Michael Berry, for his support and guidance. I appreciate very much his wise counsel, endless enthusiasm, and patience. I also thank Jack Dongarra and Jean Blair who served on my thesis committee. I am grateful to Andrew Ogielski for the opportunity to intern at Bellcore. The experience has been invaluable.

This research has been supported in part by Apple Computer through Contract No. C24-9100120.

Abstract

In this thesis, we present both sequential and data-parallel implementations of the single-vector Lanczos algorithm for computing the singular value decomposition of large unstructured sparse matrices. Our intent is to produce robust numerical software in C that is portable across a variety of high-performance workstations such as the IBM RS/6000, DEC 5000-200, Sun Sparcstation 2, and Apple Macintosh IIfx. Furthermore, this software should be easily incorporated into larger application programs such as those from large-scale information retrieval applications which require the computation of singular values and singular vectors of sparse matrices. Using the MasPar Programming Language on the MasPar MP-1 computer system, we develop a systematic approach for redesigning our software for a data-parallel environment. We approximate several of the largest singular values and singular vectors of a test suite of sparse matrices using two different equivalent eigensystems and provide benchmark and performance comparisions for each machine considered.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we present sequential C and MasPar Programming Language (MPL) implementations of the single-vector Lanczos algorithm from SVDPACK ([Berr92a]), which can be used to compute the singular value decomposition (SVD) of large unstructured sparse matrices. The motivation of our work is discussed below and pertinent background theory is presented in Chapter 2. Chapter 3 describes the computing environments of interest, and in Chapters 4 and 5 we describe the implementations and design methodology used. Performance results are presented in Chapter 6 along with a discussion on the selection of input parameters for obtaining optimal performance. Finally, in Chapter 7 we state our conclusions along with suggested future work.

1.1 Motivation

The need for computing the singular value decomposition traditionally arises in many areas such as physics, chemistry, and engineering. In information science, Deerwester

et al. ([DDF+90]) demonstrated the use of the SVD in an information retrieval technique called Latent Semantic Indexing (LSI). They assume that there is some underlying latent structure in word usage data of a document. The SVD is used in LSI to estimate this latent structure and remove the obscuring noise which arises from the variability of word choice. In doing so, LSI addresses two problems (synonymy and polysemy) that plague existing information retrieval techniques which use literal term matching. Synonymy is a problem that occurs when a concept can be described by more than one word. When the words used in a query do not match those in a document, the search fails regardless of whether the query and the document refer to the same concept or not. On the other hand, polysemy occurs when the terms in the query match those in a document even though they contextually mean different things. Thus, the search is "successful" and irrelevant documents are returned to the user.

LSI consists of two phases. In the first phase, an SVD method from SVDPACK is normally used to analyze the input matrix, a term by document matrix in which each cell indicates the frequency with which each term occurs in each document. The second phase involves the query matching process in which a cosine measure is used as a similarity indication between a given query and documents in the collection; when a query is received, the cosines between the query vector and document vectors (right singular vectors) are computed and documents, ordered by their *distance* to the query, are returned. The SVDPACK routines used in the first phase were originally written in Fortran-77. We chose to implement them in C because this language has desirable features that are lacking in Fortran-77, such as dynamically-allocated memory and pointers, and because the software used for text parsing, indexing, and retrieval is all written in C. Furthermore, there are computing environments in which a Fortran-77

compiler is not readily available; one such environment is the Apple Macintosh IIfx for which some of this work was supported. The query-matching phase of LSI is interactive, and therefore, minimizing the processing time is important. Since the cosine between the query and each document vector can be computed independently and concurrently, this step has been implemented in parallel using MPL in the MasPar MP-1. For this reason, we chose to also implement the SVD algorithms used in the first phase (pre-processing) of LSI in MPL.

1.2 Statement of Problem

Definition

Given an $m \times n$ matrix A, where $m \geq n$ and $\text{rank}(A) = r$, the singular value decomposition of A, denoted by SVD(A), is defined as

$$A = U\Sigma V^T , \tag{1.1}$$

where $U^TU = V^TV = I_n$ and $\Sigma = diag(\sigma_1, \ldots, \sigma_p)$, $\sigma_i > 0$ for $1 \leq i \leq r$, $\sigma_j = 0$ for $j \geq r + 1$. The first r columns of the orthogonal matrices U and V define the orthonormalized eigenvectors associated with the r nonzero eigenvalues of AA^T and A^TA, respectively. The singular values of A are defined as the diagonal elements of Σ which are the nonnegative square roots of the n eigenvalues of AA^T. The set $\{u_i, \sigma_i, v_i\}$ is called the i-th singular triplet ([Berr92a]).

Problem

Given the sparse $m \times n$ matrix A, determine the p-largest singular triplets of A as defined by (1.1) using a serial or data-parallel computer.

Chapter 2

Preliminaries

2.1 Theory

Given a real symmetric $p \times p$ matrix B, the standard symmetric eigenvalue problem is defined by

$$Bx = \lambda x \,, \tag{2.1}$$

where λ is a scalar, and x is a nonzero vector. An eigenpair of B is the set $\{x, \lambda\}$, where λ is an eigenvalue, and x is its corresponding eigenvector.

Given a general rectangular $m \times n$ matrix A as descibed in Section 1.2, a singular triplet of A, $\{u, \sigma, v\}$, is defined by the singular value σ and its corresponding left and right singular vectors (of unit norm) u and v, respectively, such that

$$Av = \sigma u \,, \tag{2.2}$$

$$A^T u = \sigma v \,. \tag{2.3}$$

Associated with the matrix A is the $p \times p$ symmetric 2-cyclic matrix ([Varg62])

$$B = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}, \text{ where } p = m + n. \tag{2.4}$$

We can compute $SVD(A)$ indirectly by computing the eigenvalues and eigenvectors of B. From (2.1), we have

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ or}$$

$$\begin{bmatrix} Ax_2 \\ A^T x_1 \end{bmatrix} = \begin{bmatrix} \lambda x_1 \\ \lambda x_2 \end{bmatrix}, \tag{2.5}$$

which corresponds to the definition of the SVD given in (2.2) and (2.3) with $\lambda = \sigma$, $x_1 = u$, and $x_2 = v$. From (2.5), we see that an eigenvector of B comprises both a left and right singular vector of A, and the positive eigenvalues of B[1] are the singular values of A. We can assess the numerical accuracy of the i-th singular triplet $\{u_i, \sigma_i, v_i\}$ computed via the equivalent eigensystem of B by computing the 2-norm of the eigenpair residual vector r_i corresponding to (2.1) and (2.2)

$$\|r_i\|_2 = (\|Av_i - \sigma_i u_i\|^2 + \|A^T u_i - \sigma_i v_i\|^2)^{\frac{1}{2}}. \tag{2.6}$$

$SVD(A)$ may also be computed by computing eigenpairs of the equivalent eigen-

[1] Eigenvalues of B constitute the set $\{-\sigma_n, \ldots, -\sigma_1, 0, \sigma_1, \ldots, \sigma_n\}$

value problem in AA^T or A^TA. In the latter case,

$$B = A^TA\,, \tag{2.7}$$

and from (2.1), we have

$$A^TAx = \lambda x\,, \tag{2.8}$$

where x and λ are an eigenpair of B. It follows from (2.2) that

$$u = \frac{1}{\sigma}Av\,, \tag{2.9}$$

and rewriting (2.3) using (2.9) yields

$$A^Tu = \frac{1}{\sigma}A^TAv = \sigma v\,, \text{ or}$$

$$A^TAv = \sigma^2 v\,. \tag{2.10}$$

Comparing (2.10) with (2.8), we see that eigenvectors of B are right singular vectors of A and $\sigma^2 = \lambda$, which implies the singular values of A are the positive square roots of the eigenvalues of B. The residual norm of the i-th eigenpair of the A^TA system can be determined by

$$\|r_i\|_2 = \frac{1}{\sigma_i}\|A^TAv_i - \sigma_i^2 v_i\|^2. \tag{2.11}$$

2.2 The Single-Vector Lanczos Algorithm

The Lanczos procedure, originally intended for tridiagonalizing symmetric matrices, is an effective method for solving large sparse eigenvalues problems ([PaSc79]). In this method, a given real symmetric matrix is transformed into a sequence of real symmetric tridiagonal matrices T_j whose eigenvalues and corresponding eigenvectors approximate those of the original matrix.

Let B be the $(m+n)\times(m+n)$ matrix in (2.4) or the $(n\times n)$ matrix in (2.7), where A is the $m\times n$ matrix for which $SVD(A)$ is desired, and r_0 be a randomly-generated vector of length $(m+n)$ or n such that $\|r_0\|_2 = 1$. For $j = 1, 2, \ldots, k$ define the corresponding Lanczos matrices T_j using the following recursion ([GoVa89]). Define $\beta_0 \equiv \|r_0\|_2$, and $q_0 \equiv 0$. Then for $i = 1, 2, \ldots, k$, define $Q_j = (q_1, q_2, \ldots, q_j)$, where q_j is called a *Lanczos vector*, and scalars α_i and β_i so that

$$\begin{aligned} q_j &= r_{j-1}/\beta_{j-1}\,, \\ \alpha_j &= q_j^T B q_j\,, \\ r_j &= Bq_j - \alpha_j q_j - \beta_{j-1} q_{j-1}\,, \text{ and} \\ \beta_j &= \|r_j\|_2\,. \end{aligned} \tag{2.12}$$

The equations in (2.12) form the basic *Lanczos recursion*, or *Lanczos step*. For each j, the corresponding Lanczos matrix T_j is defined as the real symmetric tridiagonal matrix with diagonal entries α_i, $1 \le i \le j$, and subdiagonal (superdiagonal) entries β_i, $1 \le i < j$ so that

$$T_j \equiv \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & . & . & & \\ & & . & . & . & \\ & & & . & . & \beta_{j-1} \\ & & & & \beta_{j-1} & \alpha_j \end{bmatrix}.$$

Furthermore, the Lanczos vectors (columns of Q_j) generated by the recursion form an orthonormal basis for the subspace $\{q_1, Bq_1, B^2q_1, \ldots, B^{j-1}q_1\}$, which is also known as the j-dimensional *Krylov subspace*, $\mathcal{K}^j(q_1, B)$, generated by B and q_1. In other words, the tridiagonal matrix T_j is the orthogonal projection of B onto the Krylov

subspace $\mathcal{K}^j(q_1, B)$ ([Parl80]). The eigenvalues and eigenvectors of T_j, also called the *Ritz values* and *Ritz vectors*, respectively, are then used as approximations to eigenvalues and eigenvectors of B.

As discussed in [CuWi85], for exact arithmetic, orthogonality with respect to only the two most recently-generated Lanczos vectors is sufficient to guarantee that each succeeding Lanczos vector is orthogonal with respect to all previously-generated Lanczos vectors. However, in finite arithmetic, losses in the orthogonality of the Lanczos vectors occur as the recursion proceeds. They are due to a combination of the effects of the roundoff errors together with the convergence of one or more of the eigenvalues of the Lanczos matrices T_j as j is increased. Since total re-orthogonalization at each step is too costly in terms of memory storage and time, iterative Lanczos methods typically use some schemes of selective re-orthogonalization. The single-vector Lanczos routines in SVDPACK employ a selective re-orthogonalization strategy (LANSO) originally proposed by [PaSc79].

During the entire execution, the Lanczos procedure performs a predetermined number of iterations and then restarts if the desired number of singular triplets is not yet found or if the number of iterations completed is still less than the maximum number allowed ([Berr92b]). In each iteration, a Lanczos step is taken and the next Lanczos vector is orthogonalized against the two most recent Lanczos vectors. Orthogonalization estimates of all Lanczos vectors are then updated, and if the largest estimate is greater than $\sqrt{\epsilon}$ ($\epsilon \equiv$ *machine epsilon*), the current and next Lanczos vectors are orthogonalized against all previous vectors. Otherwise, the procedure continues with the next iteration. A flow chart of the Lanczos procedure is given in Appendix C.

The Lanczos procedure has two important properties which make it attractive for solving large sparse eigensystems. First, we note that throughout the algorithm, the

input matrix B given by (2.4) or (2.7) remains unchanged and is referenced only through the matrix-vector multiplication Ax or x^TA. This implies that an efficient storage scheme which exploits the sparsity of A can be used. Secondly, we see that the basic recursion (2.12) performed in each iteration requires only that two most recently-generated Lanczos vectors be stored locally. The remainder may be kept on a secondary I/O medium and fetched for selective re-orthogonalization when needed. This property proves to be very useful, as we shall see, in a computing environment in which physical memory is very limited.

In SVDPACK and in this work, both the 2-cyclic and the A^TA implementations of the Lanczos procedure are included. Typically, the latter is preferred since the approximation of an n-order eigensystem is computationally less demanding than that of an $(m+n)$-order eigensystem by the 2-cyclic implementation. In some cases, due to the memory constraint in the environment and the problem size, the A^TA implementation may be the only option for computing $SVD(A)$. However, as discussed in [Berr92a], this implementation can be unstable for computing the smallest or extremely close eigenvalues of B (singular values of A). Since the matrix A^TA can be very ill-conditioned (square of the condition of A), the 2-cyclic version which is based on the 2-cyclic matrix B in (2.4) may be considered as a more robust implementation even though it is considerably slower.

Chapter 3

Computing Environments

3.1 overview

The work presented in this thesis is developed for sequential and data-parallel computing environments. In this chapter, we describe both environments in which the ANSI-C and MPL single-vector Lanczos codes were written and the specific architectures used to obtain the timings and results presented in Chapter 6. Table 3.1 lists the architectures used in this work.

Table 3.1: Architectures used in sequential and parallel computing environment.

Machine	Environment
IBM RS/6000-550 Sun Sparcstation 2 DECstation 5000-200 Macintosh IIfx	sequential
MasPar MP-1	parallel

3.2 Sequential Computing Environment

The architectures that represent our sequential computing environment range from a Macintosh IIfx to an IBM RISC workstation. The first machine in our sequential environment is an IBM RS/6000 model 550 workstation whose architecture is based on the RS6000 chip set. Its theoretical computation rate and clock speed are 83 MFLOPS and 41.5 MHz, respectively. It also has 128 Mbytes of RAM. We also have a Sun Sparcstation 2 with SunOS 4.1.1 operating system. With a clock speed of 40 MHz, it is capable of a peak computation rate of 4.1 MFLOPS and has 64 Mbytes of RAM. The third machine in this environment is a DECstation 5000 model 200 RISC workstation. Its architecture includes R3000 RISC CPU, an R3010 floating-point unit, and 32 Mbytes of RAM. It achieves 24 integer MIPS on the Dhrystone benchmark and 3.1 MFLOPS on the LINPACK benchmark (double precision). Finally, we have a 40 MHz-Macintosh IIfx, which has a 32-byte architecture with an MC68030 processor, an MC68882 floating-point unit, and 32 Mbytes of RAM.

3.3 Parallel Computing Environment

Representing our parallel environment is the massively-parallel MasPar MP-1 machine. The number of processors in a MasPar MP-1 ranges from 1024 to 16384

arranged in a rectangular grid. Our particular machine[1] has 4096 processor elements. Instructions are issued to the processors one at a time but each instruction is carried out concurrently in the PE array on different data. This typical Single Instruction Multiple Data (SIMD) is common to all data-parallel machines. The MasPar MP-1 consists of a front-end and a back-end. Its front-end is the DEC 5000-200 from the sequential environment described in the previous section. The back-end, the Data Parallel Unit (DPU), comprises the Array Control Unit (ACU) and the processor element (PE) array (see Figure 3.1 and [MasP92]). The main function of the ACU is to control the PE array by sending data instructions to each PE simultaneously. It has a register-based load/store processor, 32 32-bit registers, 128 Kbytes of data memory, and 1 Mbyte of instruction memory. The ACU can also serve as a general computing machine, albeit not a very powerful one, operating on variables declared in its memory space. These variables are referred to as *singular* or *scalar* data as opposed to *parallel* data discussed below.

Each PE in the PE array is a RISC-based load/store arithmetic processor with its own dedicated memory space which includes 40 32-bit registers and 16 Kbytes of

[1] Supported by the Joint Instituted for Computational Science (JICS) of the University of Tennessee, Knoxville.

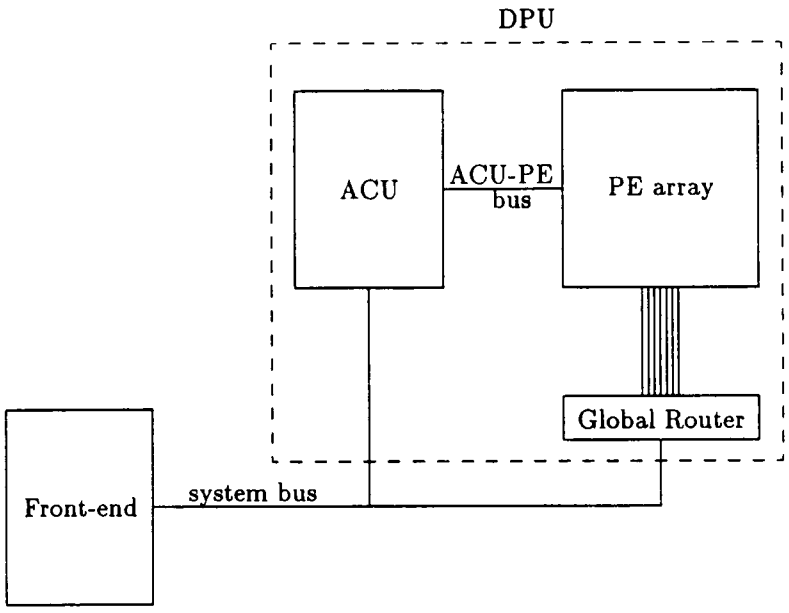


Figure 3.1: MasPar MP-1 system diagram.

RAM. Variables that are declared in PE memory are called parallel variables since memory for these variables is identically allocated in each PE. In this thesis, we use the word *parallel* and *plural* interchangeably when referring to variables allocated in the PEs.

In the MasPar MP-1 series, the PEs are laid out in a rectangular mesh with toroidal wrap at the physical boundaries of the mesh. With this arrangement, each PE, including those at the edge of the physical array, has eight neighbors. Communication between the PEs and the ACU is achieved via a dedicated ACU-PE bus while communication among the PEs in the PE array is implemented by X-Net and Global Router communications. X-Net communications allows a PE to communicate with any other PE that lies on a straight line from the original PE in the north, northeast, east, southeast, south, southwest, west, or northwest direction. Through Global Router

communications, any particular PE can communicate with members of an arbitrary subset of PEs simultaneously. The latter is slightly slower but it allows for more flexible communications since it does not have built-in directions of communication as does the X-Net construct.

Chapter 4

C Code Design

4.1 Overview

SVDPACK is a Fortran-77 library ([Berr92a]) for computing the SVD of large sparse matrices. It comprises four methods, one of which is based on the single-vector form of the basic Lanczos recursion in (2.12). In this chapter, we discuss the sequential C version of this method while the parallel version in MPL is described in the following chapter. In each case, we have two implementations that resolve the equivalent 2-cyclic and A^TA eigensystems in order to compute several of the largest singular values and corresponding singular vectors of an unstructured sparse $m \times n$ matrix A. We shall describe the input and output files required by the codes and discuss user-defined routines in detail. Listed in Table 4.1 are different versions of the single-vector SVD codes that are available.

Table 4.1: Available implementations of single-vector Lanczos codes.

Language implemented	Code Name	
	2-cyclic	A^TA-based
C	**LAS1C**	**LAS2C**
MPL	**LAS1P**	**LAS2P**
Fortran-77	**LAS1**	**LAS2**

4.2 Input and Output Files

Listed in Table 4.2 are the input files required and output files produced by each implementation of the codes. **LAS1C** and **LAS2C** require the input matrix to be stored in an

Table 4.2: Input and output files.

