Estimating the Largest Singular Values/Vectors of Large Sparse Matrices via Modified Moments

Sowmini Varadhan

Computer Science Department

CS-96-319 February 1996

Estimating the Largest Singular Values/Vectors of Large Sparse Matrices via Modified Moments

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

SOWMINI VARADHAN
December 1995

Acknowledgments

I thank my dissertation advisor, Dr. Michael Berry, for his support, guidance and patience. I also thank my committee members Prof. Jack Dongarra, Dr. Jens Gregor and Prof. Tse-wei Wang, and Prof. Gene Golub for his helpful comments and advice. I am grateful to Dr. Margaret Simmons, Dr. Ken Koch and the Los Alamos National Lab for providing me access to the CRAY T3D. Thanks are also due to Mark Dalton for his support in using the CRAY T3D, and Bob Manchek for his advice and help in using PVM. I would also like to thank Jane Comiskey and Karen Minser for their assistance in proof-reading this dissertation.

Abstract

This dissertation considers algorithms for determining a few of the largest singular values and corresponding vectors of large sparse matrices by solving equivalent eigenvalue problems. The procedure is based on a method by Golub and Kent for estimating eigenvalues of equvalent eigensystems using modified moments. The asynchronicity in the computations of moments and eigenvalues makes this method attractive for parallel implementations on a network of workstations. However, one potential drawback to this method is that there is no obvious relationship between the modified moments and the eigenvectors. The lack of eigenvector approximations makes deflation schemes difficult, and no robust implementation of the Golub/Kent scheme are currently used in practical applications. Methods to approximate both eigenvalues and eigenvectors using the theory of modified moments in conjunction with the Chebyshev semi-iterative method are described in this disseratation. Deflation issues and implicit error approximation methods are addressed to present a complete algorithm. The performance of an ANSI-C implementation of this scheme on a network of UNIX workstations using PVM is presented. The portability of this implementation is demonstrated through results on a 256 processor Cray T3D massively-parallel computer.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The singular value decomposition (SVD) is commonly used in the solution of unconstrained linear least squares problems, matrix rank estimation, and canonical corelation analysis. In applications such as information retrieval, seismic reflection tomography and real-time signal processing, the SVD of large, sparse input matrices is required in the shortest possible time. Given the growing availability of multiprocessor computer systems, there has been great interest in the development of efficient implementations of the singular value decomposition that can utilize the parallel processing power available in multiprocessor environments provided through distributed computing and massively-parallel platforms. The goal of this dissertation is to describe and develop a parallel algorithm for computing the SVD of unstructured, sparse matrices. First, a few of the fundamental characterizations of the SVD are reviewed.

Given an $m \times n$ matrix A, where $m \geq n$ and 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 $U^TU = I_m$, $V^TV = I_n$, and $\Sigma = \text{diag}(\sigma_1, \cdots, \sigma_n), \sigma_i > 0$ 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 orthonormal eigenvectors associated with the r nonzero eigenvalues of AA^T and A^TA, respectively. U and V are referred to as the left and right singular vectors, 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. A discussion of the properties of the SVD, and its applications can be found in the literature (e.g., [GL89], [Ste73]).

The SVD can reveal important information about the structure of a matrix, as illustrated by the following two well-known theorems [Ber92].

Theorem 1.1 *Let the SVD of A be given by Equation (1.1) with*

$$\sigma_1 \geq \sigma_2 \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0,$$

and let $\mathcal{R}(A)$ and $\mathcal{N}(A)$ denote the range and null space of A, respectively. Then the following properties hold.

1. *Rank property:* $rank(A) = r, \mathcal{N}(A) \equiv span\{v_{r+1}, \cdots, v_n\}$, *and* $\mathcal{R}(A) \equiv span\{u_1, \cdots, u_r\}$, *where* $U = [u_1 u_2 \cdots u_m]$ *and* $V = [v_1 v_2 \cdots v_n]$.

2. *Dyadic decomposition:* $A = \sum_{i=1}^{r} \sigma_i \cdot u_i \cdot v_i^T$.

3. *Norms:* $\|A\|_F^2 = \sigma_1^2 + \cdots + \sigma_r^2$ *and* $\|A\|_2^2 = \sigma_1$. *Here,* $\|.\|_F$ *denotes the Frobenius norm defined by*

$$\|A\|_F{}^2 = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2},$$

and $\|.\|_2$ *denotes the 2-norm defined by*

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} \; p = 1, 2, \ldots \infty \tag{1.2}$$

with p=2;

Due to the rank property, it is possible to use the singular values of A as quantitative measures of the qualitative notion of rank. The dyadic decomposition, which is the rationale for data reduction or compression in many applications, provides a canonical description of a matrix as a sum of r rank-one matrices of decreasing importance, as measured by the singular values. The three results in Theorem 1.1 can be combined to yield the following quantification of matrix rank deficiency (see [GL89] for a proof):

Theorem 1.2 (Eckart and Young) *Let the SVD of* A *be given by Equation (1.1) with* $r = rank(A) \leq p = \min(m, n)$ *and define*

$$A_k = \sum_{i=1}^{k} \sigma_i \cdot u_i \cdot v_i^T, \tag{1.3}$$

$$\min_{rank(B)=k} \|A - B\|_F^2 = \|A - A_k\|_F^2 = \sigma_{k+1}^2 + \cdots + \sigma_p^2.$$

The SVD(A) may be computed from two equivalent eigenvalue decompositions:

1. Define the 2-cyclic matrix

$$C = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}. \tag{1.4}$$

If $rank(A) = n$, it can be shown that the eigenvalues of C are the n pairs $\pm\sigma_i$, where σ_i is a singular value of A, with $(m - n)$ additional zero eigenvalues if $m > n$. The multiplicity of the zero eigenvalue of C is $m + n - 2r$, where $r = rank(A)$.

2. Alternatively, the SVD(A) can be computed indirectly by the eigenpairs of either the $n \times n$ matrix $A^T A$ or the $m \times m$ matrix AA^T. The following lemma illustrates the fundamental relations between these symmetric eigenvalue problems and the SVD.

   Lemma 1.1 *Let A be an $m \times n$ matrix with $m \geq n$ and $rank(A) = r$.*

   (a) If $V = (v_1, v_2, \ldots, v_r)$ are linearly independent $n \times 1$ eigenvectors of $A^T A$ so that $V^T(A^T A)V = diag(\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2)$, then σ_i is the i^{th} nonzero singular value of A corresponding to the right singular vector v_i. The corresponding left singular vector, u_i, is then obtained as $u_i = (1/\sigma_i)Av_i$.

   (b) If $U = (u_1, u_2, \ldots, u_r)$ are linearly independent $m \times 1$ eigenvectors of AA^T so that $U^T(AA^T)U = diag(\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2)$, then σ_i is the i^{th} nonzero singular value of A corresponding to the left singular vector u_i. The corresponding right singular vector, v_i, is then obtained as $v_i = (1/\sigma_i)A^T u_i$.

Each scheme described above has its own advantages and disadvantages. The matrix $A^T A$ is of order at most n, whereas the two-cyclic matrix C defined in Equation (1.4) is of order at most $(m + n)$. If the matrix A is over-determined, i.e. $m \gg n$, the smaller memory requirements for the $n \times n$ matrix $A^T A$ make it the more attractive choice for computing the SVD. However, this scheme only gives the right-singular vectors, and the left-singular vectors have to be obtained by scaling Av_i as defined in Lemma 1.1 . The eigenvectors of the two-cyclic matrix, on the other hand, are of the form $[{u_i}^T, {v_i}^T]^T$, and directly give complete information about the *singular triplet*: $\{u_i, \sigma_i, v_i\}$. Also, each eigenvalue of $A^T A$ is ${\sigma_i}^2$, forcing a clustering of the singular value approximations when $\sigma_i < 1$. Evaluating the SVD from the eigen-decomposition of $A^T A$ is thus most suited for problems when only the largest singular values are desired, with a potential loss of accuracy for the smaller singular-values. Using the two-cyclic matrix C does not have this drawback, however, at the price of a larger memory requirement.