Code Name	Input Files	Output Files
LAS1C	matrix	lav1
LAS1P	lap1	lao1
LAS2C	matrix	larv2, lalv2
LAS2P	lap2	lao2

ASCII file in the column-compressed sparse matrix format used in the Harwell/Boeing matrix collection ([DuGL89]). The file consists of three linear arrays having starting indices of 1, as it was originally intended for Fortran-77 software. The third array, `value[]`, stores the nonzero values of the matrix traversing all elements column-wise. In other words, each subsequent nonzero in column-wise traversal is assigned a *reduced* index which is its location in the array. The second array, `rowind[]`, contains the row index of each matrix element stored in `value[]`. The first array, `pointr[]`, contains the reduced index of each column-start nonzero in the matrix. Thus, for an $m \times n$ matrix, this array will have $n + 1$ elements, where for the first n locations, the i-th location stores the reduced index of the element that starts the i-th column.

The $(n+1)$-th location will contain $nnzeros + 1$, where $nnzeros$ is the number of nonzeros.

Even though we use the column-compressed sparse matrix format which is most suitable for Fortran-77, our C codes are readily adaptable to accept any other sparse matrix format. We note that since array indices begin with 0 in C, array elements of `rowind[]` and `pointr[]` are decremented by 1 immediately after they are read in. For example, the "sparse" matrix

$$\begin{bmatrix} 0 & 0 & a_{02} \\ a_{10} & 0 & 0 \\ 0 & a_{21} & 0 \\ a_{30} & 0 & a_{32} \end{bmatrix}$$

will be implicitly stored in three arrays described above as follows:

array	reduced index:	0	1	2	3	4
`value[]`		a_{10}	a_{30}	a_{21}	a_{02}	a_{32}
`rowind[]`		1	3	2	0	3
`pointr[]`		0	2	3	5	

Another input file required by `LAS1C` and `LAS2C` is the parameter file which contains eight fields of constants and switches on a single line:

<name> lanmax maxprs endl endr vectors kappa memory

where

- *<name>* is a string defining the name of the dataset.
- *lanmax* is an integer specifying the maximum number of iterations allowed.

- *maxprs* indicates the number of singular triplets of A (eigenpairs of the equivalent matrix B) desired.

- *endl,endr* are two end-points of an interval within which all unwanted eigenvalues of the particular matrix B lie.

- *vectors* contains the string `TRUE` or `FALSE` to indicate when singular triplets are needed (`TRUE`) and when only singular values are needed (`FALSE`).

- *kappa* contains the relative accuracy of Ritz values acceptable as eigenvalues of the matrix B.

- *memory* is a single integer between 0 and 7 with bit representation $b_1 b_2 b_3$ such that each bit b_i specifies a particular memory management option:

  b_1: When this bit is set to 0, an ASCII file containing input matrix is expected. Otherwise, input matrix is to be read from the standard input. This is useful when the matrix file is very large and must be kept compressed. In setting this bit, the user on a Unix system can `zcat` the file to produce the ASCII text on the standard output, which can then be piped to the standard input.

  b_2: This bit is considered only when the flag *vectors* is set to `TRUE`. If it is 0, the singular vectors are stored in memory as well as in a binary file. Otherwise, they are stored only in a binary file on disk and they are read from the file as needed.

  b_3: When this bit is set to 0, memory is allocated for storing the Lanczos vectors as they are generated by the Lanczos recursion. Otherwise, these vectors are written to and retrieved from a binary file which exists only through the duration of the run.

As an example,

```
BELLADI  350  100  -1.e-30  1.e-30  TRUE  1.0e-6  3
```

indicates that the dataset `BELLADI` contains the input sparse matrix whose 100-largest singular triplets are sought using at most 350 iterations. The input matrix is to be read from the standard input and the singular vectors are to be stored in binary file (bits b_2 and b_3 of *memory* are set). After each run, `LAS1C` and `LAS2C` produce ASCII output files `lao1` and `lao2`, respectively, which contain information about the input matrix, the computed Ritz values which approximate the eigenvalues of B, their relative errors, the computed singular values of A, and the corresponding residual norms ((2.6) and (2.11)). If singular vectors are requested, as in the above example, `LAS1C` produces the binary output file `lav1` that contains a left singular vector followed by a right singular vector for each of the singular values calculated and sorted in ascending order. `LAS2C` produces two binary output files: `larv2` which contains the right singular vectors and `lalv2` which contains the left singular vectors. Again, these are ordered to correspond with the singular values sorted in ascending order.

4.3 User-Defined Routines

In order for the user to incorporate `LAS1C` and `LAS2C` into his application with ease, there are several user-defined routines[1] that may be tailored to suit a particular computing environment. As we shall see, this flexibility facilitates the porting of the C codes to the parallel environment of the MasPar.

[1] See Appendix B for prologues of selected functions described in this section

init()

At the beginning of execution, `LAS1C` and `LAS2C` each have an initializing routine called `init()` which performs the following functions:

- read in the sparse matrix
- check validity of input parameters
- create and open output files
- allocate necessary memory for the entire run
- write header to output file (header information consists of dimensions and order of matrix, name of dataset, etc.)

The format of the input matrix and desired output files may vary from one application to another and should be changed accordingly.

Matrix-Vector Multiplication

We previously noted that the single-vector Lanczos algorithm does not modify the input matrix A, which is referenced only through matrix-vector multiplication. We provide `opb()` which performs sparse matrix-vector multiplication for `LAS1C` and `LAS2C`. For the latter, we also provide the kernel `opa()`. Table 4.3 lists these kernels, and where appropriate, their specific function in each implementation. The modules are specifically designed for an input matrix or its *transpose* stored in the column-compressed format in three linear arrays which were described in Section 4.2. In Table 4.4, we present an outline of the matrix-vector multiplication kernel used in `LAS1C` assuming matrix A is stored. In the case where the transpose of A is

Table 4.3: Matrix-vector multiplication kernels for **LAS1C** and **LAS2C**.

code	opb()	opa()
LAS1C	$y = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} x$	-
LAS2C	$y = A^T A x$	$y = Ax$

stored, obvious changes must be made. Step 1 of Table 4.4 can be considered as a *sparse saxpy* operation with indirectly-accessed elements of $\hat{y}$ updated by the product $(value_j \times x_{i+m})$. In Step 2, a *sparse dot product* between $value_j$ and corresponding

Table 4.4: Matrix-vector multiplication kernel of **LAS1C**.

$$y = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} x,\ y = (\hat{y}^T, \bar{y}^T)^T,\ x = (\hat{x}^T, \bar{x}^T)^T.$$

1. $\hat{y} = A\bar{x}$

for $k = 0, \cdots, m+n-1$ do:
$y_k = 0,$
for $i = 0, \cdots, n-1$ do:
for $j = pointr_i, \cdots, pointr_{i+1} - 1$ do:
$y_{rowind_j} = y_{rowind_j} + value_j \times x_{i+m}$

2. $\bar{y} = A^T\hat{x}$

for $i = 0, \cdots, n-1$ do:
for $j = pointr_i, \cdots, pointr_{i+1} - 1$ do:
$y_{i+m} = y_{i+m} + value_j \times x_{rowind_j}$

elements of $\hat{x}$ is performed. Since the matrix A is stored in the column-compressed sparse matrix format (instead of row-compressed format), data locality is better exploited by the sparse dot product when we multiply by the transpose of A. The user

should provide appropriate matrix-vector kernels comparable to these if the input matrix is not in this form.

store()

One major advantage in using C as opposed to Fortran-77 is its ability to allocate memory dynamically. In using Fortran-77 to compute the singular triplets, we must a priori set our array sizes large enough to accommodate the largest matrix to be considered or reset the static memory in the code to be the appropriate size for each dataset and recompile each time. The size of the memory that is dynamically allocated in `LAS1C` and `LAS2C` is a function of *lanmax* described above and the order of the matrix B, N, which is n for the A^TA case and $(m+n)$ for the 2-cyclic case. Table 4.5 shows the breakdown of the core memory needed and Table 4.6 lists the memory requirement of the codes for some of the datasets used in this thesis. Specifically, we

Table 4.5: Memory requirement in 8-byte units (double words).

Code Name	Memory Allocated for...		
	Lanczos Vectors	Singular Vectors	Workspace
LAS1C	$N(lanmax+2)$	$N(j+1)+$ j^2	$9N+4\times lanmax+1+$ $(n+nnzeros+1)/2+nnzeros$
LAS2C	$N(lanmax+2)$	$(N\times j)+m+$ j^2	$9N+4\times lanmax+m+1+$ $(n+nnzeros+1)/2+nnzeros$

j = number of iterations taken
$nnzeros$ = number of nonzeros in input matrix

see that `LAS1C` and `LAS2C` must store $(lanmax+2)$ Lanczos vectors of size N during execution. Prior to the Lanczos recursion (2.12), the q_j's are created and stored, and are repeatedly fetched throughout the recursion. We have provided a user-defined function called `store()` to perform the reading and writing of these vectors. Their storage location, whether in main memory, in a secondary storage medium such as

Table 4.6: Memory required for sample datasets in megabytes.

Dataset Size	Lanczos Vectors	Singular Vectors	Workspace	Total	Code
bellmedt	27.57	29.51	1.14	58.22	LAS1C
5831 × 1033	4.15	6.18	.77	11.09	LAS2C
bellcist	26.52	28.46	1.29	56.28	LAS1C
5143 × 1460	5.86	7.88	.96	14.71	LAS2C
belltect	93.06	94.87	5.64	193.57	LAS1C
16637 × 6535	26.24	28.27	4.57	59.09	LAS2C

$j = 500$
$lanmax = 500$

a disk, or even in main memory of another processor, is transparent to the calling program of `store()`. The caller simply calls `store()` with a request to read (write) a particular vector and `store()` handles the request accordingly.

From Table 4.5, we note that in the case where the user chooses to store the Lanczos vectors and the singular vectors out-of-core, the amount of memory needed is substantially smaller. This allows him to compute the SVD of much larger sparse matrix problems, but at the cost of high latency associated with secondary storage (disk) I/O.

Chapter 5

MPL Code Design

5.1 Overview

Even though we present a single parallel version for each of the single-vector Lanczos codes `LAS1P` and `LAS2P` from Table 4.1, there are two variations of each code. In one case, we are interested in parallelizing only the matrix-vector multiplication since this is usually the most time-consuming operation in the algorithm (see Figure 5.1). In the

Ticks	%	Histogram	Routine
52970	25		`opb`
52772	24		`imtql2`
45632	22		`daxpy`
25546	12		`dcopy`
13200	6		`ddot`
8164	3		`opa`
9096	8		*others*

Figure 5.1: DPU profile of `LAS2C` showing the time accumulated by each routine in clock ticks and in percent.

second case, the Lanczos recursion (2.12) along with the calculations of the singular

values and residual norms in (2.6) and (2.11) (if singular vectors are requested) are also computed in the DPU. For clarity, we refer to the parallel codes as `LAS1P` and `LAS2P` without making a distinction between the variations unless it is necessary to do so. In this chapter, we shall present our methodology for porting code from a sequential environment to a parallel environment, describe the required I/O files, and discuss some primitives and user-defined functions in detail.

5.2 MPL Implementations

The MasPar MP-1 systems support several programming languages among which is MPL. We chose to implement the Lanczos codes in MPL since it is based on ANSI-C. An extended version of ANSI-C, MPL can be used to program the DPU in a similar manner to that in which C is used in the sequential environment. In this section, we begin with a detailed description of the methodology used to port the C codes of the single-vector Lanczos algorithm to the massively-parallel MasPar MP-1. We will also look at several primitives and user-defined modules written specifically for the MPL implementations `LAS1P` and `LAS2P`. Table 5.1 gives the definitions of some system variables in the MPL environment that are used throughout this section.

Table 5.1: System variables in MPL.

Variable name	Type	Definition
`iproc`	`plural int`	self-address of each PE
`nproc`	`int`	number of processors in PE array
`nxproc`	`int`	number of processors in X dimension
`nyproc`	`int`	number of processors in Y dimension

5.2.1 Methodology for Porting

In anticipation of the porting of the C codes to the MasPar MP-1, we wrote `LAS1C` and `LAS2C` with much emphasis on data abstraction, data encapsulation and modularity; a well-designed ANSI-C program lends itself well to systematic porting and maximum reuse. Currently, C compilers in the massively-parallel environment lack the sophistication needed for converting arbitrary sequential C code into a parallel C code. Good software engineering approaches, such as an object-oriented methodology, are the best interim solution to facilitate the porting of C code between the sequential and parallel environment or between two parallel machines. In this approach to portability, the program structure of a typical C code written for a sequential machine includes header files, a driver, and two libraries. The driver and the first library contain only syntax and data types that are ANSI-C standard. Any machine-specific constructs would be encapsulatedin functions found in the second library (see Figure 5.2).

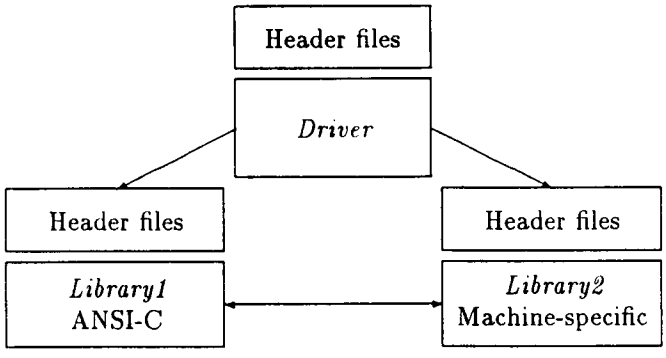


Figure 5.2: Program structure of sequential codes.

When porting the code to another machine, the user changes only the machine-specific modules in *Library2* with appropriate header files. Such an implementation

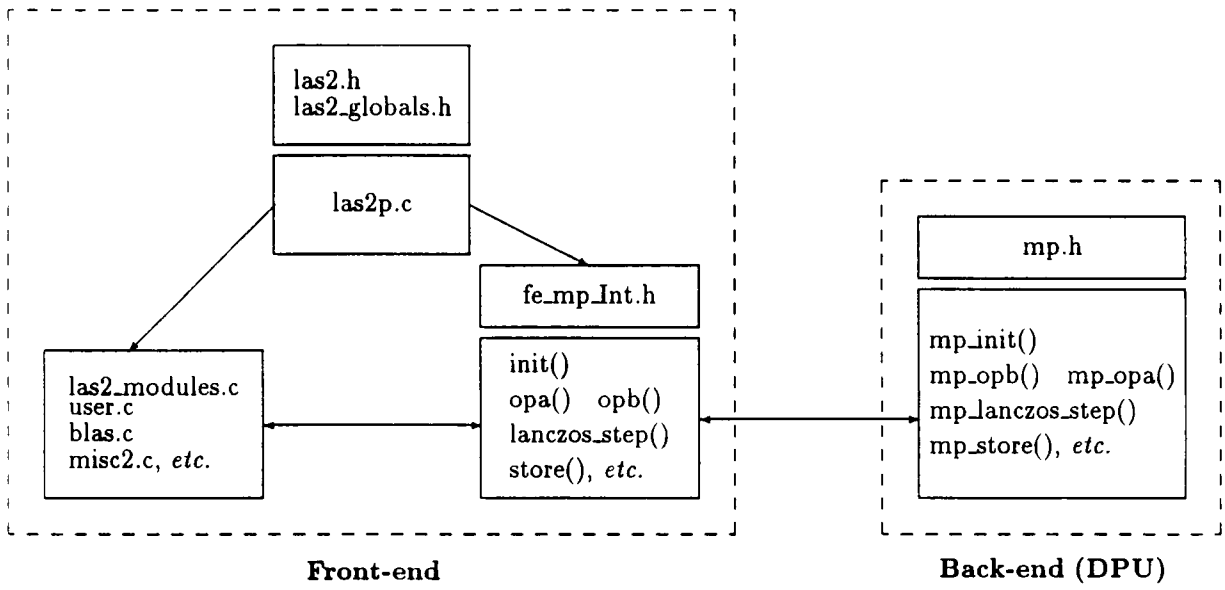


Figure 5.3: Program structure of parallel codes.

consists of the left part of Figure 5.3. The driver in this case would be `LAS2C` (`LAS1C`). Since the single-vector Lanczos algorithm is inherently sequential, we parallelize only matrix-vector multiplication and other vector-based operations. The profile of `LAS2C` running on the DEC 5000-200 in Figure 5.1 shows that 25% of the time is spent doing the matrix-vector multiplication in module `opb()`. (The breakdown of `LAS1C` follows in a similar manner.) Therefore, our first step is to parallelize this operation. In the second phase, we note that `lanczos_step()`, the routine which does the three-term Lanczos recursion in (2.12) and selective local re-orthogonalization of the next Lanczos vector (see Section 2.2) against the two most-recently generated vectors, consists of a series of Basic Linear Algebra Subprograms (BLAS) calls operated on a small number of vectors. Since the BLAS kernels are easily parallelizable, we obtain a parallel `lanczos_step()` by providing parallel versions of the BLAS routines. From

Figure 5.3, the block diagram of `LAS1P` (`LAS2P`) includes two sections. The first section resides on the front-end (FE) of the MasPar MP-1 and constitutes the sequential part of the code. It therefore has the program structure shown in Figure 5.2, with `user.c`, `blas.c`, and `misc.c` as some of the files found in *Library1*. Files in *Library1* are strictly written in ANSI-C, and they remain unchanged when the entire code is ported. On the other hand, `init()`, `opb()`, `opa()`, `lanczos_step()`, and any other modules that contain MasPar-specific constructs, are located in the second FE library (*Library2*). These are the modules that are to be parallelized or are in some way affected by the computing environment or machine architecture (`init()`). The fact that the matrix-vector multiplication and Lanczos recursion are carried out in parallel in the back-end is transparent to the caller program on the FE. We note that `opb()`, `opa()`, and `lanczos_step()` serve only as the interfacing functions to their parallel counterparts which reside in the DPU.

5.2.2 Data Distribution

In parallel computing, one major consideration is the data mapping among the processor elements. A good data distribution scheme will result in efficient communication and balanced workload. In [OgAi92], Ogielski and Aiello discuss a random mapping scheme of an unstructured sparse matrix that produces well-balanced storage in the MasPar MP-1 PE array. Along with published results, they produced a library of primitives specifically for the MasPar MP-1. In order to take advantage of these existing functions which include the matrix-vector ($y = Ax$) and vector-matrix ($y^T = x^T A$) multiplication kernels `Ax()` and `xA()`, respectively, we have adopted their layout schemes for sparse matrix objects and dense vector objects (see [Ogie92]). These user-defined data types are structures declared in `mp.h` of Figure 5.3:

```
smatrix_t      *A;    /* pointer to sparse matrix object */
cvector_t      *x;    /* pointer to column-vector object */
rvector_t      *y;    /* pointer to row-vector object    */
workspace_t    *ws;   /* pointer to workspace            */
```

We note that there is a *workspace* object which contains temporary storage needed by the `Ax()` and `xA()` kernels whose size depends on the number of nonzeros and the larger of the dimensions of the PE array. `smatrix_t` is a structure of type `struct s_matrix_d` that includes

```
struct s_matrix_d {
int ncol, nrow, nz;              /* matrix size                */
int maxl;                        /* max. load per PE           */
plural short *pel;               /* local load in PE           */
plural unsigned short *ci;       /* reduced col. indices       */
plural unsigned short *ri;       /* reduced row indices        */
plural double *a;                /* matrix elements            */
plural short *cp;                /* work buffer                */
plural short *rb;                /* work buffer                */
plural short *cb;                /* work buffer                */
};
```

Singular integers (see Section 3.3) `ncol` and `nrow` contain the column and row dimensions of the input matrix; `nz` stores the number of nonzeros in the matrix and `maxl` stores the number of maximum nonzeros in a PE. `*pel` is a plural pointer to a variable containing the number of nonzeros stored in each PE. `*ci` and `*ri`, also plural pointers, point to the reduced row (ri) and reduced column (ci) of each nonzero

stored on the PE, where

$$\begin{aligned} ri &= row_{actual}/nyproc, \\ ci &= col_{actual}/nxproc. \end{aligned} \tag{5.1}$$

row and *col* are the row and column locations of the nonzero in matrix A. Finally, plural pointer `*a` points to the nonzeros in each PE while the last three plural pointers point to the workspace needed.

The sparse matrix is read into memory by the function `read_s_matrix()`. Essentially, to map an unstructured sparse $m \times n$ matrix A onto a processor array grid of dimensions $\tilde{m} \times \tilde{n}$, the assignment $A(i,j) \longrightarrow PE(i \bmod \tilde{m}, j \bmod \tilde{n})$ is used. Hence, all nonzero elements of a matrix row (column) are assigned to a single row (column) of the PE array, and conversely, each PE stores a submatrix of matrix A even though the size of these submatrices may vary. For each element, its value, reduced row, and reduced column indices given by (5.1) are stored.

An $l \times 1$ dense column vector x can be mapped as a column-vector object (`cvector_t`) or a row-vector object (`rvector_t`) using a multi-layer lexicographic scheme, which is more efficient than mapping the vector onto a single row (column) of the Maspar MP-1 PE array ([Ogie92]). In row-major order (or row-wise lexicographic layout), $PE(i,j)$ is reassigned the single index $\tilde{k} = i \times \tilde{n} + j$, while in column-major order (column-wise lexicographic layout), $PE(i,j)$ is reassigned the single index $\tilde{k} = j \times \tilde{m} + i$. In both cases, vector element $x(k)$ is assigned to a PE using the mapping $x(k) \longrightarrow PE(\tilde{k} = k \bmod \tilde{m}\tilde{n})$. We note that a column-vector object is stored in *row-major* order in the PE array while a row-vector object is stored in the *column-major* order in the PE array. Both `cvector_t` and `rvector_t` objects are structures of the single type `struct d_vector` where

```
 struct d_vector {
int dim, maxl;                  /* max. load per PE          */
plural double *v;               /* matrix elements           */
plural double *temp;            /* matrix elements           */
 };
```

Here, `dim` and `maxl` are singular integers containing the length of the vector and the maximum number of vector elements in a PE (number of PE layers required to store the vector), respectively. `*v` is a plural pointer to parallel memory that stores the vector elements in each PE. `*temp` points to parallel memory which is used as temporary storage locations for the functions `load_cv()` and `unload_cv()` described below.

5.3 Input and Output Files

From Table 4.2, we see that `LAS1P` and `LAS2P` have the same I/O file requirements as `LAS1C` and `LAS2C`. However, while the input file of the sequential codes, `matrix`, is in ASCII text, the parallel codes expect `matrix` to be in binary format. Given the ASCII `matrix` file, the user must produce a corresponding binary file. Using the data provided in the ASCII file as input, a preprocessor ([Ogie92]) must produce an output binary file of which the first three lines should contain the column and row dimensions and the number of nonzeros in the matrix. Also, for each nonzero located at (i, j) in the input matrix, the binary file should contain a record which comprises its PE destination calculated by

$$iproc_{dest} = (i \bmod nyproc \times nxproc) + j \bmod nxproc, \tag{5.2}$$

its reduced row and column indices computed by (5.1), and the value of the nonzero. Similar to the sequential codes, `LAS1P` and `LAS2P` can be modified to directly accept the input matrix in any available format. Simple programs also can be written for the front-end to preprocess the input matrix and produce any desirable format required by the codes.

5.4 Memory Allocation

Table 4.5 shows the memory requirement of the sequential codes `LAS1C` and `LAS2C` in which the terms

$$(n + nnzeros + 1)/2 + nnzeros$$

define the amount of storage needed to store the input matrix A in 64-bit words. In `LAS1P` and `LAS2P`, the workspace memory in the front-end is reduced by this amount since the input matrix is stored strictly in the PE array. The memory needed in the front-end for storing the Lanczos vectors and the singular vectors remains essentially the same with one exception. During a run, there are occasions when a Lanczos vector or singular vector must be transferred between the sequential memory of the front end and the parallel memory space in the back end. Due to the vector mapping scheme described above, the number of PEs participating in a `READ` or `WRITE` is a *multiple* of the number of PEs in a PE row, $nxproc$. For this reason, if it takes j PE rows to store an n-element vector, the contents of up to $(nxproc - 1)$ PEs in the j-th row may be irrelevant but enough sequential memory must be allocated to accommodate the reading and writing of all $(j \times nxproc)$ PEs. For an array of i n-element vectors from which the PE array only *reads*, the sequential memory we must allocate for this array of vectors is not $(i \times n)$ locations but $(i-1) \times n + vpmem$, where $vpmem \geq n$ and must

be large enough to account for the extraneous PEs in the last row of the PE subarray participating in a `READ`. On the other hand, if the PE array is to *write* to the array of vectors one at a time, the memory we must allocate for it in the front-end would be $(i \times vpmem)$. Therefore, the sequential memory allocation for Lanczos vectors and singular vectors in the front-end in `LAS1P` and `LAS2P` may be slightly greater than that in `LAS1C` and `LAS2C`.

5.5 Primitives

System Primitives

In `LAS1P` and `LAS2P`, a number of user-defined and system primitives are used as building blocks in the parallelized modules. Communication system calls `blockIn()` and `blockOut()` are used throughout the codes to copy a block of data between the front-end and the DPU. If a `blockIn()` is issued from the DPU or a `blockOut()` from the front-end, a block of data is copied into a specified subarray of the PE array, each PE receiving a specified number of consecutive locations. If a `blockIn()` is issued from the front-end or a `blockOut()` from the DPU, each PE in a specified subarray of the PE array writes a specified number of its consecutive locations to the front-end memory. The system primitive `copyIn()` (`copyOut()`) copies a specified number of bytes from (to) the front-end to (from) the ACU if the call is issued from the front-end (ACU).

User-Defined Primitives

We use the `blockIn()` or `blockOut()` system call to copy a dense n-element vector x stored in a linear array in the front-end to the PE array. The maximum number of

consecutive elements of x read by a PE, $maxld$, is determined by

$$maxld = \begin{cases} n/nproc & \text{if } n \bmod nproc = 0 \\ (n/nproc) + 1 & \text{otherwise} \end{cases} \tag{5.3}$$

However, the vector must be mapped in a lexicographic order (as previously described) before it can be used. Function `load_cv()` is called to redistribute the vector elements of a column vector object in the row-wise lexicographic order. Figure 5.4 shows a trivial example of how a 32-element vector on the front-end is mapped into a 4×4 PE array. The inverse function of `load_cv()` is `unload_cv()`. Given a column vector

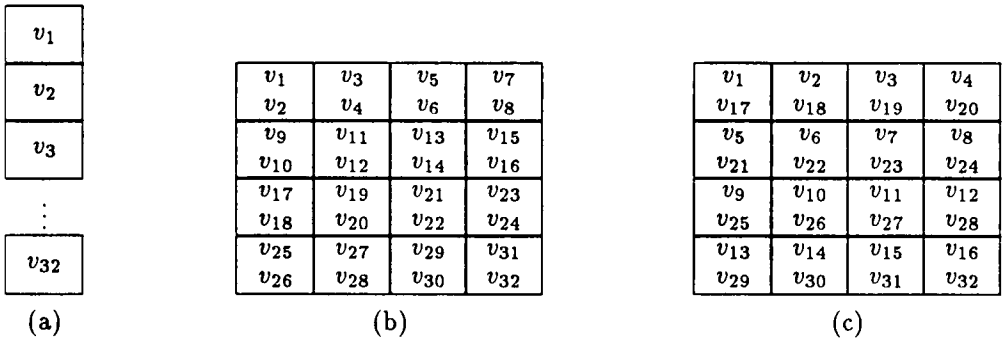


Figure 5.4: Memory mapping of a 32-element vector: (a) in front-end linear array (b) in 4×4 PE array after calling `blockIn()` (c) in 4×4 PE array after calling `load_cv()`

mapped in row-major order as shown in the example in Figure 5.4(c), `unload_cv()` redefines the mapping of the elements of the vector so that each PE contains up to $maxld$ (inclusive) consecutive vector elements (Figure 5.4(b)). We also have function `unload_rv()` which differs from `unload_cv()` only in that its input argument is a row vector object. These two functions are normally followed by a `blockOut()` call to copy the data of the vector object from the PE array to the front-end memory.

`mp_append_cv()` takes as input arguments two column vectors x and y of length n_1 and n_2 and appends the former to the latter. It is assumed that column vector y has sufficient parallel memory to store x. Upon return, the output is the column vector y whose length is increased by the length of x and the new maximum PE load of y is determined by (5.3) where $n = n_1 + n_2$.

The inverse of **`mp_append_cv()`** is the function **`mp_extract_cv()`**, which takes as input arguments a column vector of length n and an index i. It extracts the last $(n - i)$ elements of the input vector and stores them as a column vector of length $(n - i)$.

`mp_dscal()`, **`mp_daxpy()`**, and **`mp_ddot()`** are the MPL versions of LINPACK's BLAS kernels ([DoBS79]) `dscal()`, `daxpy()`, and `ddot()`. `mp_dscal()` takes as input a column vector object and multiplies (in parallel) each element by a constant. The output is stored as a column vector. Similar to `mp_dscal()`, `mp_daxpy()` scales a column vector by a constant and it also adds the scaled elements to corresponding elements of a second column vector. `mp_ddot()` calculates the dot product of two column vectors in parallel.

`Ax()` and `xA()` are two matrix-vector multiplication primitives that compute, respectively,

$$y = Ax \tag{5.4}$$

$$y^T = x^T A \tag{5.5}$$

In both functions, A is the input unstructured sparse matrix whose nonzero layout is described in Section 5.2.2. Again, we note that associated with each nonzero in a PE are three aligned arrays that store the value of the nonzero and its reduced row and column indices. The vector x in (5.4) is stored in row-major order in the PE array.

The product vector y, on the other hand, is stored in column-major order. In (5.5), a vector-matrix multiplication is performed with vector x stored in column-major order in the PE array, while the product vector y^T is stored in row-major order.

We proceed with an outline of the function `Ax()` (Table 5.2) followed by a small example that illustrates how data are laid out and how the matrix-vector multiplication is done in parallel. Without loss of generality, consider (5.4) in which A is a

Table 5.2: Outline of `Ax()`.

1. Systolic distribution of vector components:
   Vector components in column j are copied to all PEs in column j to buffer.
2. Scatter:
   Accumulator locations in column j receive vector components. In accumulator of each PE, vector components from buffer aligning with column indices of nonzeros stored in PE are copied.
3. Multiply:
   Nonzeros are multiplied with corresponding vector components in accumulator.
4. Gather:
   Products in each PE that correspond to nonzeros with same row indices are summed.
5. Systolic row sum collection:
   The sums are accumulated across each PE row.

4×8 matrix and suppose we have a PE grid of size 2×2. Hence, we have

$$A = \begin{bmatrix} 0 & a_2 & 0 & 0 & 0 & a_8 & a_{11} & 0 \\ a_0 & a_3 & a_4 & 0 & a_7 & a_9 & 0 & 0 \\ 0 & 0 & a_5 & 0 & 0 & 0 & 0 & a_{13} \\ a_1 & 0 & 0 & a_6 & 0 & a_{10} & a_{12} & a_{14} \end{bmatrix},$$

$$\text{and } x = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{bmatrix}^T,$$

where a_k are the nonzero elements of A and v_k are the elements of vector x. Figure 5.5(a) shows the layout of A in the PE memory. The matrix object consists of

`a[]` $\equiv$ array storing nonzeros of A

`row[]` $\equiv$ reduced row index of corresponding element in `a[]`

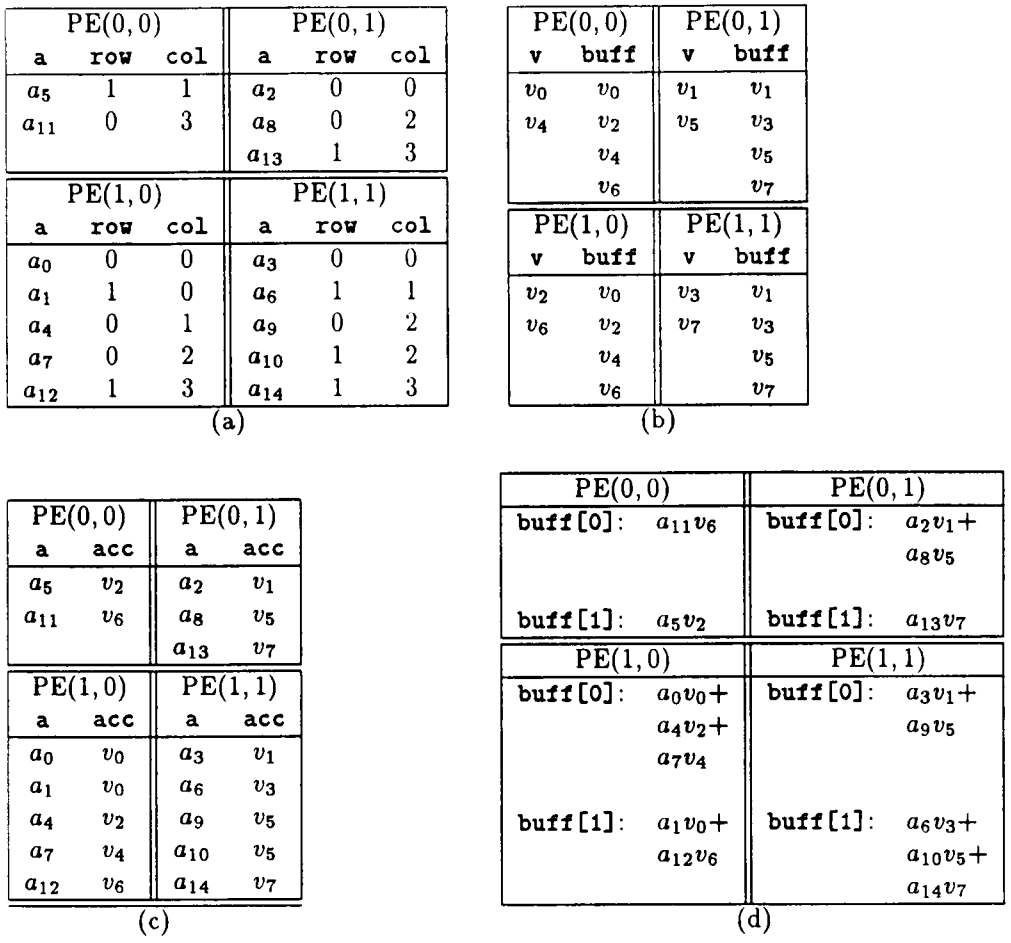
`col[]` $\equiv$ reduced column index of corresponding element in `a[]`.

The array `v[]` in Figure 5.5(b) contains the elements of vector x distributed in row-major order in the PE array, while `buff[]` holds the vector elements after they are distributed in Step 1 of Table 5.2. Figure 5.5(c) shows the contents of `acc[]` after Step 2, and after corresponding elements of `a[]` and `acc[]` are multiplied, the products are stored in `buff[]` as shown in Figure 5.5(d). Row sums are performed in parallel (Step 5 of Table 5.2) and the resulting vector y is stored in column-major order in the PE array.

Finally, we have two other primitives that operate on data objects located locally on the DPU. `pp_opa()` and `pp_opb()` are parallel matrix-vector multiplication routines, and they are described in the next section.

5.6 User-Defined Routines

The set of user-defined functions called by `LAS1C` and `LAS2C`, which were described in the previous chapter, are also called by `LAS1P` and `LAS2P`. However, in the parallel codes, these functions serve mainly as interface modules between the front-end and the parallel domain of the MasPar MP-1 PE array. The main role of these functions is to invoke their parallel counterparts which reside in the DPU (see Figure 5.6). In this section, we will look at `init()`, `mp_init()`, the matrix-vector multiplication



PE(0,0)			PE(0,1)		
a	row	col	a	row	col
a_5	1	1	a_2	0	0
a_{11}	0	3	a_8	0	2
			a_{13}	1	3

PE(1,0)			PE(1,1)		
a	row	col	a	row	col
a_0	0	0	a_3	0	0
a_1	1	0	a_6	1	1
a_4	0	1	a_9	0	2
a_7	0	2	a_{10}	1	2
a_{12}	1	3	a_{14}	1	3

(a)

PE(0,0)		PE(0,1)	
v	buff	v	buff
v_0	v_0	v_1	v_1
v_4	v_2	v_5	v_3
	v_4		v_5
	v_6		v_7

PE(1,0)		PE(1,1)	
v	buff	v	buff
v_2	v_0	v_3	v_1
v_6	v_2	v_7	v_3
	v_4		v_5
	v_6		v_7

(b)

PE(0,0)		PE(0,1)	
a	acc	a	acc
a_5	v_2	a_2	v_1
a_{11}	v_6	a_8	v_5
		a_{13}	v_7

PE(1,0)		PE(1,1)	
a	acc	a	acc
a_0	v_0	a_3	v_1
a_1	v_0	a_6	v_3
a_4	v_2	a_9	v_5
a_7	v_4	a_{10}	v_5
a_{12}	v_6	a_{14}	v_7

(c)

PE(0,0)		PE(0,1)	
buff[0]:	$a_{11}v_6$	buff[0]:	a_2v_1+ a_8v_5
buff[1]:	a_5v_2	buff[1]:	$a_{13}v_7$

PE(1,0)		PE(1,1)	
buff[0]:	a_0v_0+ a_4v_2+ a_7v_4	buff[0]:	a_3v_1+ a_9v_5
buff[1]:	a_1v_0+ $a_{12}v_6$	buff[1]:	a_6v_3+ $a_{10}v_5+$ $a_{14}v_7$

(d)

Figure 5.5: Example demonstrating a parallel matrix-vector multiplication.

routines, `mp_lanczos_step()`, and `mp_s_vector()`. Prologues of some of the functions described in this section are provided in Appendix B.

init() and mp_init()

LAS1P and LAS2P have two routines, `init()` and `mp_init()`, that initialize the front-end and the back-end, respectively. The job of the main initializing function `init()` is to

- invoke `mp_init()`

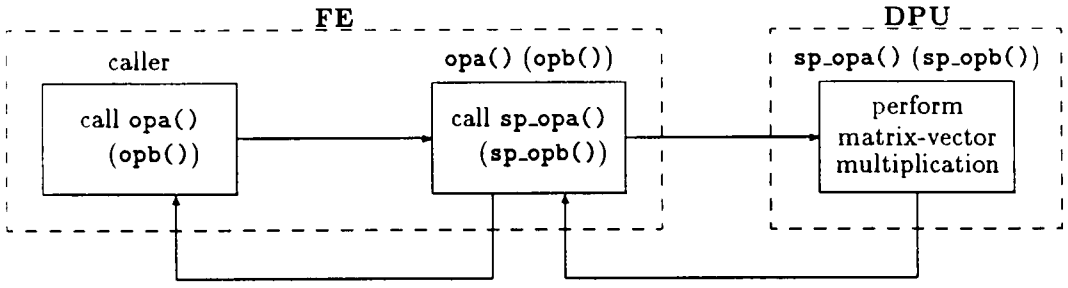


Figure 5.6: Calling sequence of the matrix-vector multiplication routine.

- create and open output files
- allocate necessary memory for the entire run
- write header to output file
- check validity of input parameters

The function `mp_init()` performs the following functions in the DPU:

- read in the sparse matrix
- create workspace necessary in the PE array for the entire run which includes PE memory to hold column and row vector objects
- calculate the needed sequential memory for a vector to accommodate a PE `READ` and/or `WRITE` (*vpmem* as described above)
- return relevant data (matrix dimensions, number of nonzeros, etc.)