Although several direct methods (e.g., Householder's method) exist for computing the eigenvalues of the canonical matrices described in Lemma 1.1, these methods are undesirable when applied to large, unstructured sparse matrices. Direct methods involve factorizations which result in intermediate, full submatrices. For sparse eigenvalue problems, the input matrix is typically stored in compressed format, and the undesirable fill-in and large memory requirements associated with direct methods limit their applicability to such problems.

Preferred methods for large, sparse, symmetric eigenproblems include the Lanczos method [GL89] and Arnoldi's method [Saa92]. Other related methods include subspace-iteration [Par80], and trace minimization [Ber92]. All of these methods obtain approximations to the eigenvalues and vectors of symmetric matrix A by constructing elements from a Krylov-like basis through the operation $A\mathcal{S} \equiv \{\mathbf{As} : \mathbf{s} \in \mathcal{S}\}$ [Par80] for the subspace spanned by the eigenvectors. Thus, the matrix A is used only to

compute the matrix-vector product As, and these algorithms may be implemented without making any assumptions about the structure/storage format of A. The efficiency of these methods is determined primarily by the performance of the matrix-vector product and the storage scheme used for the matrix.

The Lanczos method solves the eigenvalue problem $Cx = \lambda x$ through partial tridiagonalizations of the matrix C. Unlike factorization methods, no intermediate, full submatrices are generated. Also, information about C's extremal eigenvalues tend to emerge long before tridiagonalization is complete. Hence, the Lanczos algorithm particularly useful in situations where only a few of C's largest or smallest eigenvalues are desired.

An alternative method, CSI-MSVD, to tridiagonalize large, sparse matrices is presented in [GK89]. This method is based on the extraction of modified moments from the Chebyshev semi-iterative method [Var62]. The attractiveness of this algorithm lies in the scope for parallelism and reduced memory requirements.

A brief review of Krylov subspace methods will now be presented in order to provide sufficient background information for the CSI-MSVD algorithm.

The Krylov subspace is of considerable importance in the theory of iterative methods for the solution of the eigenvalue problem $Cx = \lambda x$. The Krylov subspace associated with the $m \times m$ matrix C with real elements are determined by a single non-zero vector f by

$$K^m(f) = \{f, Cf, \ldots, C^{(m-1)}f\}$$
$$\text{and denote } \mathcal{K}^m = \text{span } K^m(f).$$

Theoretically, the natural basis for $\mathcal{K}^m$ is the Krylov basis $K^m(f)$. In practice, the orthonormal basis

$$Q_m \equiv (q_1, \ldots, q_m), \tag{1.5}$$

obtained by the QR factorization of the columns of $K^m(f)$ is used as a basis for $\mathcal{K}^m$. If the dimension of $\mathcal{K}^m$ is m, i.e., $K^m(q_1)$ has full rank, it can be shown that ${Q_m}^* C Q_m$ is an unreduced tridiagonal matrix [Ste73], where * denotes the Hermitian-transpose. Thus, one way to view the Lanczos process is as a construction of the Krylov basis for C. Alternatively, the Lanczos process can be viewed as a method for reducing C to tridiagonal form [Par80].

A third method of deriving the Lanczos algorithm is by considering the relationship between Krylov subspaces and orthogonal polynomials. The Krylov subspace $\mathcal{K}^m$ can be considered as the subspace of all vectors in $\mathbb{R}^n$ which can be written as $x = p(C)v$, where $p(x)$ is a polynomial of degree not exceeding $m-1$. Let $p_m(x)$ be the nonzero monic polynomial of lowest degree such that $p_m(v) = 0$. Then it can be shown [Saa92] that $\mathcal{K}^m$ is of dimension m if and only if the degree of the minimal polynomial with respect to C is larger than $m-1$. Thus, it is possible to derive the isomorphism between $\mathbb{P}_{m-1}$ and $\mathcal{K}_m$, the space of polynomials of degree $\leq m-1$ defined by

$$q \in \mathbb{P}_{m-1} \rightarrow x = q(C)v_1 \in \mathcal{K}^m.$$

The subspace $\mathbb{P}_{m-1}$ is typically associated with an inner product

$$\langle p, q \rangle_{v_1} = (p(C)v_1, q(C)v_1) \tag{1.6}$$

which is a nondegenerate bilinear form.

The Lanczos vectors v_i are of the form

$$v_i = q_{i-1}(C)v_1 \text{ with}$$
$$q_i(x) = x^i. \tag{1.7}$$

The orthogonality of the $v_i's$ is equivalent to the orthogonality of the polynomials, with respect to the inner product defined in Equation (1.6). A discussion of the relations between the Lanczos biorthogonalization method [GL89], the theory of orthogonal polynomials, and Gaussian quadrature is provided in [Bre80].

Several possibilities exist for the choice of the orthogonal polynomials $q_i(x)$ in Equation (1.7). Golub and Kent in [GK89] propose a scheme to approximate eigenvalues using Chebyshev polynomials [GV61] for $q_i(x)$. The inner products generated by Equation (1.6) are used to derive moments [Wil62] which are then used in the modified Chebyshev algorithm [Gau82] to generate a set of orthogonal polynomials. The quasi-symmetric tridiagonal matrix obtained from the coefficients of the orthogonal polynomials, known as the Jacobi matrix, is then used to approximate the eigenvalues of the matrix C.

Berry and Golub in [BG91] have shown the effectiveness of the scheme proposed in [GK89] through an implementation on the Cray Y-MP that approximates singular values of a sparse matrix A from the eigenvalues of the corresponding two-cyclic matrix C defined by Equation (1.4). They also point out that the potential asynchronous computation of the orthogonal polynomials with the iterations of an adaptive Chebyshev semi-iterative method allows multiple processors to execute different sections of the algorithm in parallel. Thus, there is more inherent scope for parallelism with this scheme than with Lanczos algorithms. Also, the recurrence relations defining the Chebyshev polynomials allow an accelerated construction of the moments, and it is possible to approximate the singular values in relatively few iterations.

The implementation proposed in [BG91] describes a scheme to approximate singular values only, i.e., singular vectors are not estimated. The tridiagonal Jacobi matrix is constructed to approximate, through its eigenvalues, the roots of the characteristic equation. Moreover, while recurrence relations may be used to accelerate the construction of modified moments from the Chebyshev iterates, there is no known scheme to obtain the converse result, i.e., to approximate the Chebyshev iterates (and thus the singular vectors) from the moments. The absence of singular-vector (and eigenvector) approximations gives rise to problems such as the lack of an error estimate on the approximated singular value (or eigenvalue) and inefficient deflation schemes. Due to these problems, there has been no robust, complete implementation of this algorithm, in spite of its theoretical efficiency and suitability for parallel computers.

This dissertation presents one scheme to obtain the SVD from the canonical eigenvalue problems described in Lemma 1.1. Methods to approximate both eigenvalues and eigenvectors using the theory of modified moments in conjunction with the Chebyshev semi-iterative method are described. Deflation issues and implicit error approximation methods are addressed to present a complete algorithm. The performance of an ANSI-C implementation of this scheme on a network of UNIX workstations using PVM [GBD+94] is presented. The portability of this implementation is demonstrated through results on a 256 processor Cray T3D massively-parallel computer. A synopsis of the remainder of this dissertation is as follows.