Matrix-Vector Multiplication

The parallel counterparts of `opb()` and `opa()` in Table 4.3 are `sp_opb()`, `sp_opa()`, `pp_opb()` and `pp_opa()`(see Table 5.3). `sp_opa()` and `sp_opb()` are identical to

pp_opa() and pp_opb() except for the fact that they operate on data objects already located in the PE array. At the beginning of execution, sp_opa() and sp_opb() copy

Table 5.3: Matrix-vector multiplication kernels for LAS1P and LAS2P.

code	sp_opb() pp_opb()	sp_opa() pp_opa()
LAS1P	$y = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} x$	-
LAS2P	$y = A^T A x$	$y = Ax$

necessary data from the front-end. Then, they perform the appropriate matrix-vector multiplication and copy the resulting product vector into the front-end memory. In Figure 5.6, a request to perform a matrix-vector multiplication in LAS1P and LAS2P involves an extra step. Function opa() (opb()) located on the front-end serves as the interface instead of actually doing the multiplication as in LAS2C (LAS1C). We present an outline of sp_opb() for LAS1P and LAS2P in Table 5.4(a) and (b), respectively. From Table 5.4(b), we see that pp_opb of LAS2P comprises only Steps 3 and 4 since no data transfer is necessary between the front-end and the DPU. An outline of pp_opb of LAS1P is presented in Table 5.5. Lastly, we note that these matrix-vector multiplication modules also provide for the case in which the transpose of the input matrix A, instead of A, is stored in the PE array.

mp_lanczos_step() and mp_s_vector()

mp_lanczos_step() performs one iteration of the Lanczos recursion. At the start of the routine execution, four Lanczos vectors needed to compute the next Lanczos step

Table 5.4: Matrix-vector multiplication kernels for (a) **LAS1P** and (b) **LAS2P**.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

1. Copy x_1 and x_2 from front end
   `blockIn()`
2. Distribute x_1 and x_2 in lexicographic order
   `load_cv()`
3. Compute $y_1 = Ax_2$ and $y_2^T = x_1^T A$
   `Ax()` and `xA()`
4. Redistribute y_1 and y_2 for unload to front end
   `unload_cv()`
5. Copy y_1 and y_2 to front end to appropriate location in linear array storing x
   `blockOut()`

(a)

$$y = A^T A x$$

1. Copy vector x from front end
   `blockIn()`
2. Distribute x in lexicographic order
   `load_cv()`
3. Compute $z = Ax$
   `Ax()`
4. Compute $y^T = z^T A$
   `xA()`
5. Redistribute y for unload to front end
   `unload_cv()`
6. Copy vector y to front end
   `blockOut()`

(b)

Table 5.5: Matrix-vector multiplication kernel using local data for **LAS1P**.

$$y = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} x,\ y = (y_1^T, y_2^T)^T,\ x = (x_1^T, x_2^T)^T.$$

1. $z \longleftarrow x_2$
   mp_extract()
2. Compute $y_1 = Az$
   Ax()
3. Compute $y_2^T = x_1^T A$
   xA()
4. $y = [y_1^T, y_2^T]^T$
   mp_append()

are copied to and stored as column vector objects in the PE array. A series of BLAS calls are made to determine the next Lanczos vector, α_j, and β_j in (2.12). Since all the data needed are located in the DPU, `mp_dscal()`, `mp_daxpy()`, `mp_ddot()`, `pp_opa` and `pp_opb` can be used. At the end of the recursion, the updated vectors are copied back to the front-end memory.

`mp_s_vector()` is invoked to calculate the left singular vectors corresponding to the singular values already computed in **LAS2P**. It is called only when the user requests the computation of singular vectors and when the singular vectors are stored strictly out-of-core (determined by an input parameter). If the user specifies that the singular vectors are to be stored in-core also, then the left singular vectors are computed on the front-end. At this point in the code, the right singular vectors are already computed and the left singular vectors can be obtained using (2.9). For each left singular vector to be computed, `mp_s_vector()` issues a parallel **READ** to the PEs to obtain the corresponding right singular vector from the binary file. The left singular vectors are computed and the PEs write the vector elements to another binary file.

Chapter 6

Results

6.1 Overview

In this chapter, we discuss the performance of our single-vector Lanczos codes within the sequential and parallel computing environments. For a more thorough discussion of comparisons between the 2-cyclic and the A^TA-based implementations of the Lanczos algorithm, we refer to [Berr92a]. Our test suite consists of nine unstructured sparse matrices (listed in Table 6.1), which primarily arise from information retrieval applications (see Section 1.1).

As noted in Section 5.1, there are two variations of the parallel codes, `LAS1P` and `LAS2P`. We distinguish the variations (see Table 6.2) by the suffixes "(MV)" and "(LS)". The former refers to the use of parallelism for matrix-vector multiplication exclusively. The latter refers to the use of parallelism for matrix-vector multiplication *and* each iteration of (2.12) or Lanczos step.

All computations were performed using double (64-bit) precision and all timings reflect elapsed wallclock times. In each case, we made every effort to ensure the

Table 6.1: Matrices used in test suite.

Name	Source	Dimensions	number of nonzeros	Density (%)	μ_r	μ_c
ILLC	Harwell/Boeing Collection	1850 × 713	10608	0.80	5.7	14.9
WELL	Harwell/Boeing Collection	1850 × 713	10608	0.80	18.2	34.9
LATERAL2	Apple Computer	2695 × 747	47115	2.34	17.5	63.1
LATERAL5	Apple Computer	1614 × 747	43061	3.57	26.7	57.7
MED	Bellcore	5831 × 1033	52012	0.86	8.9	50.4
CRAN	Bellcore	4997 × 1400	78942	1.13	15.8	56.4
CISI	Bellcore	5143 × 1460	66340	0.88	12.9	45.4
TECH	Bellcore	16637 × 6535	327244	0.30	20.0	50.0
CUPENC	Columbia University Press	29670 × 15460	539098	0.12	18.2	34.8

μ_r = average number of nonzeros per row
μ_c = average number of nonzeros per column

Table 6.2: Machines and computing environments used.

Program name	Environment	Machine
LAS1P(MV) LAS2P(MV)	parallel: matrix-vector mult. computed in DPU	MasPar MP-1
LAS1P(LS) LAS2P(LS)	parallel: matrix-vector mult., Lanczos step, residual norms computed in DPU	MasPar MP-1
LAS1C LAS2C	sequential	IBM RS/6000-550 Sparc 2 DEC 5000-200 Macintosh IIfx

execution was done in a dedicated (single user) computing environment. The singular value approximations obtained from identical runs across all five machines are equivalent. For example, we used the IBM RS/6000-550 to approximate 300 singular triplets of matrix MED via the A^TA eigensystem, and the largest singular value of the 328 actually computed only varied in the 14^{th} decimal digit across all the machines. The residual norms (2.11) for this particular run are plotted in Figure 6.1. Scatter plots of the 300-largest singular values for each sparse matrix in our test suite as approximated by LAS2C (using eigensystems of A^TA) are provided in Appendix A.

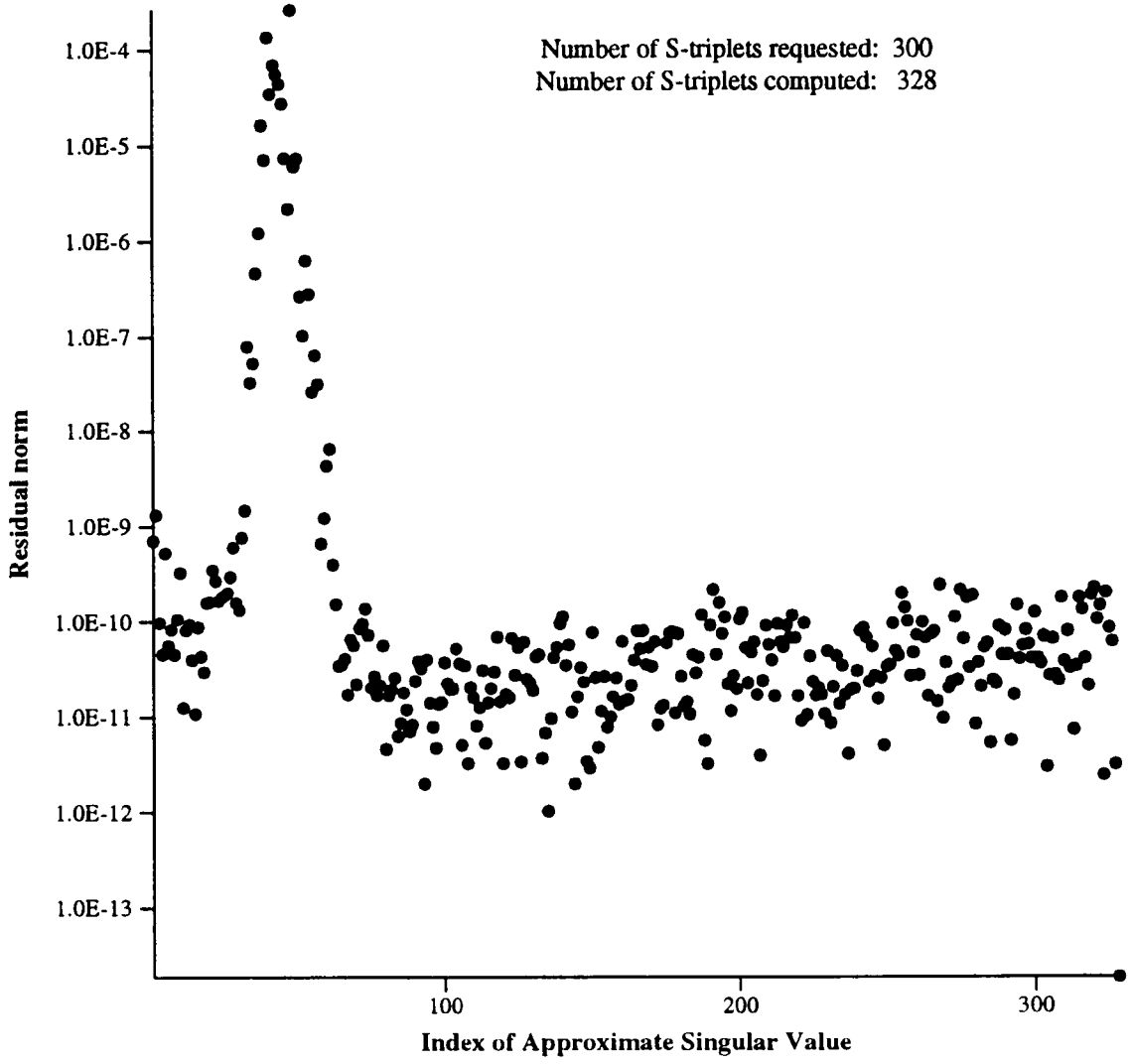


Figure 6.1: Residual norms obtained from approximated eigenpairs of $A^T A$ of matrix MED. Singular value approximations are ordered by increasing unsigned magnitude.

In Figure 6.2, the residual norms obtained from approximated eigenpairs of the 2-cyclic system (2.6) of matrix MED are also plotted. In this run, 100 singular triplets were requested and 128 were actually computed. One characteristic of the Lanczos algorithm is that the extremal eigenvalues of B will converge first. As the number of iterations increases, interior eigenvalues are discovered while approximations of outer eigenvalues of B improve ([PaSc79]). Since eigenvalues of the 2-cyclic system occur in pairs (same magnitude but opposite in signs), we see from Figure 6.2 that the residual norm plot is somewhat symmetric with respect to the spectrum shown. The eigenvalues on both ends of this interval are well-approximated while the accuracies significantly drop for the approximations of the eigenvalues toward the center of the interval. The same behavior of accuracy can be seen in Figure 6.1, where B is the A^TA eigensystem. Again, the extremal eigenvalues of B are approximated with greater accuracy while the approximations of the innermost eigenvalues tend to be much less accurate.

6.2 Comparison between Variations of LAS1P and LAS2P

As previously noted, there are two variations in the parallel codes, LAS1P and LAS2P. From Table 6.2, we see that LAS1P(MV) and LAS2P(MV) only compute the matrix-vector multiplication in the DPU. LAS1P(LS) and LAS2P(LS), on the other hand, compute at least one other routine (`lanczos_step()`) in the DPU. With an increase in parallelism, one would expect to see a speed improvement in the performance of LAS1P(LS) and LAS2P(LS) with respect to LAS1P(MV) and LAS2P(MV). However, Figures 6.3 and 6.4 indicate that this is not the case.

Figure 6.5 demonstrates the DPU profiles of LAS2P(MV) and LAS2P(LS) when

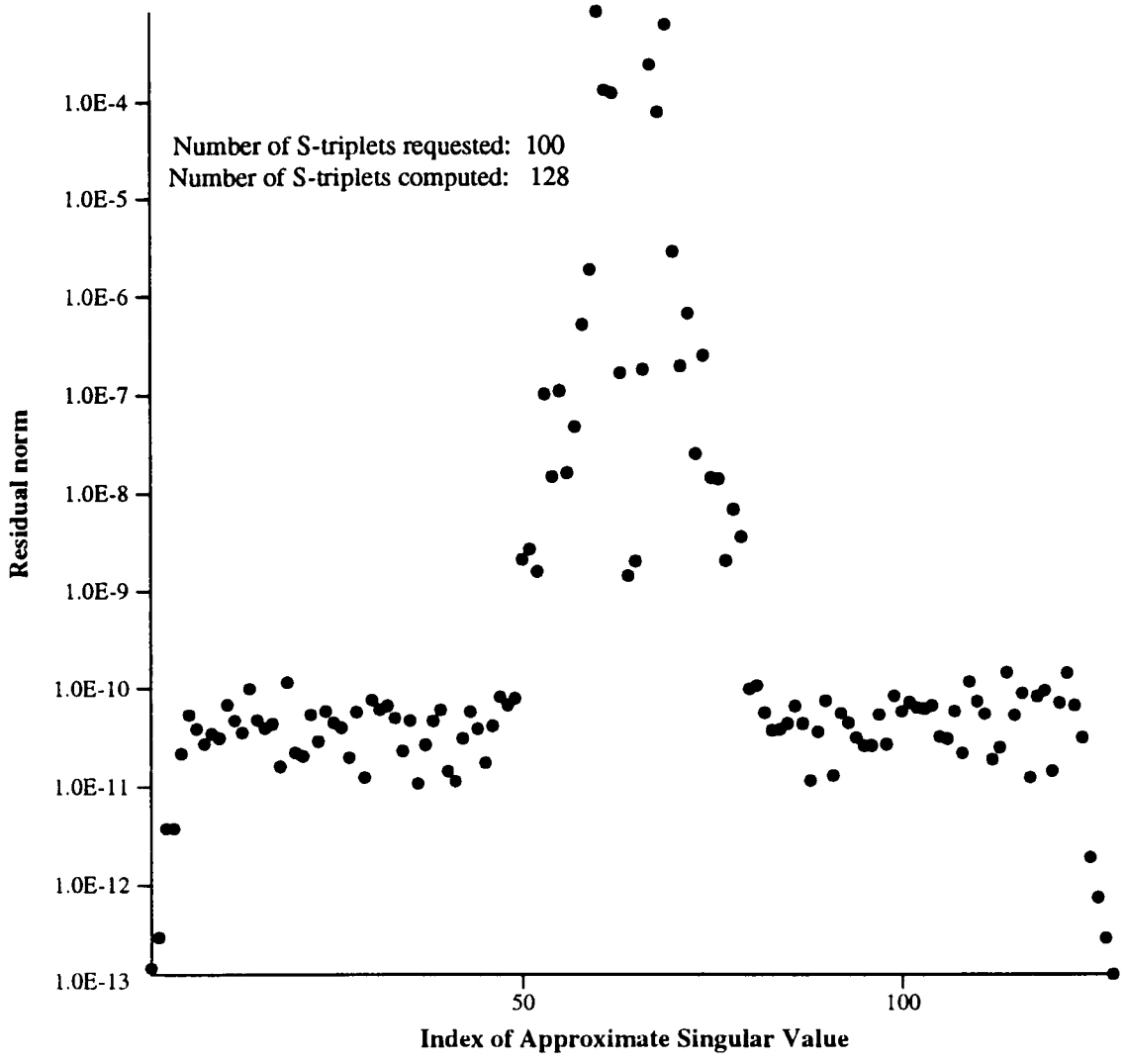


Figure 6.2: Residual norms obtained from eigenpairs of the 2-cyclic approximation of matrix MED. Singular value approximations are ordered by increasing signed magnitude.

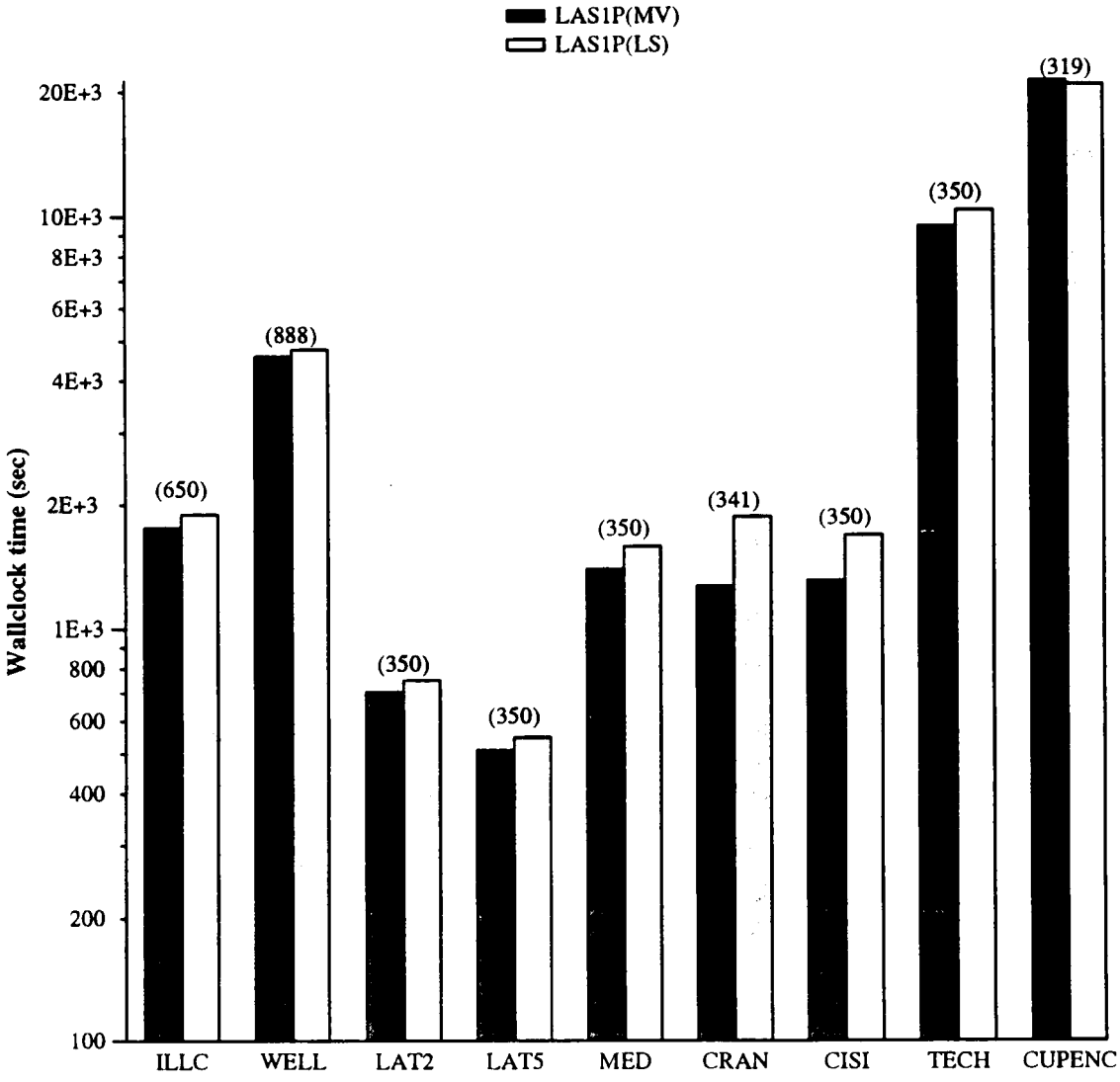


Figure 6.3: Run times of the two variations of **LAS1P** for the test suite. The value in parentheses denotes the number of iterations executed for the corresponding dataset.