The mathematical backgrounds of iterative methods, and the theory of modified moments are presented in Chapter 2, followed by a detailed description of the CSI-MSVD algorithm and associated problems in approximating eigenvectors using this scheme. Solutions to these problems are also addressed, and a complete algorithm is described. Chapter 3 describes typical application areas in which sparse SVD problems are encountered and provides an overview of issues related to sparse matrix manipulation. The methodology and computational environments used to evaluate performance of the CSI-MSVD algorithm is described in Chapter 4 and the results obtained on various platforms are presented in Chapter 5. Lastly, Chapter 6 presents conclusions and future work.

Chapter 2

Algorithms

Following [GK89], consider the basic iteration

$$x^{(m+1)} = Mx^{(m)} + b \tag{2.1}$$

to solve the system of linear equations

$$(I - M)x = b, \tag{2.2}$$

where M is either the $n \times n$ matrix $A^T A$ or the $(m+n) \times (m+n)$ matrix defined by Equation (1.4), and M is suitably scaled so that $\rho(M) < 1$. As shown in Section 2.1 of this chapter, the Chebyshev semi-iterative method [GV61] produces an alternative iteration to Equation (2.1) of the form

$$\xi^{(k)} = p_k(M)\xi^{(0)}, \tag{2.3}$$

where $p_k(M)$ is a polynomial of degree k in M, and $\xi^{(k)}$ is a column vector of dimension $(m+n) \times 1$ or $n \times 1$ depending on whether M is the two-cyclic matrix of Equation (1.4) or the matrix $A^T A$. Sections 2.2 and 2.3 of this chapter discuss how one can estimate the eigenvalues of M (corresponding to the largest singular values of A) using Equation (2.3) with the method of modified moments. The next section reviews some of the theory of iterative methods addressing issues such as convergence criteria and rates of convergence to establish the optimality of the Chebyshev semi-iterative method. A three-term recurrence for the Chebyshev iterates $\xi^{(k)}$ defined by Equation (2.3) will be derived.

The following notation will be used for matrix and vector operations for the remainder of the dissertation. The vector space of all $m \times n$ real matrices is denoted by $\mathbb{R}^{m \times n}$. Capital letters are used to denote matrices (e.g., A, B, Δ) and the subscript ij refers to the (i, j) entry in the matrix. The letters y and t are used to denote scalar parameters, and all other lower-case letters (e.g., x, ξ) are used to indicate vectors, with single subscripts (e.g., x_i) denoting specific elements in the vector. $\|.\|$ will be taken to indicate the euclidean norm, unless stated otherwise.

2.1 The Chebyshev Semi-iterative method

The error vector for the m^{th} iterate from Equation (2.1), $x^{(m)}$, can be written as

$$\epsilon^{(m)} = x^{(m)} - x, \text{ for } m \geq 0,$$

so that

$$\epsilon^{(m)} = M^m \epsilon^{(0)}, \text{ for } m \geq 0.$$

Since M is symmetric, $\|M\| = \rho(M) < 1$, the *average rate of convergence* for m iterations is defined as

$$R(M^m) = \frac{-ln\|M^m\|}{m} \leq R_\infty(M) = -ln\rho(M), \text{ for } m \geq 1. \tag{2.4}$$

From the theory of summability of sequences [Var62], consider the more general iterative procedure

$$y^{(m)} \equiv \sum_{j=0}^{m} \nu_j(m) x^{(j)}.$$

The requirement that if $x^{(0)} = x$, then $y^{(m)}$ must be x, results in the constraint

$$\sum_{j=0}^{m} \nu_j(m) = 1, \text{ for } m \geq 0.$$

The iterative method resulting from the sequence $y^{(m)}$ will be referred to as a *semi-iterative method* and the error vector corresponding to $y^{(m)}$ is given by the expression (see Appendix A.1)

$$\tilde{\epsilon}^{(m)} = y^{(m)} - x = p_m(M)\epsilon^{(0)}, \tag{2.5}$$

where $p_m(t) = \sum_{j=0}^{m} \nu_j(m) t^j$ is an m^{th} degree polynomial with $m \geq 0$ and $p_m(1) = 1$.

A generalization of Equation (2.4) gives the *average rate of convergence* for m iterations of the semi-iterative method

$$R[p_m(M)] \equiv \frac{-ln\|p_m(M)\|}{m}.$$

Note that when $p_m(t) = t^m$, $y^{(m)}$ becomes identical to $x^{(m)}$, and the iterative and semi-iterative methods are equivalent.

In order to accelerate the convergence of the semi-iterative method, it is necessary to minimize the average rate of convergence, or, equivalently, obtain the solution of the minimization problem

$$\min_{p_m(1)=1} \|p_m(M)\|. \tag{2.6}$$

The solution of this problem requires *a priori* determination of the eigenvalues. In its place, consider the new minimization problem

$$\min_{p_m(1)=1} \{ \max_{-1<a\leq t\leq b<1} |p_m(t)| \}$$

where $-\rho(M) \leq a \leq b \leq \rho(M) < 1$. The solution of the new minimization problem is given in terms of the Chebyshev polynomials, $C_m(t)$, defined by

$$C_m(t) = \begin{cases} cos(mcos^{-1}t), & |t| \leq 1, \\ cosh(mcosh^{-1}t), & |t| \geq 1, \end{cases} \tag{2.7}$$

for $m \geq 0$.

Using the trigonometric identity

$$cos[(m-1)\theta] + cos[(m+1)\theta] = 2cos(\theta)cos(m\theta)$$

and Equation (2.7), the following 3-term recurrence can be derived.

$$\begin{gathered} C_0(t) = 1, C_1(t) = t, \\ C_{m+1}(t) = 2tC_m(t) - C_{m-1}(t) \text{ for } m \geq 1. \end{gathered} \tag{2.8}$$

Consider the polynomial $\tilde{p}_m(t)$ defined by

$$\tilde{p}_m(t) = \frac{C_m\left(\frac{2t-(b+a)}{b-a}\right)}{C_m\left(\frac{2-(b+a)}{b-a}\right)}. \tag{2.9}$$

This polynomial $\tilde{p}_m$ is a real polynomial, satisfying $\tilde{p}_m(1) = 1$. Also,

$$\max_{a\leq t\leq b} |\tilde{p}_m(t)| = \frac{1}{C_m\left(\frac{2-(b+a)}{b-a}\right)},$$

since $y = \frac{2t-(b+a)}{b-a}$ is a 1-1 mapping of $a \leq t \leq b$ onto $-1 \leq y \leq 1$. The following theorem can be derived from the properties of $\tilde{p}_m$.

Theorem 2.1 *For each $m \geq 0$, let S_m be the set of all real polynomials $p_m(t)$ of degree m satisfying $p_m(1) = 1$. Then, the polynomial $\tilde{p}_m(t) \in S_m$ is the unique polynomial which solves the minimization problem defined by Equation (2.6).*

Proof: See [Var62].