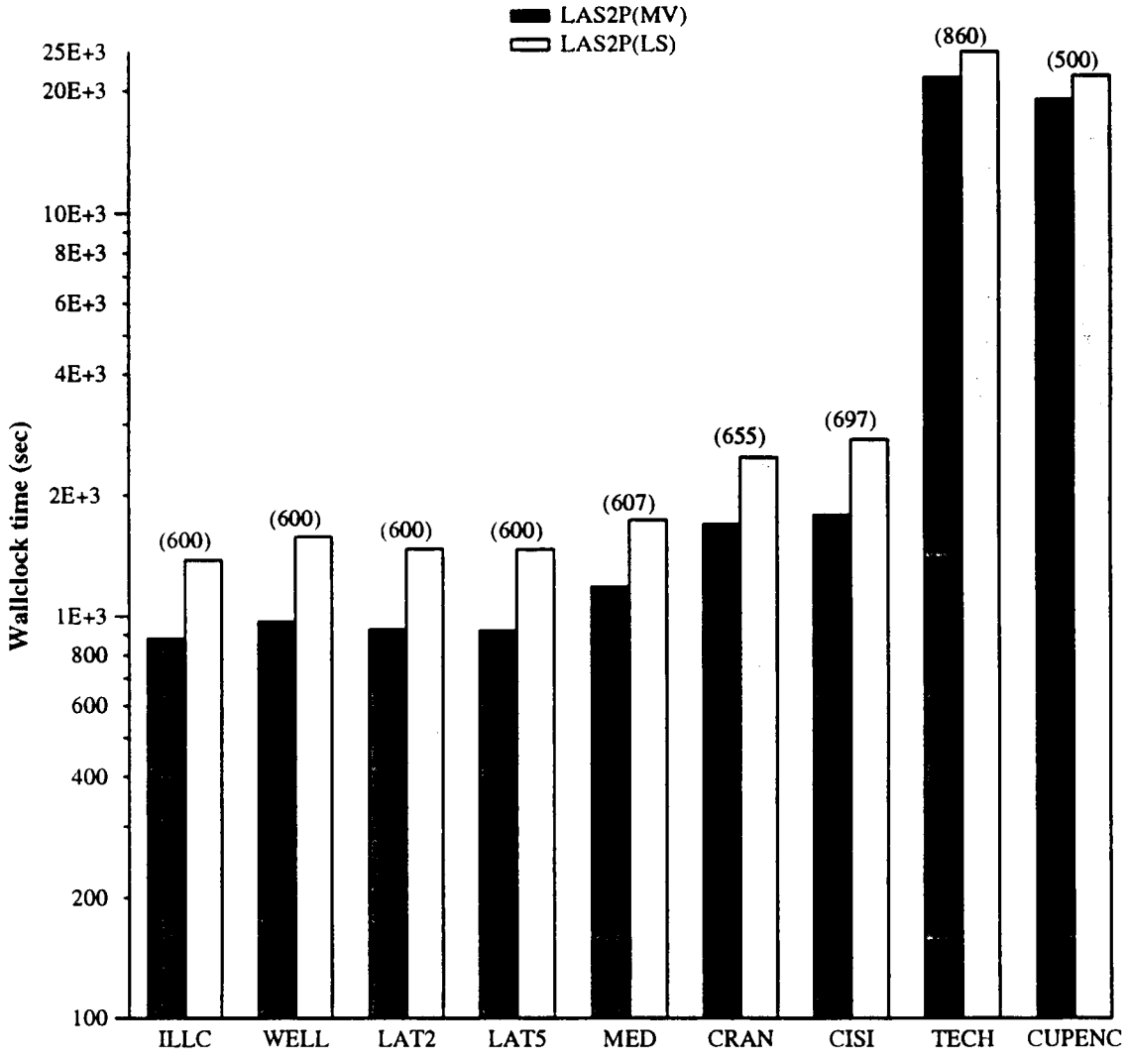


Figure 6.4: Run times of the two variations of **LAS2P** for the test suite. The value in parentheses denotes the number of iterations executed for the corresponding dataset.

Ticks	%	Histogram	Routine
18274	38		`Ax`
10830	22		`xA`
7142	15		`unload_rv`
5982	12		`load_cv`
4412	9		`unload_cv`
1148	2		`sp_opb`
875	2		*others*

(a)

Ticks	%	Histogram	Routine
8216	25		`unload_cv`
7978	25		`Ax`
7592	23		`load_cv`
4818	15		`xA`
1656	5		`mp_lanczos_step`
2045	7		*others*

(b)

Figure 6.5: DPU Profiles of (a) LAS2P(MV) and (b) LAS2P(LS) when computing 132-largest singular triplets (300 iterations) of matrix MED.

computing 132 of the largest singular triplets (300 iterations) of matrix MED . In the first case, we see that 60% of the DPU time is spent in the matrix-vector multiplication kernels and 36% of the DPU time is spent copying arrays of vector elements between the FE and the DPU via the `unload_rv`, `unload_cv` and `load_cv` modules.

We recall that in LAS2P(LS), the Lanczos recursion is performed in the DPU. In Figure 6.5(b), the histogram shows that 48% of the DPU time is attributed to FE-DPU data transfer. Finally, to demonstrate the effect of this bottleneck in elapsed wallclock time we compute, for each variation, the average time spent in the Lanczos recursion and the average cost in data transfer from the start to the end of the recursion (see Table 6.3). We also include timings for LAS2C on the front-end (DEC 5000-200) using the same input parameters and matrix for comparison.

We see from Table 6.3 that without considering the bottleneck due to data copying,

Table 6.3: Average cost in seconds of the Lanczos recursion and data transfer in one iteration of the A^TA-based eigenpair approximation.

Code	Machine	Parallel modules performed in DPU	Average time (sec)	
			Lanczos recursion	Data transfer
LAS2P(MV)	MasPar MP-1	`sp_opb()`	.0834	–
LAS2P(LS)	MasPar MP-1	`mp_lanczos_step()`	.0546	.127
LAS2C	DEC 5000-200	*none*	.183	–

the average time to compute the Lanczos recursion in LAS2P(LS) is 35% and 70% faster than that of LAS2P(MV) and LAS2C, respectively.

6.3 Comparison between parallel and sequential executions

In this section, we look at the total execution time of the 2-cyclic and the A^TA-based implementations obtained from each machine for our test suite. In some cases, a particular run could not be executed due to an insufficient amount of physical memory and/or disk storage. Table 6.4 lists the input parameters used, number of iterations performed, and number of triplets accepted (with $kappa = 1.0^{-6}$) for each dataset. These values are consistent across all the executions performed. In Figures 6.6 and 6.7, execution times of four sequential machines for the 2-cyclic and the A^TA-based eigensystems, respectively, are plotted for comparison. Clearly, the runs of LAS1C and LAS2C on the IBM RS/6000-550 have the best timings. Although the timings on the Macintosh IIfx may not be comparable to those obtained on the other machines, the fact that a personal computer can be used to solve large problems makes it a viable alternative for modest computing environments.

In Figures 6.8 and 6.9, we compare the parallel runs of LAS1P(MV) and LAS2P(MV) of each dataset to corresponding sequential runs made on the IBM RS/6000-550.

Table 6.4: Parameters of the (a) 2-cyclic and the (b) A^TA-based implementations.

2-cyclic eigensystem			
Dataset	Number of iterations	Number of S-triplets accepted	Storage of Lanczos Vectors
ILLC1850	650	123	in-core
WELL1850	888	248	in-core
LATERAL2	350	101	in-core
LATERAL5	350	106	in-core
MED	350	128	in-core
CRAN	341	124	in-core
CISI	350	112	in-core
TECH	350	108	out-of-core
CUPENC	319	122	out-of-core

$kappa = 1.0^{-6}$
Number of singular triplets requested = 100
Singular vectors stored out-of-core

(a)

A^TA-based eigensystem			
Dataset	Number of iterations	Number of triplets accepted	Storage of Lanczos Vectors
ILLC1850	600	429	in-core
WELL1850	600	600	in-core
LATERAL2	600	363	in-core
LATERAL5	600	359	in-core
MED	607	328	in-core
CRAN	655	328	in-core
CISI	697	332	in-core
TECH	860	330	out-of-core
CUPENC	500	200	out-of-core

$kappa = 1.0^{-6}$
Number of singular triplets requested = 300
Singular vectors stored out-of-core

(b)

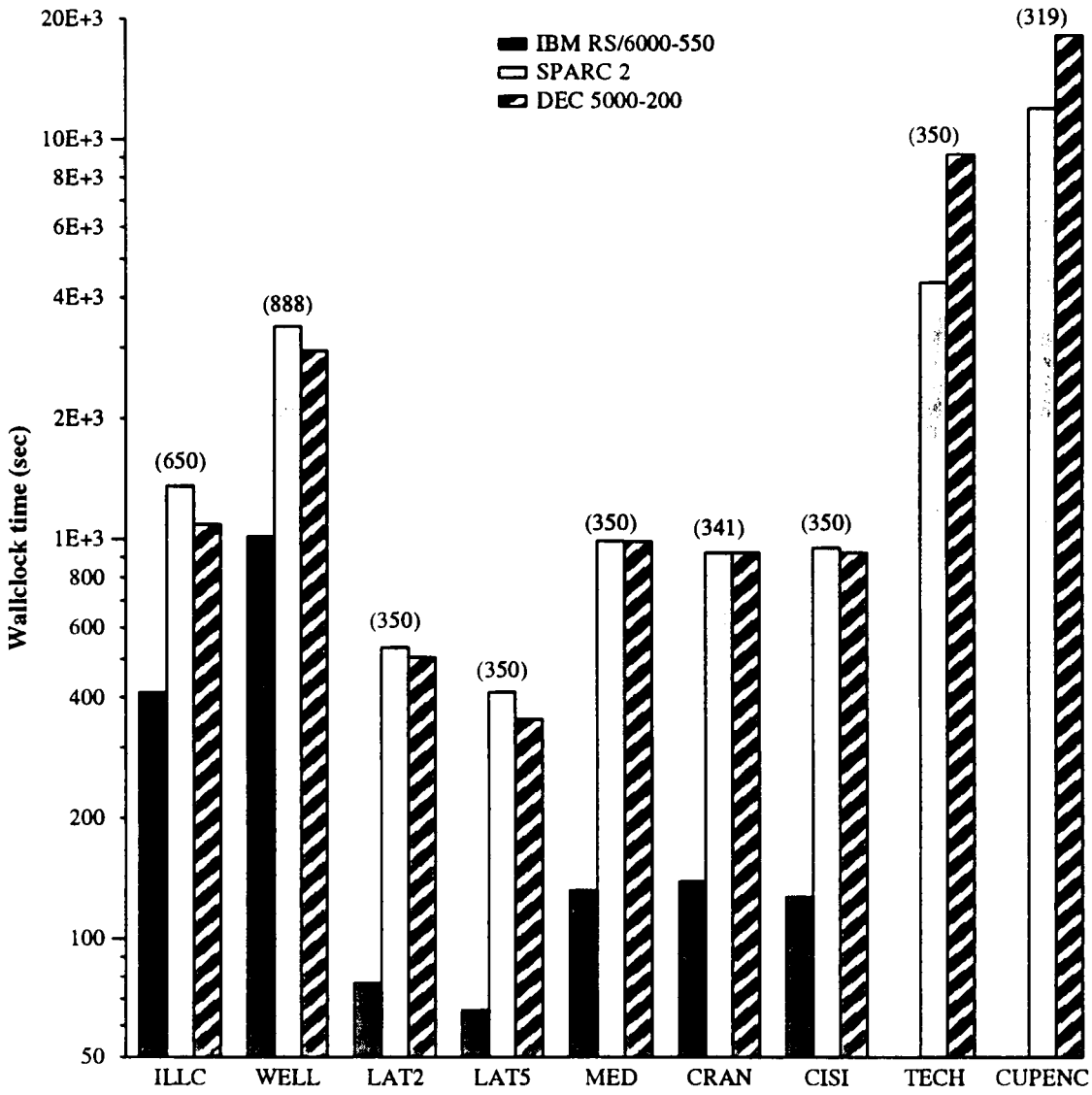


Figure 6.6: Run times of `LAS1C` on 3 sequential machines. The value in parentheses denotes the number of iterations executed for the corresponding dataset.

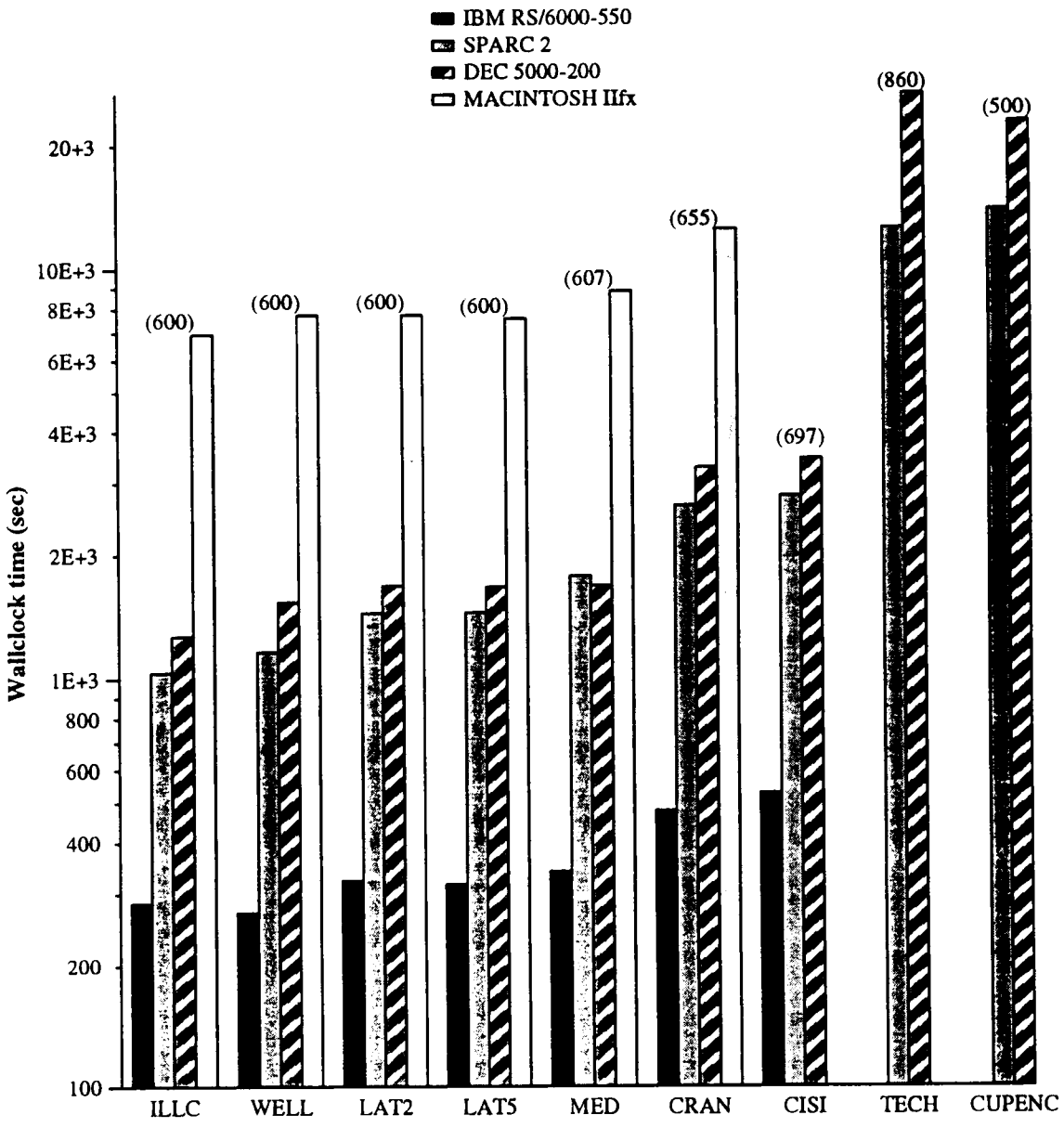


Figure 6.7: Run times of LAS2C on 4 sequential machines. The value in parentheses denotes the number of iterations executed for the corresponding dataset.

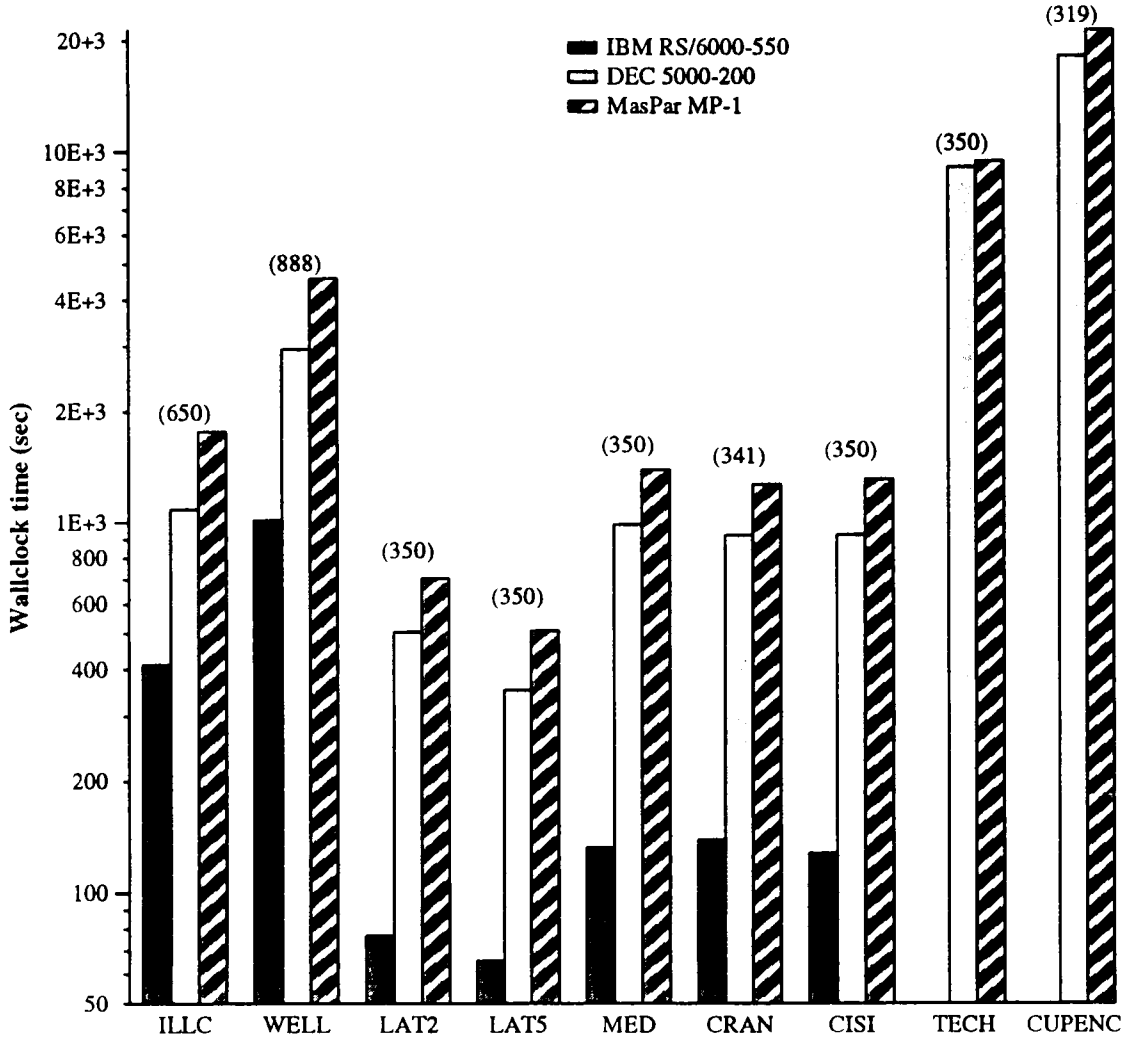


Figure 6.8: Run times of `LAS1C` and `LAS1P`(MV). The value in parentheses denotes the number of iterations executed for the corresponding dataset.

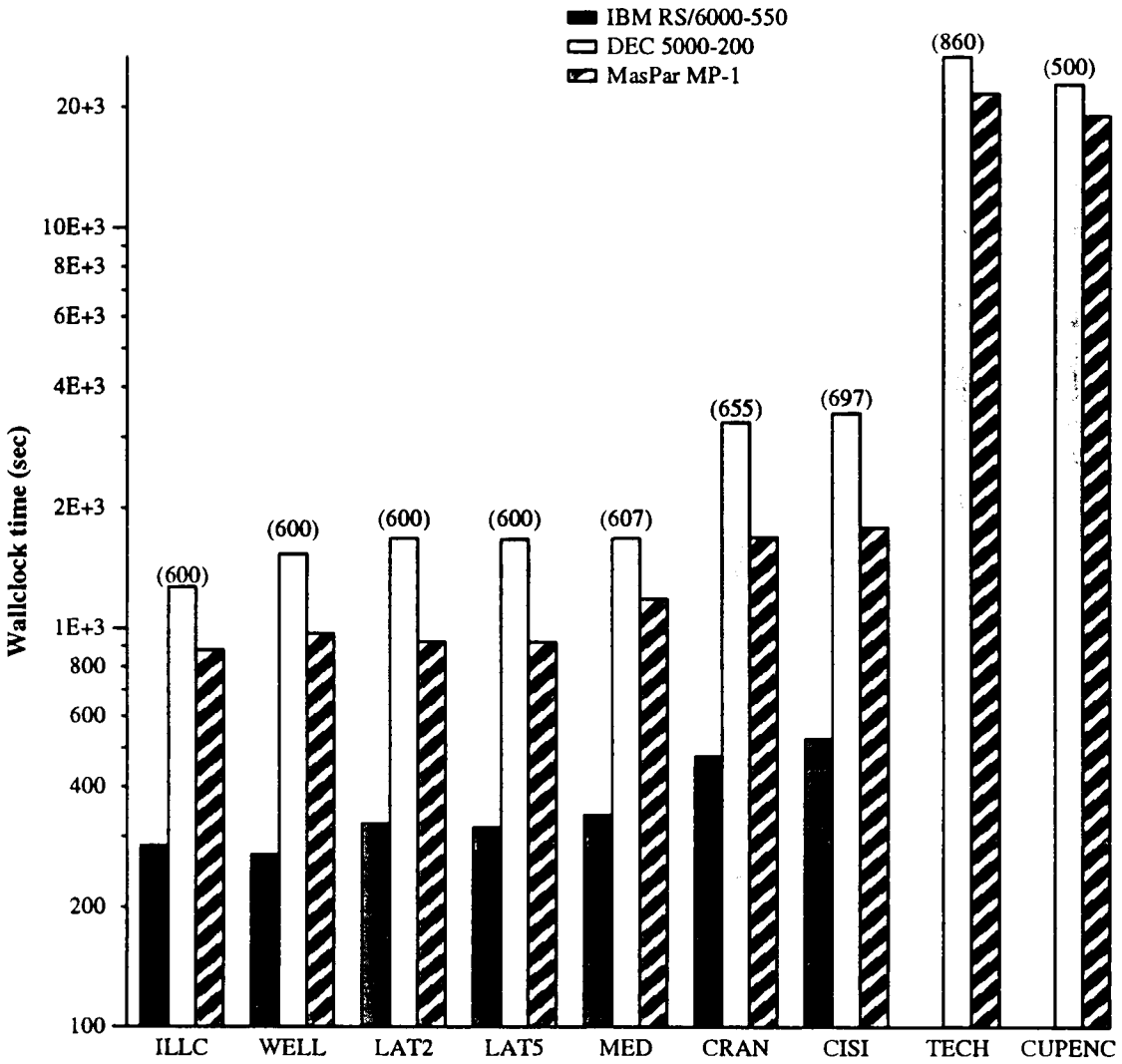


Figure 6.9: Run times of `LAS2C` and `LAS2P`(MV). The value in parentheses denotes the number of iterations executed for the corresponding dataset.

Where IBM RS/6000-550 timings are not available, we provide timings on the DEC 5000-200, i.e. , the MasPar MP-1 front-end. From Figures 6.8 and 6.9, we see that the sequential runs of `LAS1C` and `LAS2C` on the IBM RS/6000-550 yield the lowest execution time among runs from all the machines considered. Whereas `LAS2P(MV)` is on average 1.6 times faster than `LAS2C` on the front-end (see Figure 6.9), this is not the case with the 2-cyclic implementation. In fact, from Figure 6.8, we see that the sequential runs of `LAS1C` on the DEC 5000-200 produce better timings (smaller elapsed wallclock time) than the corresponding parallel runs. We note that the order of the problem in the 2-cyclic implementation is $N = m + n$, while it is $N = n$ for the A^TA-based implementation. The effect of the FE-DPU communication overhead (discussed in previous section) is much more prominent with the former since vectors of length $m + n$, instead of n, are copied in and out of the DPU.

6.4 Selecting Input Parameters

In order to use the single-vector Lanczos codes to their maximum potential, the user must understand the behavior of the algorithm and how it is affected by various input parameters. In this section, we describe some simple strategies to guide the user in selecting appropriate parameters, avoiding the trial-and-error approach in obtaining the desired output. For applications such as information retrieval and seismic reflection tomography ([Berr92a]) which do not necessarily require that the SVD be computed in real time, the availability of sufficient memory is generally the main constraint in any computing environment. This constraint places a limit on the number of possible iterations, and therefore indirectly affects the the number of triplets obtained. Even though the number of triplets desired is normally much

smaller than the order of the problem, memory limitation still poses a challenge when the problem size is very large.

In Section 4.2, we described in detail the parameter input file required by the single-vector Lanczos codes. The *lanmax* and *maxprs* parameters are the maximum number of iterations allowed and the number of triplets of the input matrix desired, respectively. As a stopping criterion, *lanmax* must be small enough so that the memory constraint is satisfied, yet sufficiently large to obtain at least *maxprs* triplets. It is important to note that the ratio between the number of iterations performed and the number of triplets obtained depends on the spectrum of the input matrix, and is thus problem-dependent. For the single-vector algorithm, if the singular values of a matrix are tightly clustered, substantially more iterations are typically needed to obtain the same number of triplets than in the case where the singular values are well-separated ([Berr92b]). *maxprs* is another stopping criterion for these methods. Even though the number of triplets desired is *maxprs*, slightly more than *maxprs* triplets may have converged at program termination. During execution, the Krylov subspace associated with the Lanczos recursion (2.12) may become large enough so that there are more Ritz approximations within the specified tolerance than what was originally desired. In general, when the user does not have any knowledge about the spectrum of a given input matrix A, he can estimate the number of iterations needed for approximating the largest (smallest) *maxprs* triplets of A by

$$3 \times maxprs \leq lanmax \leq N, \tag{6.1}$$

where $N = m + n$ for the cyclic-based eigensystem (`LAS1C`), and $N = n$ for the A^TA eigensystem (`LAS2C`).

Another parameter in the input file for optimal use is *memory*. It allows the user to specify where the Lanczos and the singular vectors are to be stored during execution – in physical memory or secondary I/O medium. Obviously, it is best to store all data in-core to avoid the cost of high latencies associated with secondary storage I/O. Given the memory constraint of the computing environment and the size of the problem, the user can use Table 4.5 to calculate the memory needed or use the curves in Figure 6.10 to get a rough idea of how the memory should be allocated with the given problem. Applicable to both `LAS1C` and `LAS2C`, Figure 6.10 illustrates the memory requirement for the cases in which

- both the Lanczos and singular vectors are stored in-core, and

$$memory_{required} \approx N(j+2) + N(j+1) + j^2 + 9N + 4j; \tag{6.2}$$

- the singular vectors are stored out-of-core, and

$$memory_{required} \approx N(j+2) + 2N + j^2 + 9N + 4j; \tag{6.3}$$

- the Lanczos vectors are stored out-of-core, and

$$memory_{required} \approx N(j+1) + j^2 + 9N + 4j; \tag{6.4}$$

- both the Lanczos and singular vectors are stored out-of-core, and

$$memory_{required} \approx 2N + j^2 + 9N + 4j, \tag{6.5}$$

in the order of preference. The approximated memory required given by (6.2), (6.3), (6.4), and (6.5) excludes storage of the sparse matrix and assumes that j, the number of iterations taken, is equal to the maximum number of iterations allowed $lanmax$. As an example, assume that the physical memory available in a computing environment is

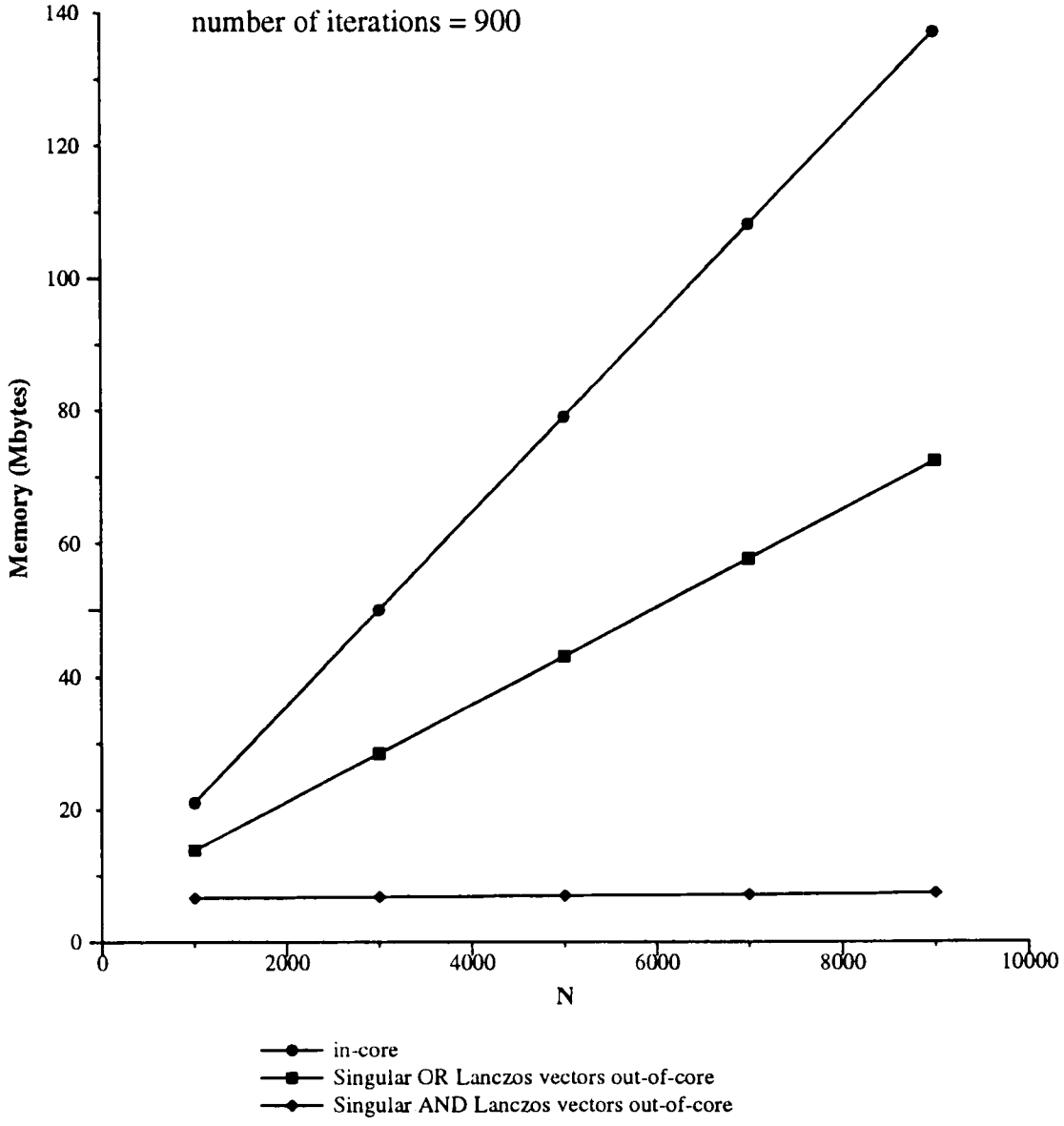


Figure 6.10: Memory requirement of `LAS1C` and `LAS2C` for different problem sizes and different memory configurations.

50 Mbytes and that the dimensions of the input matrix A is 2000×2000. The order of the matrix B is 2000 and 4000 for the A^TA-based and cyclic-based implementations, respectively. From Figure 6.10, we see that in order to use `LAS1C` to compute the SVD of A with 900 iterations, at least 60 Mbytes of physical memory must be available. To reduce this amount to less than 50 Mbytes, the singular vectors must be stored out-of-core. Using the `LAS2C` code, the memory required is roughly 35 Mbytes. Therefore, this problem can be solved in-core using `LAS2C` on a machine whose memory is limited to 50 Mbytes. In most cases, the secondary I/O associated with the storage of the Lanczos and/or singular vectors should be performed on a local disk. However, in a multi-user environment, solving very large problems with both the Lanczos and the singular vectors stored out-of-core may not be possible since disk space may be scarce. The user must then reduce the number of iterations that the code is required to perform.

Chapter 7

Conclusion

We have produced a portable set of C codes that compute the SVD of large unstructured sparse matrices. By the merit of the C language, these codes allow the user to exploit the available memory in a given computing environment. The portability of `LAS1C` and `LAS2C` also allows for better integration of sophisticated systems that consist of many smaller components. This is especially true with the LSI application in which there are many pre-processing and post-processing stages (involving C programs and Unix Shell scripts), in addition to the SVD computation. One drawback with the software is that it requires the user to have some basic understanding of the behavior of the Lanczos algorithm in order to select appropriate input parameters for optimal performance. Even though this takes away the "black box" characteristic desired in sophisticated software, the codes offer flexibility that facilitates their integration in different applications. From the performance results collected, we have produced a set of benchmarks that may provide useful information for future applications. For extremely large matrices A in which the SVD cannot be computed on the existing computing environment, our benchmarks can aid the user in partitioning

the original SVD computation into a sequence of smaller computations involving a manageable number of singular triplets.

The parallel versions of `LAS1C` and `LAS2C` have demonstrated that the single-vector Lanczos procedure is not necessarily a viable algorithm for data-parallelization on SIMD architecture such as the MasPar MP-1. Using the matrix-vector multiplication primitives and the data layout scheme from [Ogie92], we have modest improvement in performance in the A^TA case. We believe, however, significant improvement can be achieved with better data distribution schemes. Also from this work, we offer a simple and systematic approach for writing software in a parallel environment that makes porting across architectures easier.

For future work, we suggest

- exploring the software engineering aspects of the object-oriented method used to facilitate code porting, especially among different architectures in a parallel environment,
- developing effective ways to distribute data across the PE array on the MasPar MP-1 or other SIMD architectures, and
- implementing alternative re-orthogonalization strategies that might improve execution time of the SVD codes while maintaining the same level of accuracy.

Bibliography

Bibliography

[Berr92a] M. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, Spring 1992.

[Berr92b] *Computing the sparse singular value decomposition via SVDPACK*, Minneapolis, February 1992. Proceedings of the IMA Workshop on Iterative Methods for Sparse and Structured Problems.

[CuWi85] J. K. Cullum and R. A. Willoughby. *Lanczso Algorithm for Large Symmetric Eigenvalue Computations*, volume 1 Theory. Birkhäuser, Boston, 1985.

[DoBS79] J. Dongarra, J. Bunch, and G. W. Stewart. *LINPACK Users' Guide.* SIAM, Philadelphia, 1979.

[DDF+90] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, September 1990.

[DuGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[GoVa89] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins, Baltimore, 1989. Second Edition.

[MasP92] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Programming Manuals*, July 1992.

[OgAi92] A. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. Technical report, Bellcore Technical Report, May 1992. Submitted to SIAM J. Sci. Stat. Comp., 1991.

[Ogie92] A. Ogielski. Sparse matrix computations on the MasPar. 1992.

[Parl80] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[PaSc79] B. N. Parlett and D. S. Scott. The Lanczos algorithm with selective orthogonalization. *Mathematics of Computation*, 33(145):217–238, January 1979.

[Varg62] R. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1962.

Appendices

Appendix A

Spectra of Matrices in Test Suite

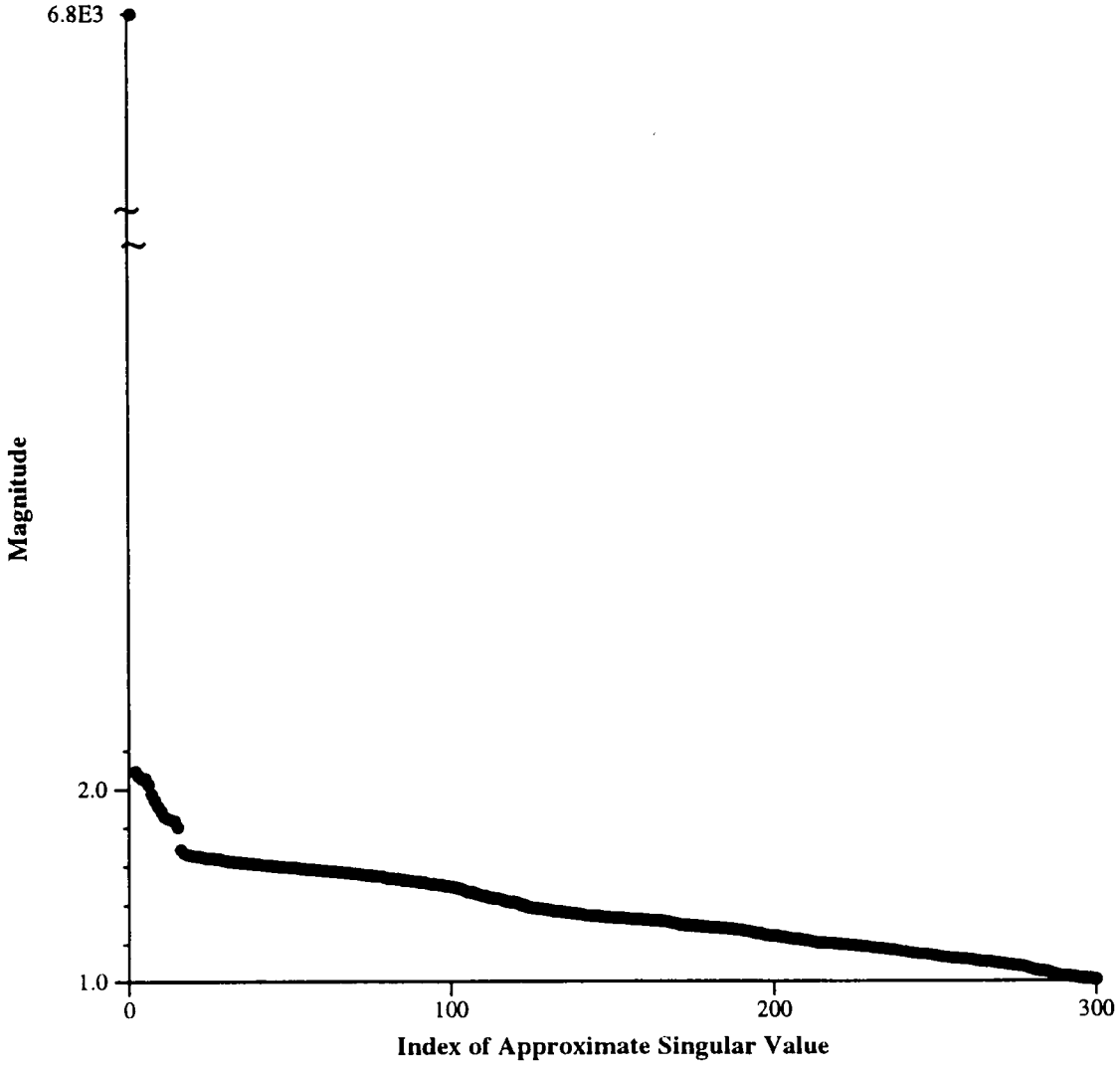


Figure A.1: The 300-largest singular values of dataset ILLC in Table 6.1.