□

Since $-\rho(M) \leq a \leq b \leq \rho(M) < 1$, if $b \equiv \rho \equiv \rho(M) \equiv -a$ it can be shown (see Appendix A.2) that

$$C_{m+1}\left(\frac{1}{\rho}\right)\tilde{p}_{m+1}(t) = \frac{2t}{\rho}C_m\left(\frac{1}{\rho}\right)\tilde{p}_m(t) - C_{m-1}\left(\frac{1}{\rho}\right)\tilde{p}_{m-1}(t),$$

for $m \geq 1$. Using Equation (2.5) and applying the polynomial $\tilde{p}_m$ to the matrix M, the following recurrence for $\tilde{\epsilon}^{(m)}$ is obtained for $m \geq 1$,

$$C_{m+1}\left(\frac{1}{\rho}\right)(\tilde{\epsilon}^{(m+1)}) = \frac{2M}{\rho}C_m\left(\frac{1}{\rho}\right)(\tilde{\epsilon}^{(m)}) - C_{m-1}\left(\frac{1}{\rho}\right)(\tilde{\epsilon}^{(m-1)}).$$

Since $\tilde{\epsilon}^{(m)} = y^{(m)} - x$, the above recurrence can be rewritten as

$$C_{m+1}\left(\frac{1}{\rho}\right)(y^{(m+1)} - x) = \frac{2M}{\rho}C_m\left(\frac{1}{\rho}\right)(y^{(m)} - x) - C_{m-1}\left(\frac{1}{\rho}\right)(y^{(m-1)} - x) \tag{2.10}$$

for $m \geq 1$. Equations (2.8) and (2.10) can then be combined to yield an iteration of the form

$$y^{(m+1)} = \omega_{m+1}\{My^{(m)} + b - y^{(m-1)}\} + y^{(m-1)}, \tag{2.11}$$

where $\omega_{m+1} = \frac{2C_m(\frac{1}{\rho})}{\rho C_{m+1}(\frac{1}{\rho})}$. The above result specifies the *Chebyshev semi-iterative method* with respect to the iteration defined in Equation (2.1). A brief discussion of the convergence of this method is now presented (for a more detailed discussion see [GV61]).

Since M is a symmetric matrix with eigenvalues μ_i satisfying

$$-\rho(M) \leq \mu_i \leq \rho(M),$$

then

$$\max_{1\leq i\leq n} |\tilde{p}_m(\mu_i)| = \max_{1\leq i\leq n} \frac{|C_m(\mu_i/\rho)|}{C_m(1/\rho)} \text{ for } m \geq 0.$$

Let $|\mu_j| = \rho(M)$ for some j. By the definition of Chebyshev polynomials in Equation (2.8) and $|C_m(\pm 1)| = 1$,

$$\|\tilde{p}_m(M)\| = \frac{1}{C_m(1/\rho)}, m \geq 0.$$

From Equation 2.7, it follows that the above sequence of matrix norms is strictly decreasing for all $m \geq 0$ so that the error $\tilde{\epsilon}^{(m)}$ of Equation (2.5) approaches zero as m becomes large.

The iteration defined by Equation (2.11) can be used in combination with the theory of modified moments (discussed in the next section) to produce approximations to the largest eigenvalues of the matrix M. Specifically, Equation (2.11) may be used to generate iterates $\xi^{(i)}$ through the iteration

$$\xi^{(m+1)} = \omega_{m+1}(M\xi^{(m)} + b) + (1 - \omega_{m+1})\xi^{(m-1)}, \tag{2.12}$$

$$\text{where } \omega_{m+1} = \frac{2C_m(\frac{1}{\rho})}{\rho C_{m+1}(\frac{1}{\rho})}. \tag{2.13}$$

Similar to the Lanczos algorithm (see [Bre80], [GL89]) the next section will show how modified moments derived from the iterates $\xi^{(k)}$ may be used to generate a sequence of bidiagonal matrices whose largest singular values approximate those of the sparse matrix A.

2.2 Orthogonal Polynomials and the Eigenvalue Problem

Some of the definitions relating to the theory of orthogonal polynomials will now be presented, and the relevance of orthogonal polynomials to the general eigenvalue problem $Cx = \lambda x$ will be shown. This theory will then be used to obtain approximations to the eigenvalues of the iteration matrix M defined earlier.

2.2.1 Modified Moments and Orthogonal Polynomials

Definition 2.1 ([Wil62]) *An integrable function λ is called a* **weight function** *on $[a, b]$ if $\lambda(t) \geq 0$ for $t \in [a, b]$ and the moments*

$$\mu_r \equiv \int_a^b t^r \lambda(t) dt \tag{2.14}$$

exist and are finite.

Given a weight function $\lambda(t)$, we can construct a set of polynomials $\pi_k(t)$ such that the following orthogonality property is satisfied.

Definition 2.2 *The set of polynomials $\pi_k(t)$ are said to be orthogonal with respect to the weight function $\lambda(t)$ if and only if*

$$\int_a^b \pi_r(t)\pi_s(t) d\lambda(t) \begin{cases} > 0, & r = s \\ = 0, & r \neq s \end{cases} .$$

Theorem 2.2 *The set of polynomials $\{\pi_0, \pi_1, \ldots, \pi_n\}$ defined in the following way is orthogonal on $[a, b]$ with respect to the weight function λ.*

$$\begin{aligned} &\pi_0(t) \equiv 1, \\ &\pi_1(t) = t - \alpha_1, && \text{for each } t \in [a, b] \end{aligned}$$

where $\alpha_1 = \dfrac{\int_a^b t d\lambda(t)}{\int_a^b d\lambda(t)}$;

and when $k \geq 2$,

$$\pi_k(t) = (t - \alpha_k)\pi_{k-1}(t) - \gamma_k \pi_{k-2}(t), \text{ for each } t \in [a, b], \tag{2.15}$$

where $\alpha_k = \dfrac{\int_a^b t[\pi_{k-1}(t)]^2 d\lambda(t)}{\int_a^b [\pi_{k-1}(t)]^2 d\lambda(t)}$

and $\gamma_k = \dfrac{\int_a^b t\pi_{k-1}(t)\pi_{k-2}(t) d\lambda(t)}{\int_a^b [\pi_{k-2}(t)]^2 d\lambda(t)}$

Proof: See [BF81].

□

Given a weight function $\lambda(t)$ represented in terms of the moments μ_r defined in Equation (2.14) [Gau82] describes a procedure for the recursive computation of α_k and γ_k which is numerically unstable. A more stable procedure may be obtained if $\lambda(t)$ is codified in terms of the *modified moments*

$$\nu_r = \int p_r(t) d\lambda(t), \tag{2.16}$$

where $p_r(t)$ is a set of orthogonal polynomials satisfying a recurrence relation

$$tp_j(t) = b_j p_{j+1}(t) + a_j p_j(t) + c_j p_{j-1}(t). \tag{2.17}$$

It has been shown in [GK89] that some simplifications to this more stable procedure are possible when the polynomials $p_k(t)$ are chosen to be the Chebyshev polynomials. The simplified procedure is described in Section 2.4 as part of the CSI-MSVD algorithm.

2.2.2 Relation to the Eigenvalue Problem

The connection between the moments and eigenvalues is well-known (e.g., [Lan50], [Hou64], [Gol74]). The associated matrix equation for the set of polynomials $\pi_k(t)$ orthogonal to the modified moments defined by Equation (2.16) has the form

$$\begin{bmatrix} \alpha_0 & 1 & & & \\ \gamma_1 & \alpha_1 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{k-1} & \alpha_{k-1} & 1 \\ & & & \gamma_k & \alpha_k \end{bmatrix} \begin{bmatrix} \pi_0(t) \\ \pi_1(t) \\ \vdots \\ \pi_k(t) \end{bmatrix} = t \begin{bmatrix} \pi_0(t) \\ \pi_1(t) \\ \vdots \\ \pi_k(t) \end{bmatrix} - e_{k+1}\pi_{k+1}(t). \tag{2.18}$$

The tridiagonal matrix $J_k = [\gamma_k, \alpha_k, 1]$ of coefficients defined in the above equation is called the **Jacobi matrix**. From Equation (2.18) it can be inferred that the zeros of the polynomial $\pi_{k+1}(t)$ may be found by solving the standard eigenvalue problem $J_k x = tx$. Thus, the roots of $\pi_{k+1}(t)$ may be obtained as the eigenvalues of the Jacobi matrix J_k.