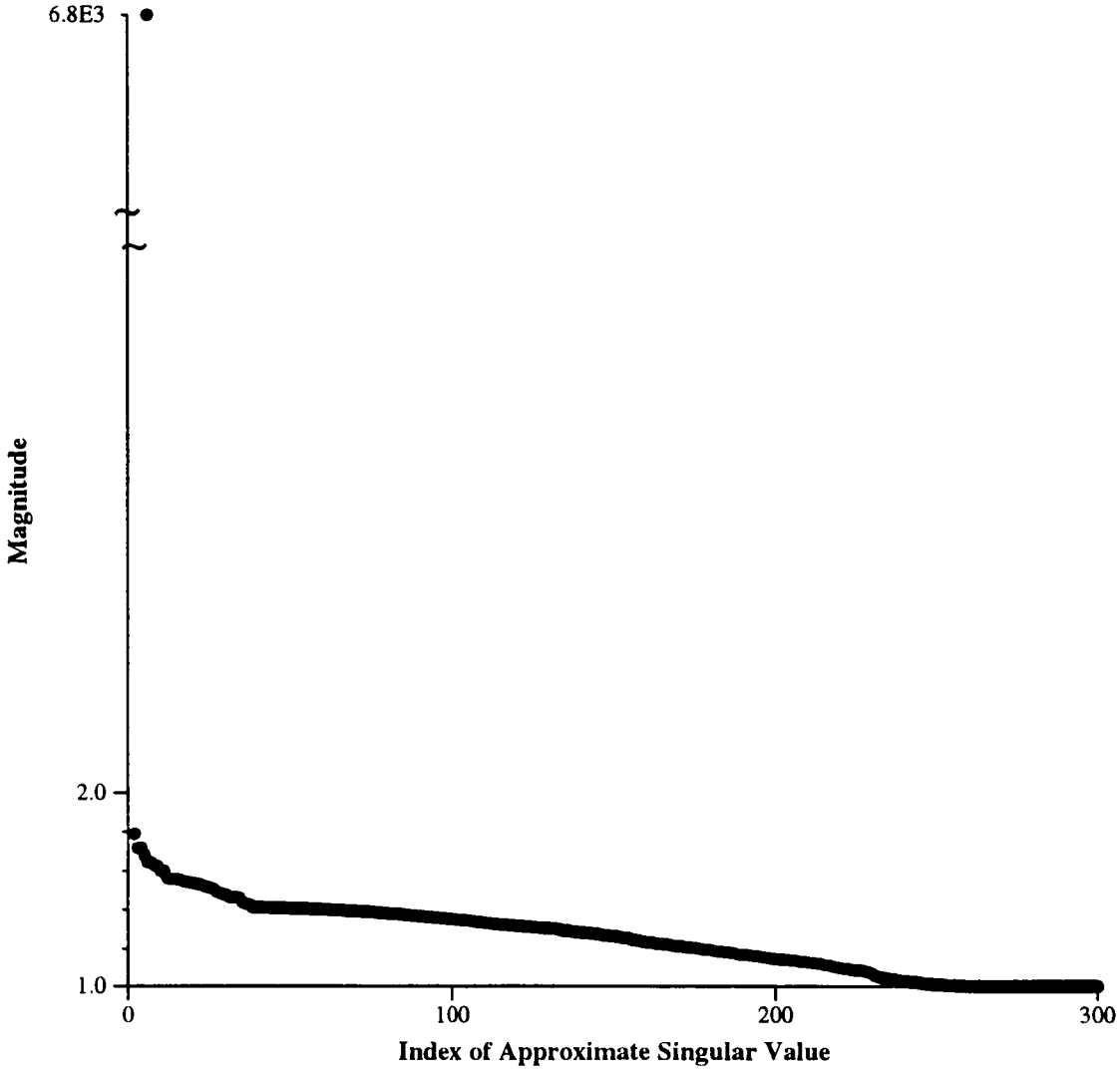


Figure A.2: The 300-largest singular values of dataset WELL in Table 6.1.

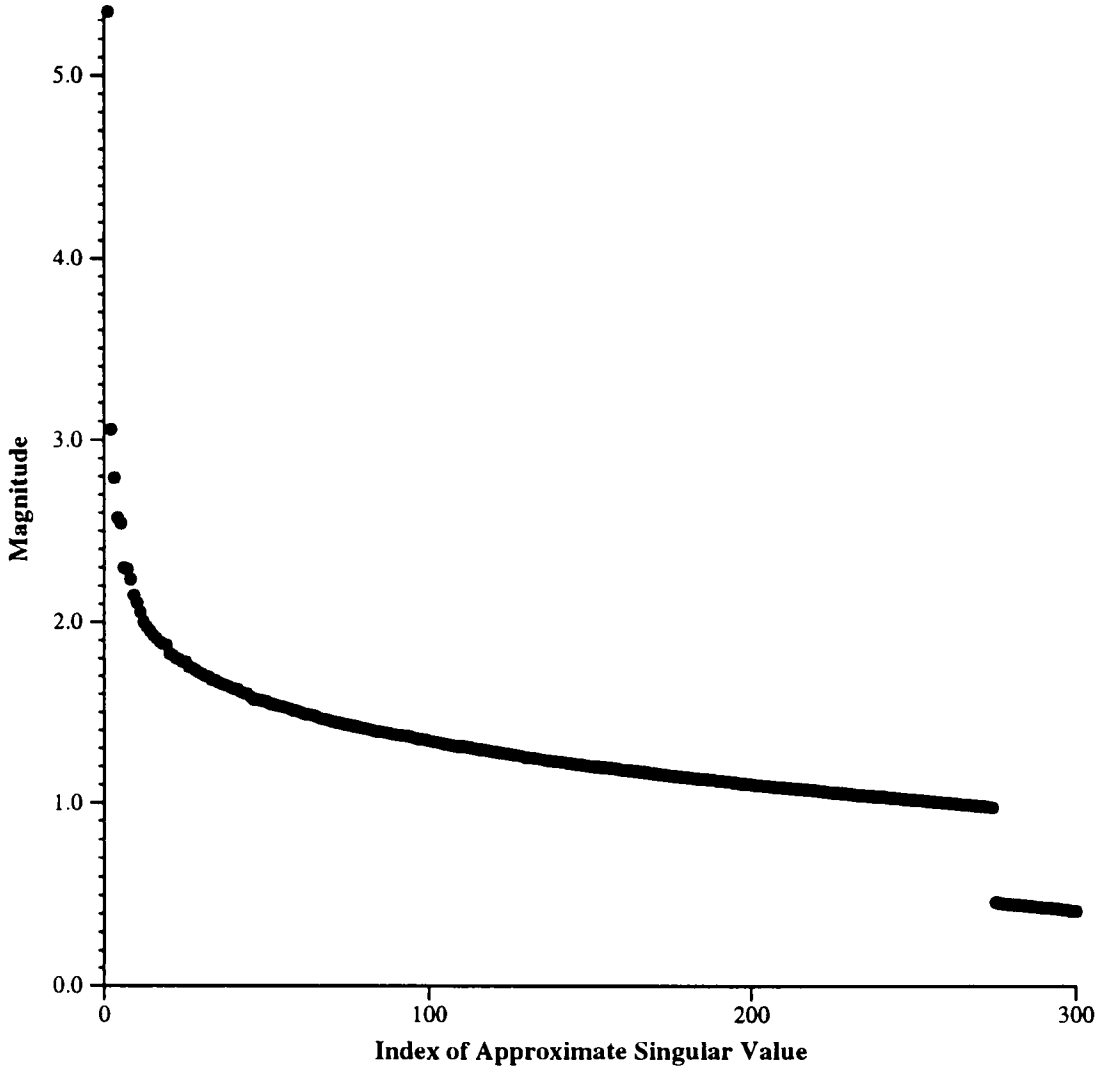


Figure A.3: The 300-largest singular values of dataset LATERAL2 in Table 6.1.

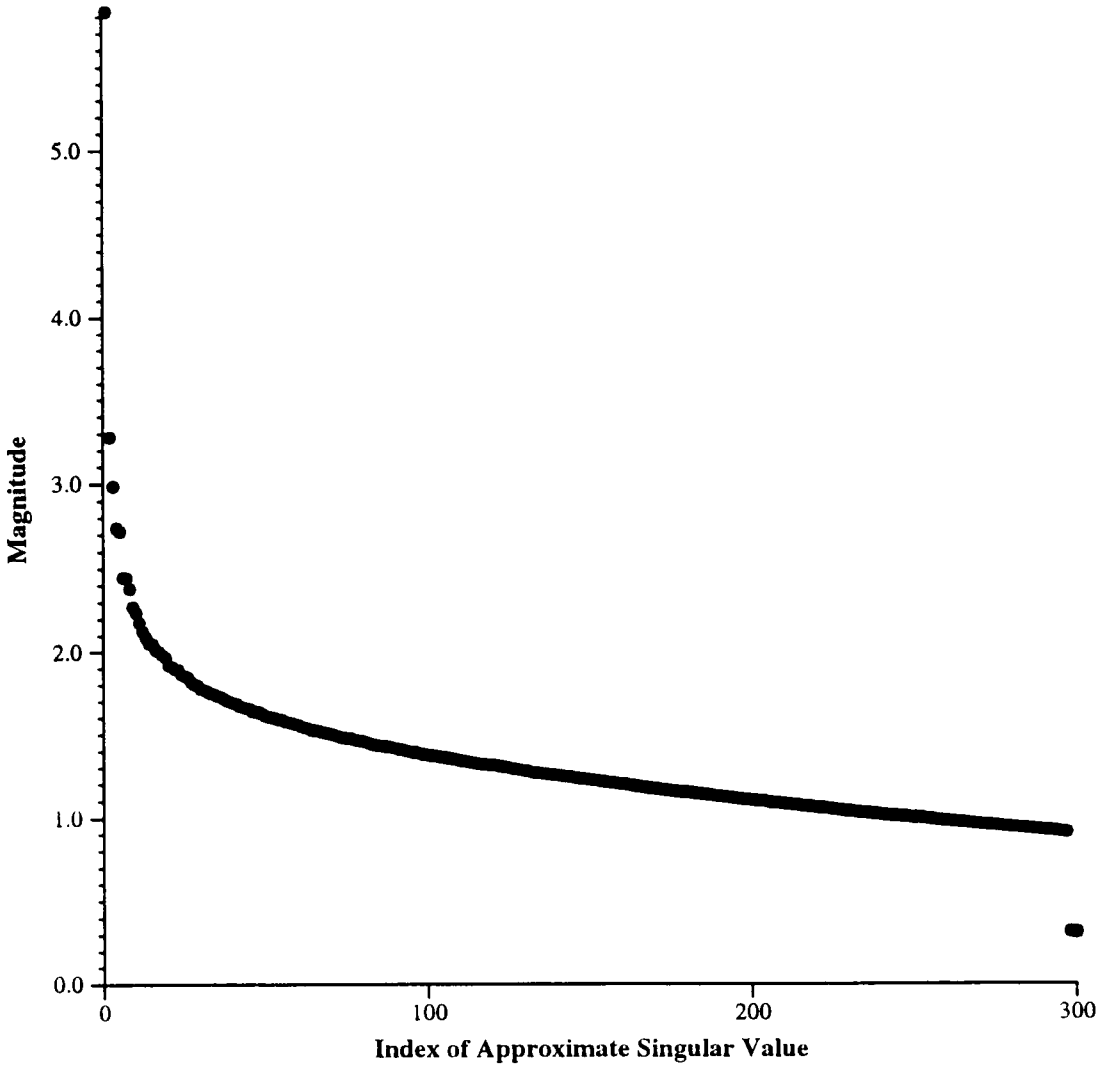


Figure A.4: The 300-largest singular values of dataset LATERAL5 in Table 6.1.

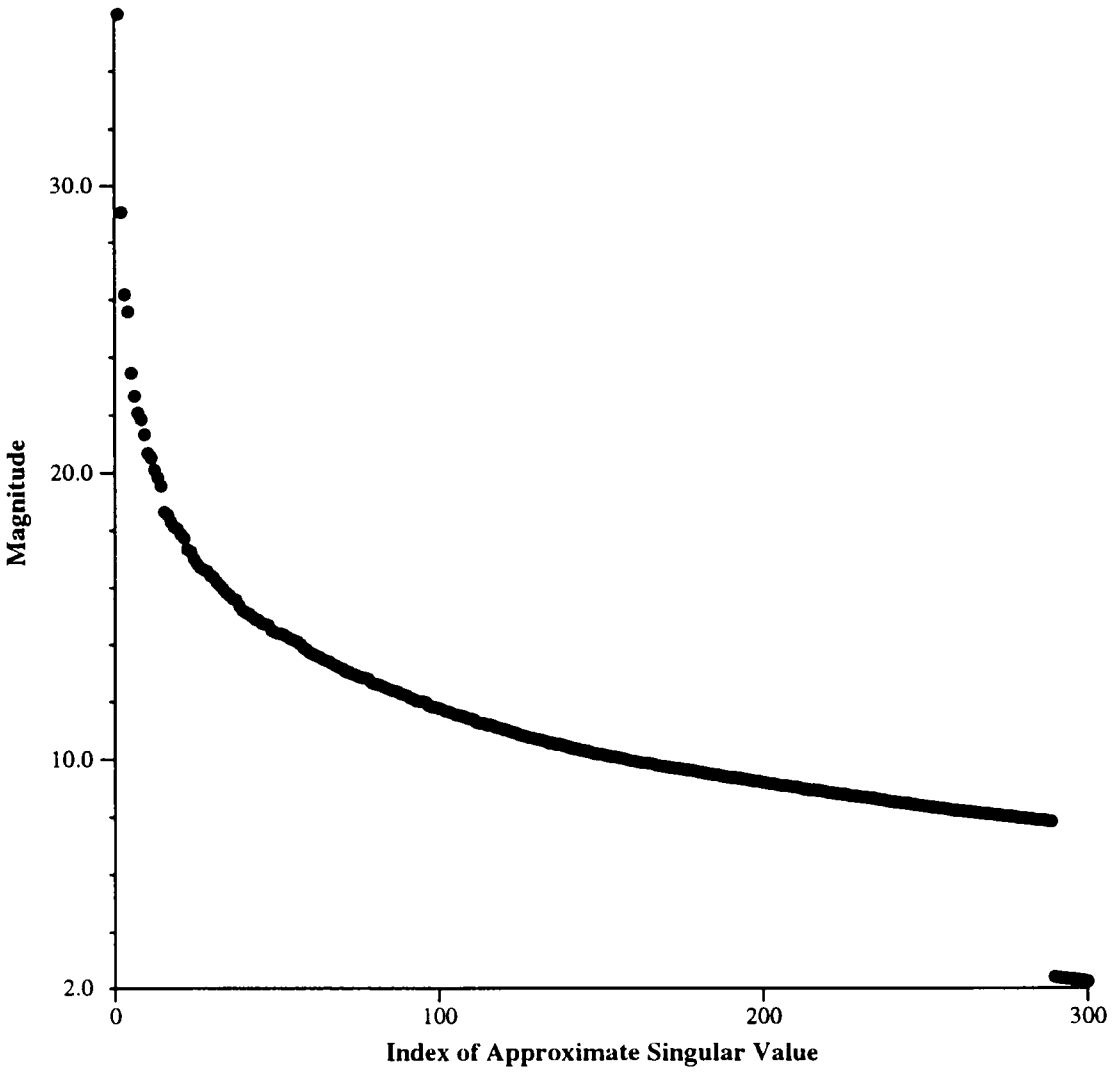


Figure A.5: The 300-largest singular values of dataset MED in Table 6.1.

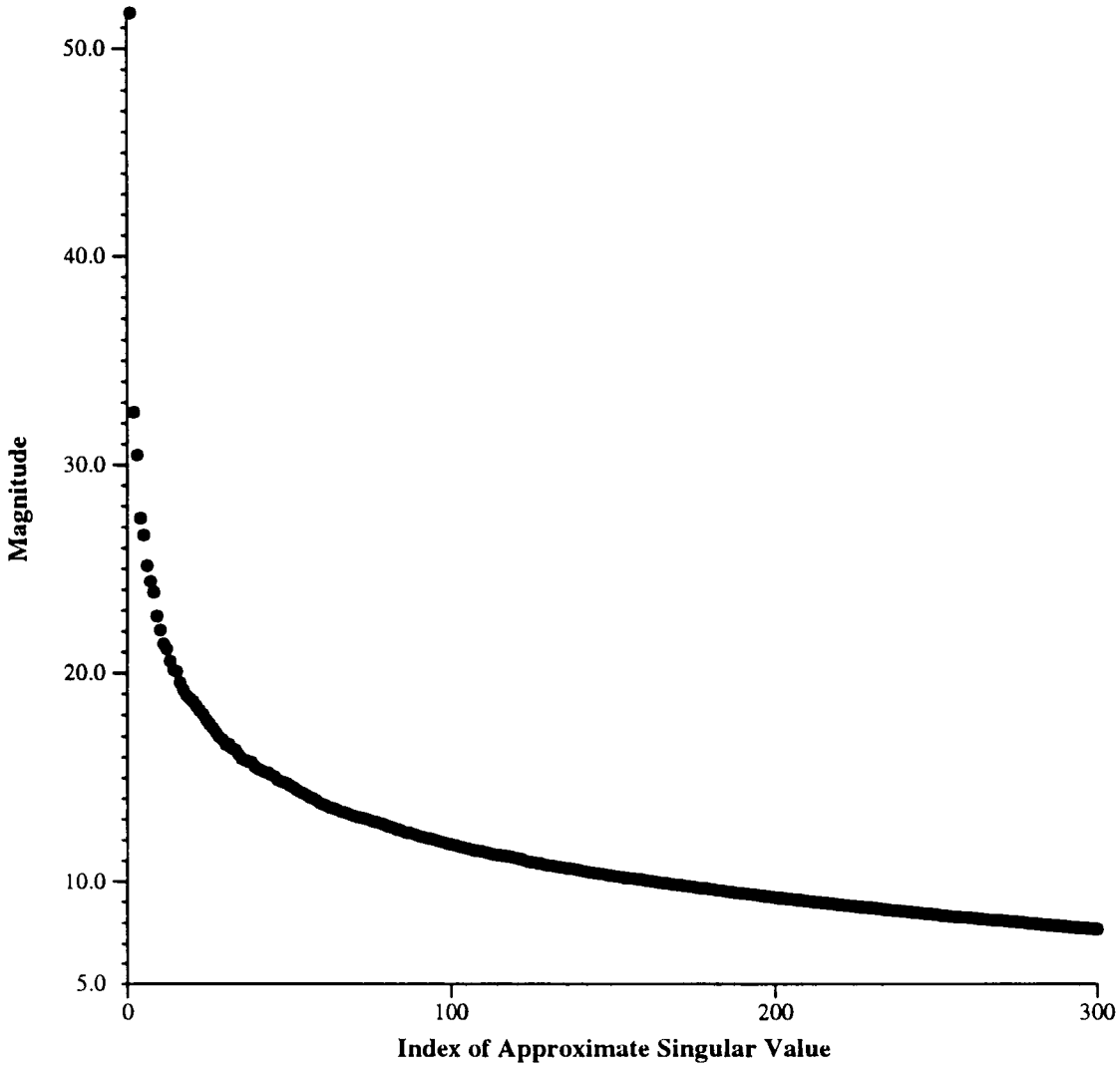


Figure A.6: The 300-largest singular values of dataset CRAN in Table 6.1.