As pointed out in [GK89], this procedure is analogous to the Lanczos algorithm and may be used to approximate the eigenvalues of the iteration matrix M in the Chebyshev semi-iterative method. A scheme to extract modified moments based on the theory described in [GK89] from the Chebyshev iterates will be described below. The modified moments will then be used to define a Jacobi matrix whose eigenvalues approximate those of the iteration matrix.

2.3 Modified Moments from the Chebyshev Semi-iterative Method

Since the iteration matrix M defined defined by the canonical eigenvalue problems in Lemma 1.1 is symmetric, M has a complete set of orthogonal eigenvectors which form a basis for $\mathbb{R}^n$. Let this set be denoted by

$$Q = [q_1, q_2, \ldots, q_n].$$

Then,

$$\xi^{(0)} = \sum_{i=1}^{r} \alpha_i q_i,$$
$$\xi^{(k)} = \sum_{i} \alpha_i \pi_k(\lambda_i) q_i$$

where, for $i > 0$, $\xi^{(i)}$ is the i^{th} Chebyshev iterate generated by Equation (2.12), $\xi^{(0)}$ is the initial iterate, and λ_i is an eigenvalue of M corresponding to the eigenvector q_i. Consider the inner product of the k^{th} and i^{th} iterates, i.e.,

$$\langle \xi^{(k)}, \xi^{(l)} \rangle = \sum_{i=1}^{r} {\alpha_i}^2 \pi_k(\lambda_i) \pi_l(\lambda_i). \tag{2.19}$$

Equation (2.19) is equivalent to the continuous integral

$$\langle \xi^{(k)}, \xi^{(l)} \rangle = \int \pi_k(\lambda) \pi_l(\lambda) d\alpha(\lambda), \tag{2.20}$$

when $\alpha(\lambda)$ is defined to be the discrete non-negative distribution [GK89]

$$\alpha(\lambda) = \begin{cases} 0, & \text{if } \lambda \leq \lambda_1, \\ {\alpha_1}^2 + {\alpha_2}^2 + \ldots + {\alpha_t}^2, & \text{if } \lambda_t < \lambda \leq \lambda_{t+1}, \\ {\alpha_1}^2 + {\alpha_2}^2 + \ldots + {\alpha_n}^2, & \text{if } \lambda_n < \lambda. \end{cases}$$

The discrete distribution $\alpha(\lambda)$ is illustrated in Figure 2.1. By choosing $l = 0$ in Equation (2.20) and noting that $\pi_0(t) = 1$ from Theorem 2.2, it follows that

$$\langle \xi^{(k)}, \xi^{(0)} \rangle = \int \pi_k(\lambda) d\alpha(\lambda).$$

Note that the *final* orthogonal polynomial $\pi_n(t)$ has a zero at each eigenvalue i.e., $\pi_n(\lambda_i) = 0, i = 1, 2, \ldots r$. Hence, at each step of the Chebyshev semi-iterative method, we can extract the k^{th} modified moment

$$\langle \xi^{(k)}, \xi^{(0)} \rangle = \nu_k. \tag{2.21}$$

The extraction of moments from iterates can be accelerated by using the recurrence relations for the Chebyshev polynomials defined in Equation (2.8). It can be shown (see Appendix A.3) that

$$\nu_{2k} = \langle \xi^{(k)}, \xi^{(k)} \rangle + \frac{1}{C_{2k}(\frac{1}{\rho})} \{ \langle \xi^{(k)}, \xi^{(k)} \rangle - \nu_0 \}, \text{ and} \tag{2.22}$$

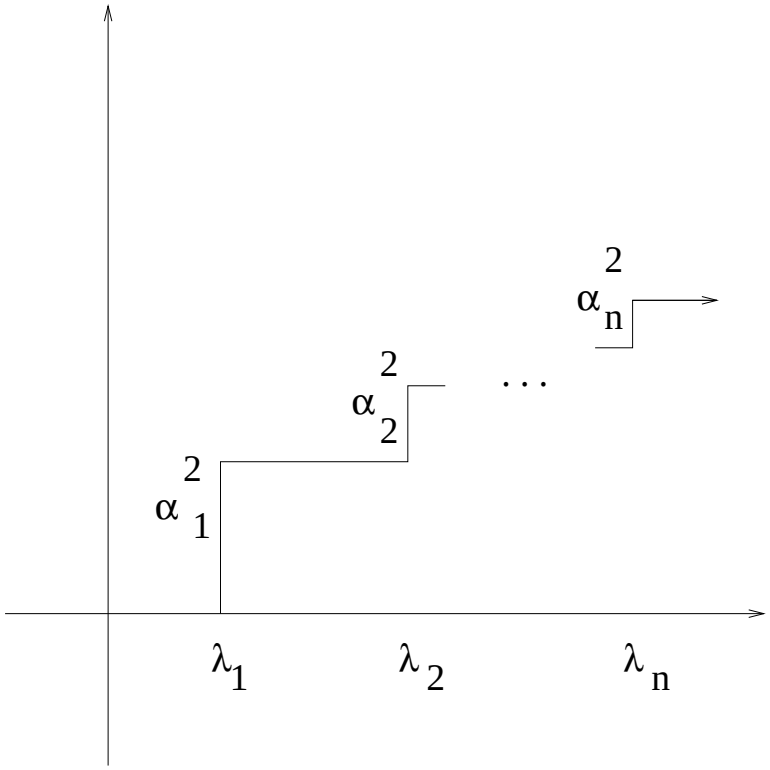


Figure 2.1. The discrete distribution $\alpha(\lambda)$

$$\nu_{2k+1} = \langle \xi^{(k)}, \xi^{(k+1)} \rangle + \frac{1}{\rho C'_{2k+1}(\frac{1}{\rho})} \{ \langle \xi^{(k)}, \xi^{(k+1)} \rangle - \langle \xi^{(0)}, \xi^{(1)} \rangle \}. \tag{2.23}$$

Note that the polynomial $p_r(t)$ in Equation (2.16) associated with the modified moments ν_{2k} and ν_{2k+1} satisfies

$$p_{k+1}(t) = \omega_{k+1} t p_k(t) + (1 - \omega_{k+1}) p_{k-1}(t),$$

where the coefficients a_k, b_k, c_k of the polynomials $p_r(t)$ in Equation (2.17) are given by

$$a_k = 0, b_k = \frac{1}{\omega_{k+1}}, c_k = \frac{\omega_{k+1} - 1}{\omega_{k+1}}. \tag{2.24}$$

2.4 The CSI-MSVD Algorithm

Let $\lambda(t)$ be a weight function codified in terms of the $2n$ modified moments ν_r as defined in Equations (2.22) and (2.23). A procedure to compute the coefficients of polynomials $\pi_k(t)$ orthogonal with respect to $\lambda(t)$ is desired. From Equation (2.15), the polynomials

$\pi_k(t)$ are of the form

$$\pi_{k+1}(t) = (t - \alpha_k)\pi_k(t) - \gamma_k \pi_{k-1}(t).$$

Following [GK89] with the choice $\beta_k = 1$, the coefficients α_k and γ_k may be determined using the recurrences below.

For $k = 1, 2, \ldots, m-1$,
  For $l = k, k+1, \ldots, 2m-k-1$,

$$\sigma_{kl} \equiv b_l\sigma_{k-1,l+1} - (\alpha_{k-1} - a_l)\sigma_{k-1,l} - \gamma_{k-1}\sigma_{k-2,l} + c_l\sigma_{k-1,l-1}, \tag{2.25}$$

$$\alpha_k = a_k + \frac{\sigma_{k,k+1}}{\sigma_{kk}} - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}},$$

$$\gamma_k = \frac{\sigma_{kk}}{\sigma_{k-1,k-1}}. \tag{2.26}$$

Here, a_k, b_k and c_k are defined by Equation (2.24), and initially,

$$\sigma_{-1,l} = 0, \sigma_{0,l} = \nu_l, \alpha_0 = \nu_1/\nu_0, \gamma_0 = 0.$$

The computation of α_k's and γ_k's , $k = 1, 2, \ldots$ effectively constructs the elements of the Jacobi matrix from Equation (2.18), whose eigenvalues approximate those of the iteration matrix M. This procedure will be referred to as the CSI-MSVD algorithm.

Thus, by setting M to either of the two canonical matrices described in Lemma 1.1, one can obtain the SVD of a general matrix by solving an equivalent symmetric eigenvalue problem. The implementation of this scheme for each of the canonical eigenvalue problems described in Lemma 1.1 will now be examined.

2.4.1 Two-Cyclic CSI-MSVD

Let M be in the form of a weakly cyclic matrix of index 2, defined in Equation (1.4), and partition the Chebyshev iterate $\xi^{(i)}$ into the $m \times 1$ vector x_1 and the $n \times 1$ vector x_2 and the vector b from Equations (2.2) into the $m \times 1$ vector b_1 and the $n \times 1$ vector b_2 so that

$$\xi^{(i)} = \begin{pmatrix} {x_1}^{(i)} \\ {x_2}^{(i)} \end{pmatrix}, \; b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Equation (2.12) can be re-written as

$${x_1}^{(m+1)} = \omega_{m+1}\{A{x_2}^{(m)} + b_1 - {x_1}^{(m-1)}\} + {x_1}^{(m-1)}, \tag{2.27}$$

$${x_2}^{(m+1)} = \omega_{m+1}\{A^T{x_1}^{(m)} + b_2 - {x_2}^{(m-1)}\} + {x_2}^{(m-1)}. \tag{2.28}$$

The elements of successive iterates generated by Equations (2.27) and (2.28) are related

$$\begin{bmatrix} x_1{}^{(0)} \\ x_2{}^{(0)} \end{bmatrix} \searrow \begin{bmatrix} x_1{}^{(1)} \\ x_2{}^{(1)} \end{bmatrix} \nearrow \begin{bmatrix} x_1{}^{(2)} \\ x_2{}^{(2)} \end{bmatrix} \searrow \begin{bmatrix} x_1{}^{(3)} \\ x_2{}^{(3)} \end{bmatrix}$$

Figure 2.2. Updating the Chebyshev iterate

by the dependency shown in Figure (2.2), so that choosing $x_2{}^{(0)} = 0$ when $b_1 = b_2 = 0$ forces $x_1{}^{(1)} = x_2{}^{(2)} = \ldots = 0$. If $\xi^{(i)}$ denotes the i^{th} iterate, then

$$\xi^{(2s)} = \begin{pmatrix} x_1{}^{(2s)} \\ 0 \end{pmatrix}, \text{ and } \quad \xi^{(2s-1)} = \begin{pmatrix} 0 \\ x_2{}^{(2s-1)} \end{pmatrix}. \tag{2.29}$$

Thus, at each step, only one of the Equations (2.27) and (2.28) needs to be computed, reducing the number of computations by a factor of two. Also, this new iteration requires only one initial vector approximation to $x_1{}^{(0)}$ as opposed to the two approximations ($x_1{}^{(0)}$ and $x_2{}^{(0)}$) required for the general case.

The simplifications provided by Equation (2.29) also reduce the number of computations involved in the calculation of the coefficients of the Jacobi matrix in Equation (2.18). From Equations (2.29) and (2.23), it follows that $\nu_{2k+1} = 0$, with $\nu_1 = 0$, so that $\alpha_0 = 0$. It can be shown by induction that

$$\sigma_{k,k+2i-1} = 0, i = 1, 2, \ldots, m-k, \tag{2.30}$$

which forces $\alpha_k = 0$ for all k. Hence, the only unknown quantities in the Jacobi matrix are the elements of the sub-diagonal, γ_k.

As demonstrated in [GLO81] the eigenvalues of the $2k \times 2k$ Jacobi matrix are the same as the singular values of the matrix

$$B_k = \begin{bmatrix} \sqrt{\gamma}_1 & \sqrt{\gamma}_2 & & & \\ & \sqrt{\gamma}_3 & \sqrt{\gamma}_4 & & \\ & & \ddots & \ddots & \\ & & & \sqrt{\gamma}_{2k-3} & \sqrt{\gamma}_{2k-2} \\ & & & & \sqrt{\gamma}_{2k-1} \end{bmatrix}. \tag{2.31}$$

The QR-iteration for bidiagonal matrices [DBMS79] may be applied to the matrix B_k to obtain its singular values, thus approximating the singular values of the original $m \times n$ matrix A.

2.4.2 CSI-MSVD Applied to the Matrix A^TA

The SVD of the $m \times n$ matrix A may be obtained from the eigenvalue decomposition of the matrix A^TA. When $m \gg n$, the matrix A^TA is considerably smaller than the corresponding two-cyclic matrix C of Equation (1.4). The CSI-MSVD algorithm could be applied to approximate the eigenvalues and eigenvectors of A^TA. However, the

simplifications possible for the two-cyclic matrix may not be applied to this case. For example, it is not possible to re-write Equation (2.12) by partitioning the iterates to obtain two dependent iterations as was done to obtain Equations (2.27) and (2.28). The disadvantages arising from iterates $\xi^{(i)}$ of this form are listed below.

- It is not possible to halve the number of operations at each step, as in the two-cyclic case. Each element in the iterates needs to be computed.
- The odd-moments are non-zero, and both odd and even moments exist.
- The simplification provided by Equation (2.30) does not apply.
- The coefficients α_k are not nonzero, in general, and it is necessary to compute both diagonal and sub-diagonal elements in the Jacobi matrix J_k.
- The bidiagonal-QR iteration cannot be used. The eigenvalues of the tridiagonal Jacobi matrix must be computed through less efficient algorithms such as the QL method [Par80].

The discussions in Sections 2.4.1 and 2.4.2 suggest that the CSI-MSVD algorithm with a 2-cyclic iteration matrix has the advantages of faster convergence to the singular values, with fewer intermediate computations. For this reason, this dissertation focuses on the 2-cyclic eigenvalue problem. Unless otherwise stated, references to CSI-MSVD in this dissertation will pertain to the algorithm as applied to 2-cyclic iteration matrices as defined in Equation (1.4).

The three main steps that constitute the CSI-MSVD algorithm are:

1. calculation of the CSI-iterate using Equations (2.27) and (2.28),
2. calculation of the new moments for the current iterate, and
3. updating the bidiagonal matrix and approximating the eigenvalues of the two-cyclic iteration matrix through the QR-iteration.