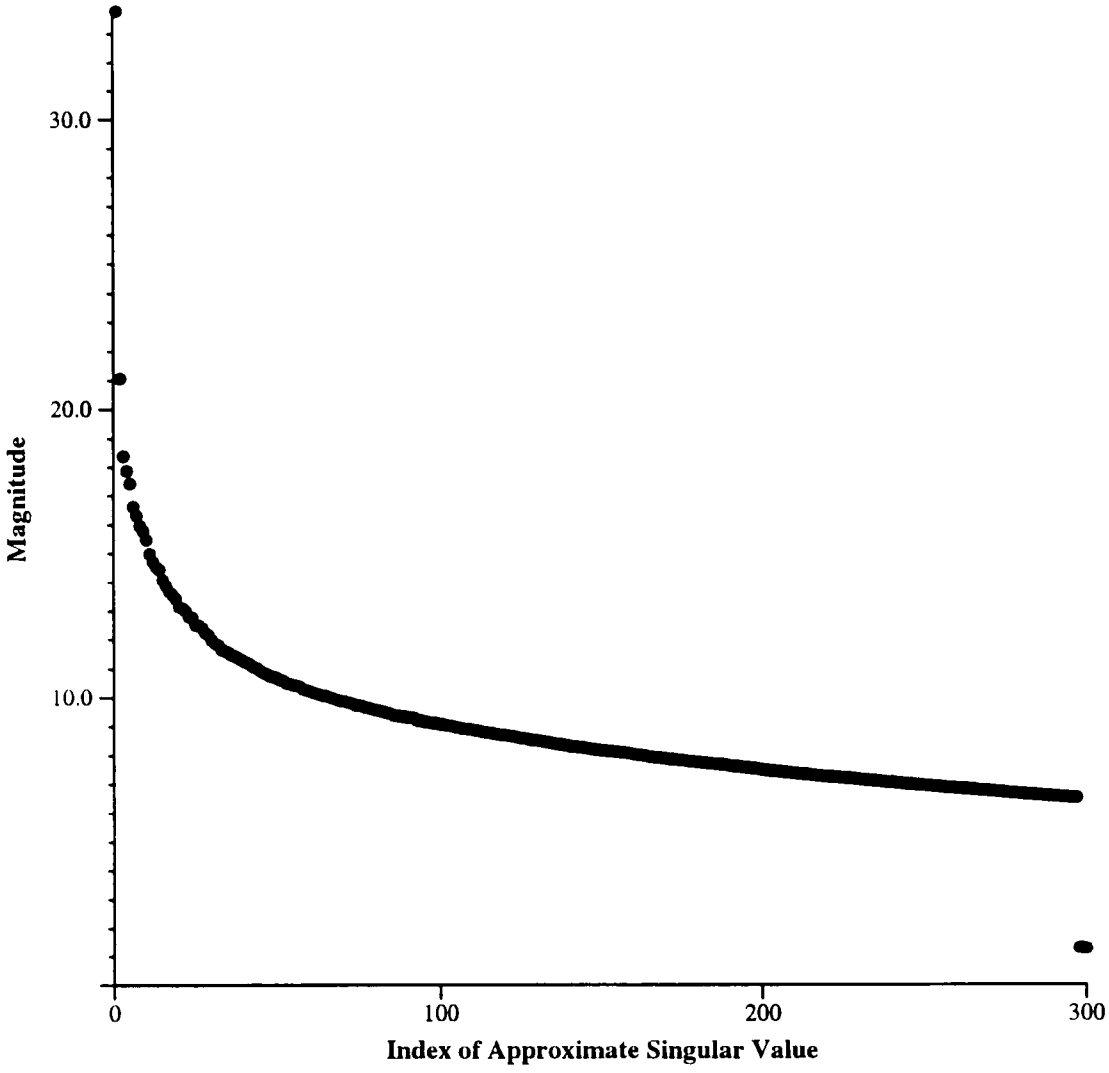


Figure A.7: The 300-largest singular values of dataset CISI in Table 6.1.

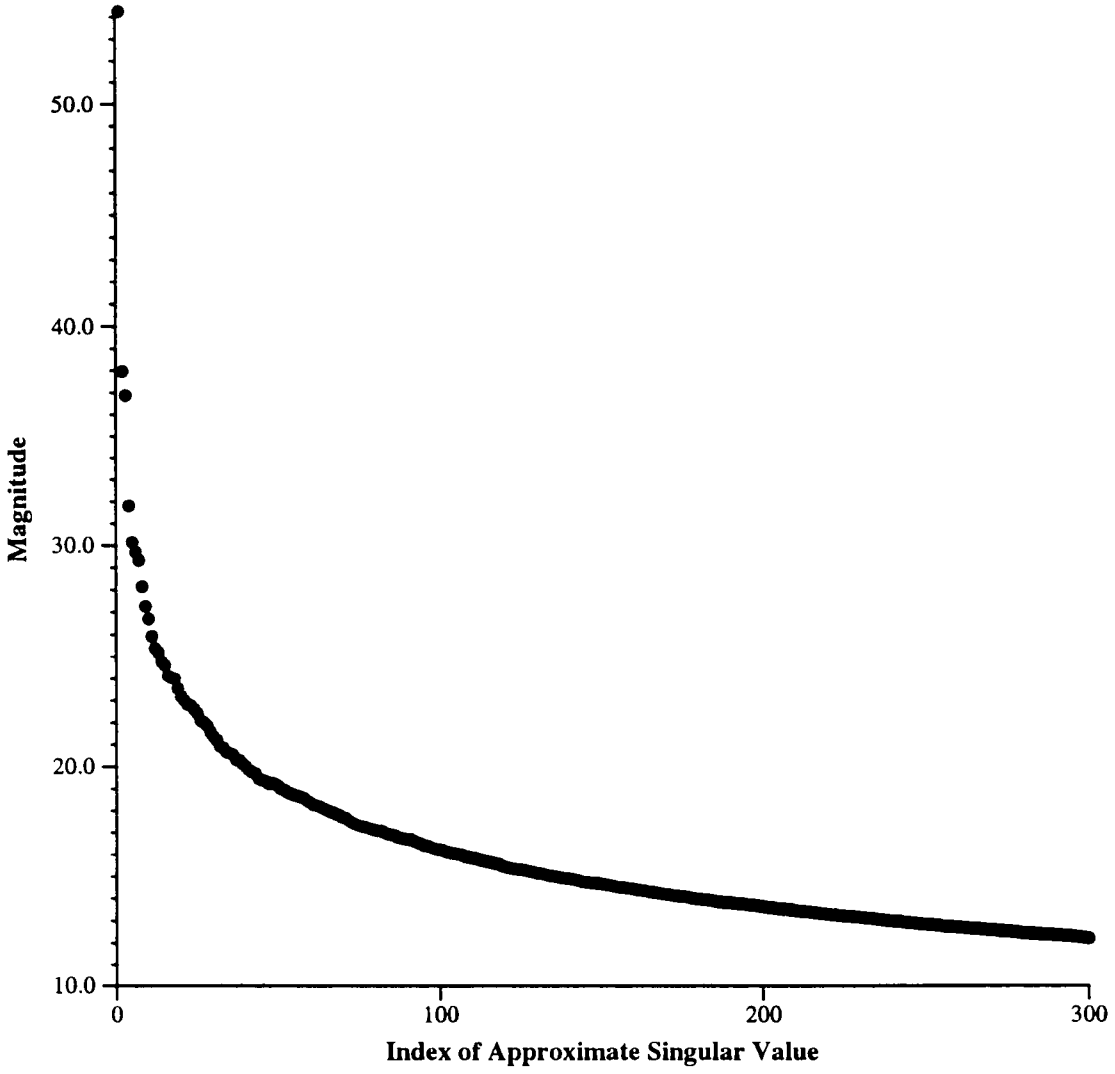


Figure A.8: The 300-largest singular values of dataset TECH in Table 6.1.

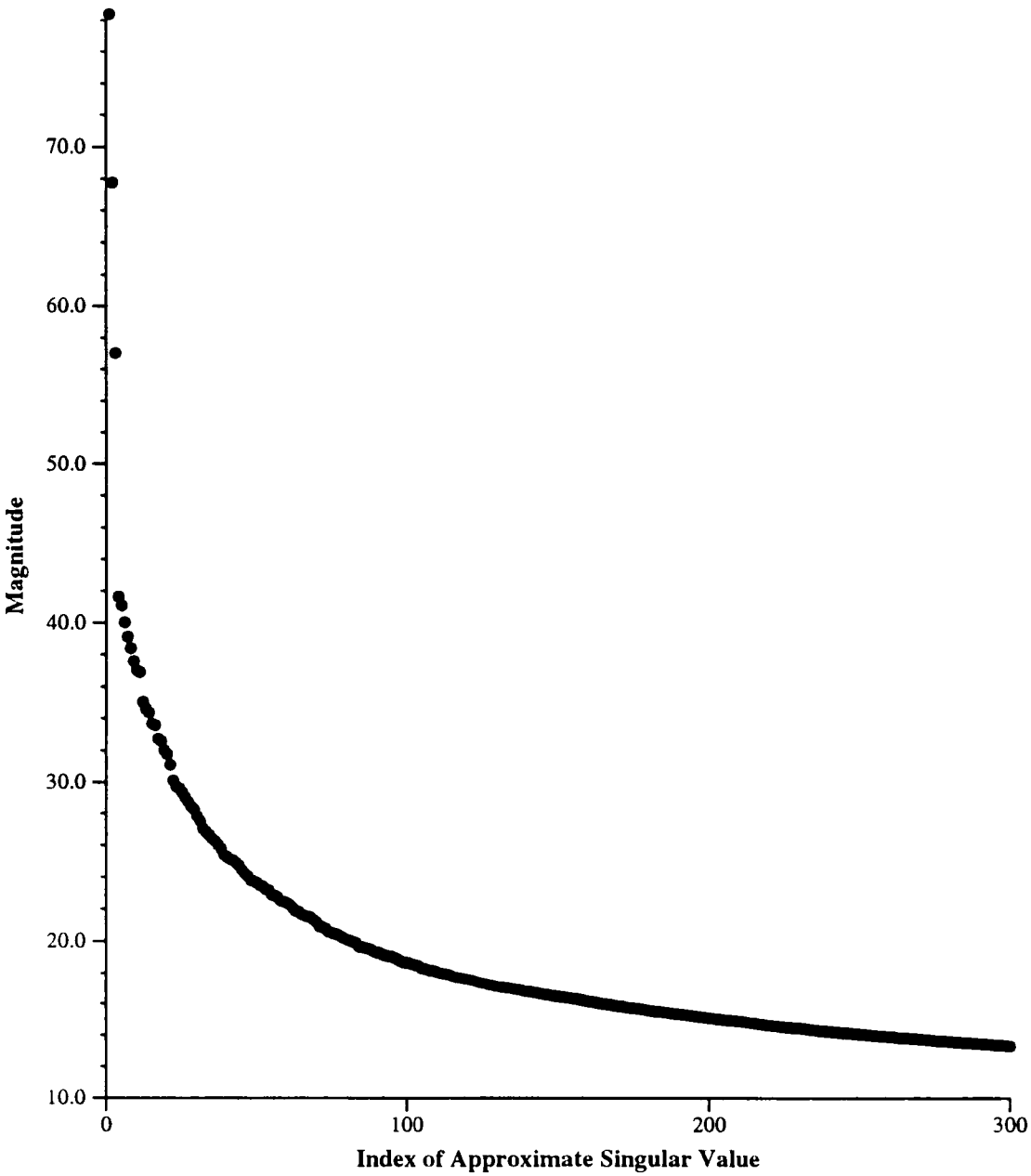


Figure A.9: The 300-largest singular values of dataset CUPENC in Table 6.1.

Appendix B

Prologues of Selected Procedures

```
 void init(int *ibuff, double *dbuff, double **pbuff)

/*************************************************************************
 *                                                                       *
 *                        init() of LAS1C                                *
 *                                                                       *
 *************************************************************************/
/*************************************************************************

   Description
   -----------

   init() is the initializing routine called by the LAS1C driver routine to

      o  read in the sparse matrix
      o  check validity of input parameters
      o  allocate neccessary memory
      o  write header to output file


   arguments
   ---------

   (output)
   ibuff[0] = number of rows of matrix A
   ibuff[1] = number of cols of matrix A
   ibuff[2] = order of the 2-cyclic matrix B (ibuff[0] + ibuff[1])
   ibuff[3] = number of nonzeros in matrix A
   ibuff[4] = maximum number of iterations allowed
   ibuff[5] = number of singular triplets of A (eigenpairs of B) desired
   ibuff[6] = a single integer between 0 and 7 with bit representation
              b1_b2_b3
              b1 = 0 ASCII file containing input matrix is expected
                 = 1 input matrix is to be read from standard input
              b2 = 0 singular vectors are stored in memory and binary file
                 = 1 singular vectors are stored in binary file only
                   note:  b2 is checked only if ibuff[7] = 1
              b3 = 0 Lanczos vectors are stored in memory
                 = 1 Lanczos vectors are stored in binary file
   ibuff[7] = 1 if singular triplets of A are needed
              0 if only singular values of A are needed
```

```
  dbufb[0] = left-end interval of an interval within which all unwanted
             eigenvalues of B lies
  dbufb[1] = right-end interval of an interval within which all
             unwanted eigenvalues of B lies
  dbuff[2] = relative accuracy of Ritz values acceptable as eigenvalues
             of B

  pbuff[0] = address of starting Lanczos vector
  pbuff[1] = address of array storing Ritz values
  pbuff[2] = address of array storing error bounds of Ritz values
  pbuff[3] = address of array storing approximate singular values of
             matrix A
  pbuff[4] = address of storage area for Lanczos vectors

*************************************************************************/
```

Note: Function `init()` of LAS2C is similar to that of LAS1C given above.

```
visible void mp_init(void *fe_infile, void *fe_buff)

/*************************************************************************
 *                                                                       *
 *                        mp_init() of LAS1P                             *
 *                                                                       *
 *************************************************************************/
/*************************************************************************

   Description
   -----------

   mp_init() is invoked by init() function from the front-end to initialize
   the back-end of the MasPar (DPU) to

      o  read in and distribute the sparse matrix
      o  allocate neccessary memory in PE array


   arguments
   ---------

   (input)
   fe_infile  name of file containing input matrix

   (output)
   fe_buff[0]  number of rows of matrix A
   fe_buff[1]  number of columns of matrix A
   fe_buff[2]  linear array size needed to allocate in front-end to
               accomodate data transfer between front-end and DPU
   fe_buff[3]  number of nonzeros in matrix A

 *************************************************************************/
```

Note: Function `mp_init()` of `LAS2P` is similar to that of `LAS1P` given above.

```
void store(int n, int isw, int j, double *s)

/*****************************************************************************
 *                                                                           *
 *                              store()                                      *
 *                                of                                         *
 *                  LAS1C, LAS2C, LAS1P, and LAS2P                           *
 *                                                                           *
 *****************************************************************************/
/*****************************************************************************

   Description
   -----------

   store() is a user-supplied function which, based on the input
   operation flag, stores to or retrieves from memory a Lanczos vector.
   Upon entering, store() checks the global memory management flag to
   see whether the vectors are stored in physical memory or on a local
   disk.


   Arguments
   ---------

   (input)
   n       length of vector to be stored or retrieved
   isw     operation flag:
     isw = 1 request to store j-th Lanczos vector q[j]
     isw = 2 request to retrieve j-th Lanczos vector q[j]
     isw = 3 request to store q[j] for j = 0 or 1
     isw = 4 request to retrieve q[j] for j = 0 or 1
   s    contains the vector to be stored for a "store" request

   (output)
   s    contains the vector retrieved for a "retrieve" request


   Functions used
   --------------
   BLAS dcopy
 *****************************************************************************/
```

```
void lanczos_step(int n, int first, int last, double *wptr[],
                  double *alf, double *eta, double *oldeta,
                  double *bet, int *ll, int *enough)

/*************************************************************************
 *                                                                       *
 *                           lanczos_step()                              *
 *                                of                                     *
 *                   LAS1C, LAS2C, LAS1P, and LAS2P                      *
 *                                                                       *
 *************************************************************************/
/*************************************************************************
   Description
   -----------
   Function performs one iteration of the Lanczos recursion.  In this
   recursion, diagonal and subdiagonal elements of tridiagonal matrix
   T are determined (T is the orthogonal projection of B onto the
   j-dimensional Krylov subspace)


   Arguments
   ---------
   (input)
   n        dimension of the eigenproblem for matrix B
   first    start of index through loop
   last     end of index through loop (size of subspace dimension)
   wptr     array of pointers pointing to work space
   alf     array to hold diagonals of the tridiagonal matrix T
   eta      orthogonality estimate of Lanczos vectors at step j
   oldeta   orthogonality estimate of Lanczos vectors at step j-1
   bet      array to hold off-diagonals of T
   ll       has value of 0, 1 or 2; number of intitial Lanczos
    vectors in local orthogonalization
   enough   stop flag

   (output)
   all arguments are updated upon exiting function


   Functions used
   --------------
```

```
  BLAS          ddot, dscal, daxpy, datx, dcopy
  USER          store
  LAS           purge, ortbnd, startv
  UTILITY       imin, imax
*************************************************************************/
```

Note: Function `lanczos_step()` of `LAS1P`(LS) and `LAS2P`(LS) has the same arguments as that which is given above but its function is to invoke `mp_lanczos_step()`. The Lanczos recursion (2.12) is performed in `mp_lanczos_step()`, and the BLAS kernels it invokes are the parallel counterparts of those listed in the prologue above.

Appendix C

Flow Chart of LANSO

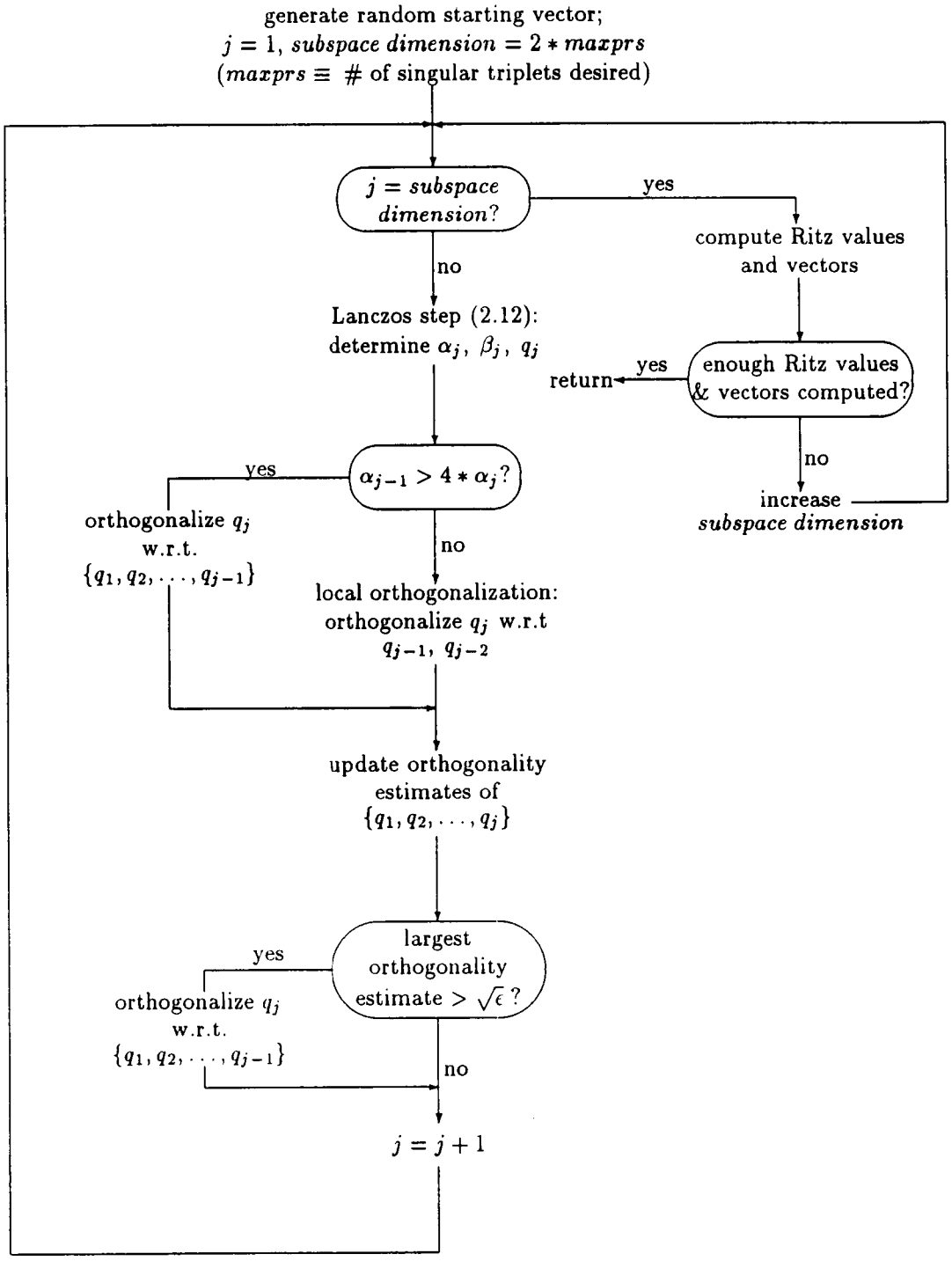


Figure C.1: A flow chart of LANSO which implements the single-vector Lanczos algorithm with selective re-orthogonalization as described in Section 2.2.

VITA

Theresa Hong Do was born in Vietnam on June 25, 1963. She graduated from Rhea County High School in Evensville, Tennessee in 1981 and received the Bachelor of Science degree in Electrical Engineering from Tennessee Technological University in May 1985. In August 1990, she entered the Computer Science program at the University of Tennessee and was awarded the Master of Science degree in Computer Science in May 1993.