Figure 2.3 shows the dependencies involved in the steps of the above outlined procedure. The pipelined nature of the computation indicates that Steps 1, 2, and 3 described could be carried out concurrently. For example, the computation of the anti-diagonal elements $\sigma_{13}, \sigma_{22}, \sigma_{15}, \sigma_{24}, \sigma_{33}$ (shown in the box labeled SIGMA in Figure 2.3) could be overlapped with the computation of the iterates $\xi^{(4)}$ and $\xi^{(5)}$. Also, when the bidiagonal matrix has been updated with the elements γ_2 and γ_3 (the box labeled GAMMA) by using Equation (2.26), the approximation of eigenvalues through the bidiagonal-QR iteration could be done in parallel with the computation of the next anti-diagonal elements (σ_{kl}, $k + l = 8$ or $k + l = 10$). Thus, the three functional components MATVEC, SIGMA, and GAMMA could be executed on three different processors. Information about the two-cyclic matrix M of Equation (1.4) is required only for the computations in MATVEC, so that the implementations of SIGMA and GAMMA is independent of the format used for storing the matrix.

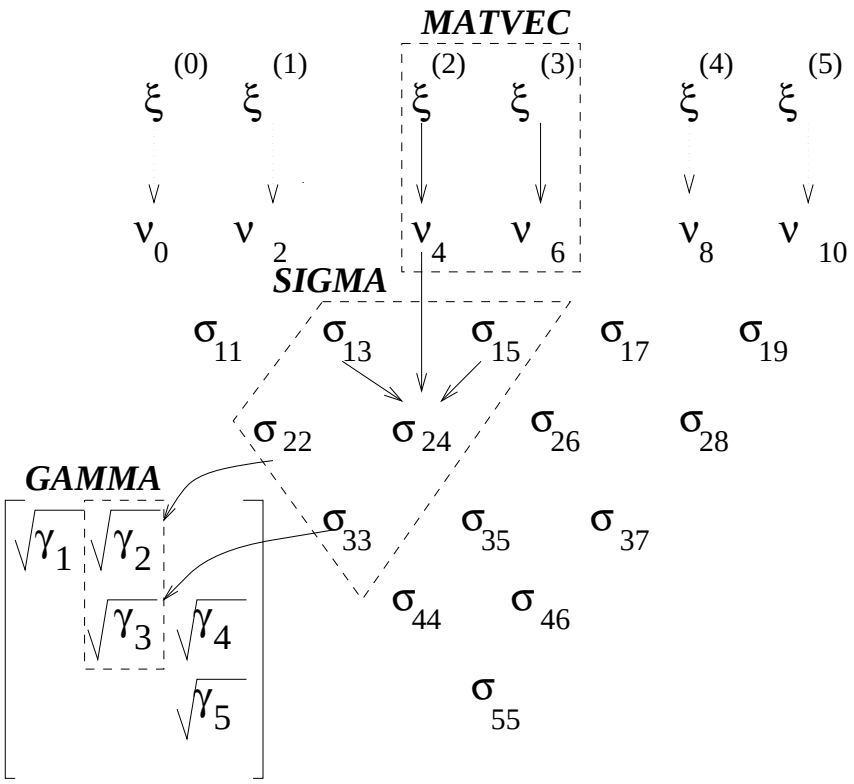


Figure 2.3. A single pass of two-cyclic CSI-MSVD

2.5 Singular Vector Calculation

Although [BG91] describes an implementation to approximate singular values accurately and efficiently using modified moments from Chebyshev iterates, singular vectors are not estimated. Two problems which arise from the lack of singular vector approximations are listed below.

1. Estimation of the error in approximated singular values necessitates a comparison with *true* singular values generated by some other scheme, in order to detect if the current approximation to the singular value is within some acceptable bound.

2. A set of singular values of acceptable accuracy must not be recomputed in succeeding iterations to prevent wasteful computation. Also, since CSI-MSVD approximates the largest singular values first, it is necessary to deflate the converged singular values out of the computation to allow convergence of the smaller singular value approximations. Some candidate schemes for deflation are discussed in [Saa92]. Most of these schemes require singular vector approximations.

The implementation described in [BG91] addresses these two issues in the following manner.

- **Convergence Tests:** At the k^{th} step of the Chebyshev iteration, the quantity
$$|\tilde{\sigma}_i^{(k)} - \tilde{\sigma}_i^{(k-1)}|/\tilde{\sigma}_i^{(k)}$$
is computed, where $\tilde{\sigma}_i^{(k)}$ is an approximation to the i^{th} largest singular value of A. The procedure terminates when this quantity is within some desired tolerance ϵ_{tol}, or when k exceeds the user-specified input of k_{max}, the maximum number of iterations allowed.

- **Deflation:** No deflation schemes are attempted.

The estimation of singular vectors from the eigenvectors derived by CSI-MSVD is a non-trivial task. The tridiagonal Jacobi matrix J_k defined by Equation (2.18) has been constructed so that characteristic equation corresponding to J_k has the same roots as the iteration matrix. Thus, the method does not explicitly construct a basis for the subspace spanned by the eigenvectors, so that the methods used in Lanczos or other popular Krylov-subspace methods do not have obvious analogs in CSI-MSVD.

However, as discussed in Section 2.1, especially when parameters such as ω_i defined in Equation (2.11) are chosen optimally, the iterative method defined by Equation (2.12) should converge to ξ such that $(I - M)\xi = b$. As $b \to 0$, the eigenvector ξ of the appropriately scaled matrix M corresponds to the eigenvalue nearest 1. The parameters that affect the convergence of CSI-MSVD are

- the scaling factor δ, chosen so that the $m \times n$ matrix $(1/\delta)A$ has singular values $\sigma_i < 1, i = 1, .., r, r \leq n$ (or equivalently, the two-cyclic matrix M has eigenvalues $\lambda_i = \sigma_i < 1$). Ideally, since the eigenvector is the solution of a system of the form $(I - M)x = b$, an optimal choice for the case $b = 0$ would be $\delta = \sigma_{max}(A)$. Here, M is one of the canonical matrices defined in Lemma 1.1.

- the damping-parameter $\mu \equiv \rho$ for the Chebyshev polynomials defined in Equations (2.22) and (2.23). As discussed in [BG91], setting $\mu = \bar{\mu}$ effectively suppresses all singular values σ_i having magnitude less than $\bar{\mu}$. It is therefore desirable to set $\mu \approx \sigma_2/\sigma_1$ in order to accelerate convergence to the largest singular value.

The convergence of the iterative method to the eigenvector ξ is affected by the choice of the above parameters, which are not known *a priori*. Also, after k steps of the iterative method, the recurrences provided by Equation (2.23) and (2.22) allow the calculation of $2k$ moments from the Chebyshev iterates, so that the construction of the Jacobi matrix is accelerated. However, in the absence of any obvious method to approximate Chebyshev iterates from the Jacobi matrix, the convergence of the iterative method to the eigenvector is slower than the convergence of the Jacobi matrix to the eigenvalues. To overcome these problems, a two-pass scheme has been developed. Figure 2.4 illustrates the following steps of the two-pass scheme.

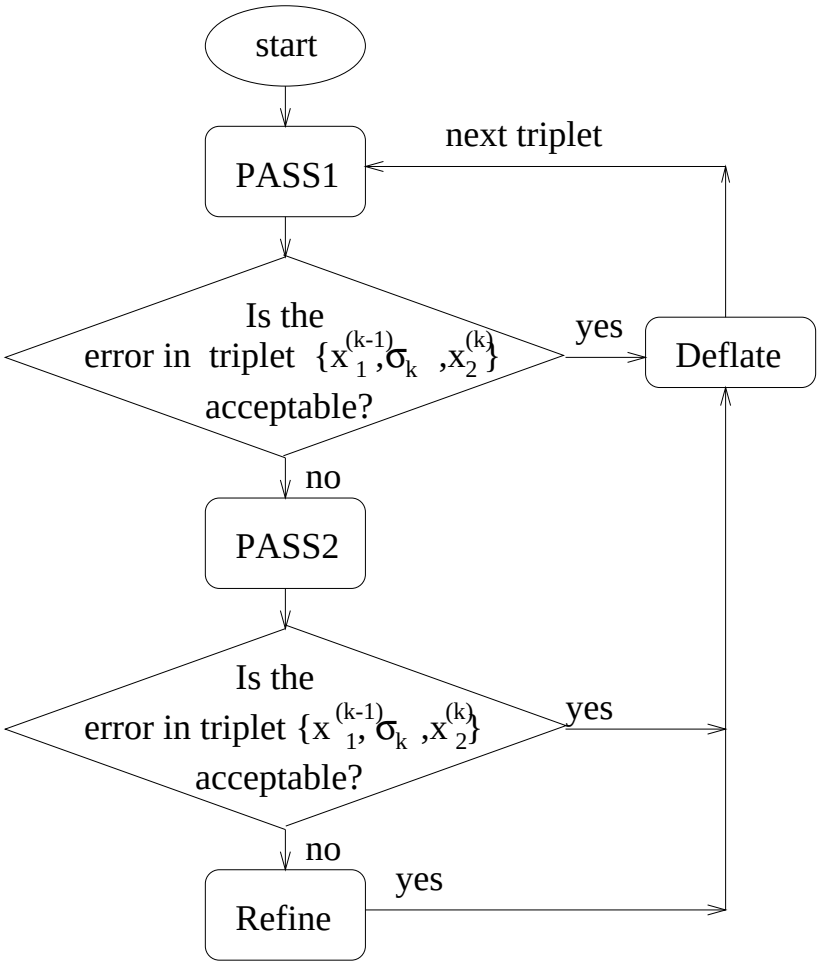


Figure 2.4. Flow chart for computing the k-largest singular triplets using CSI-MSVD

1. PASS1: In the first pass, perform at most η_1 steps of CSI-MSVD. This involves the execution of all of the steps shown in Figure 2.3. The iteration is terminated when either
   - the convergence tests defined in [BG91] are satisfied, or,
   - exactly η_1 iterates have been calculated.
2. PASS2: Let $\tilde{\lambda}_1 \geq \tilde{\lambda}_2 \geq \ldots \geq \tilde{\lambda}_k$ be the eigenvalues of J_k at the end of PASS1, and $\xi^{(k)}$ the current Chebyshev iterate. If the residual norm $\|M\xi^{(k)} - \tilde{\lambda}_1\xi^{(k)}\|$ is not within some desired tolerance ϵ_{tol}, the scaling parameter δ should be set to $\tilde{\lambda}_1$, and the damping parameter μ set to $\tilde{\lambda}_2/\tilde{\lambda}_1$. Then, at most η_2 steps of the Chebyshev iteration are performed while examining the convergence as in Step 1.
3. ACCEPT: Accept the approximate singular value corresponding to $\tilde{\lambda}_1$ (as defined in Lemma 1.1). If a higher accuracy in the current approximations to the singular value and corresponding singular vector obtained from $(\tilde{\lambda}_1, \xi^{(k)})$ is desired, some refinement procedure may be used, with starting values set to the current estimates. In practice, the accelerated construction of the Jacobi matrix produces eigenvalue approximations of at least 10^{-3} accuracy, and as discussed in [Par80] a single step of inverse-iteration could be used to approximate the eigenvectors to 10^{-6} accuracy. However, as described in Section 1, for large, sparse matrices it is desirable to avoid fill-in from direct methods. The current PVM implementation of CSI-MSVD uses an ANSI-C translation of the subroutine SYMMLQ [PS75] [B$^+$94] for refinement of the eigenvector approximation. SYMMLQ is a Conjugate Gradient method for symmetric indefinite systems of the form $(B - \tau I)x = b$ where τ is a specified scalar value. By setting $b = 0$, τ to $\tilde{\lambda}_1$, the computed vector x may approximate an eigenvector of the matrix B. After the residual error has been reduced to the desired tolerance, deflation in the form of a Wielandt scheme [Saa92] can be employed to repeat the above 3 steps in order to approximate the next triplet.

2.6 Estimation of Error in Singular Triplet

As the Chebyshev semi-iterative method proceeds, iterates of the form

$$\begin{bmatrix} {x_1}^{(k)} \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ {x_2}^{(k+1)} \end{bmatrix}, \begin{bmatrix} {x_1}^{(k+2)} \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ {x_2}^{(k+3)} \end{bmatrix}, \ldots$$

are generated. Approximations to the left singular vector corresponding to the largest singular value are obtained from $\lim_{l\to\infty} {x_1}^{(k+2l)}$ and the corresponding right singular vector is obtained from $\lim_{l\to\infty} {x_2}^{(k+2l+1)}$. An estimate of the error in these singular vectors is desired. Let

$$\nu_1 = A{x_2}^{(k+1)}, \tilde{\nu_1} = \nu_1/\|\nu_1\|, \nu_2 = A^T{x_1}^{(k)}, \tilde{\nu_2} = \nu_2/\|\nu_2\|, \tag{2.32}$$

and consider the error vectors defined by

$$\epsilon_1 = \tilde{\nu_1} - {x_1}^{(k)} \text{ and } \epsilon_2 = \tilde{\nu_2} - {x_2}^{(k+1)}. \tag{2.33}$$

These vectors measure the error in the *singular vector approximations alone*, and do not reflect the error in the *singular triplet*, i.e., $\nu_1 - \sigma_1 {x_1}^{(k)}$ and $\nu_2 - \sigma_1 {x_2}^{(k+1)}$. The vectors ϵ_1 and ϵ_2 are generated as part of the solution of the system of equations defined by Equation (2.2) with $b = 0$ and σ_1 is approximating the singular value of A (now scaled by δ) closest to 1.

The vectors ${x_1}^{(k+2)}$ and ${x_2}^{(k+3)}$ are generated by substituting $m = k + 1$ in Equation (2.27) and $m = k + 2$ in Equation (2.28) (with $b_1 = b_2 = 0$) to get

$${x_1}^{(k+2)} = \omega_{k+2}\{A{x_2}^{(k+1)} - {x_1}^{(k)}\} + {x_1}^{(k)}, \tag{2.34}$$

$${x_2}^{(k+3)} = \omega_{k+3}\{A^T{x_1}^{(k+2)} - {x_2}^{(k+1)}\} + {x_2}^{(k+1)}. \tag{2.35}$$

Hence, the quantity $A{x_2}^{(k+1)}$ is calculated as an intermediate result in the calculation of ${x_1}^{(k+2)}$, and ν_1 can be calculated at step $k + 2$.

An analogous result for ν_2 is harder to derive since the right multiplication of A^T is by ${x_1}^{(k+2)}$ which is calculated in Equation (2.34) (rather than ${x_1}^{(k)}$). Consider the intermediate product

$$\begin{aligned}
\nu_a &= A^T * \{\omega_{k+2}\{A{x_2}^{(k+1)} - {x_1}^{(k)}\} + {x_1}^{(k)}\} \\
&= A^T\{(1 - \omega_{k+2}){x_1}^{(k)} + \omega_{k+2}A{x_2}^{(k+1)}\} \\
&= (1 - \omega_{k+2})A^T{x_1}^{(k)} + \omega_{k+2}A^TA{x_2}^{(k+1)}.
\end{aligned}$$

From Equation (2.32), $\nu_1 = A{x_2}^{(k+1)}$ and so

$$\nu_a = (1 - \omega_{k+2})A^T{x_1}^{(k)} + \omega_{k+2}A^T\nu_1. \tag{2.36}$$

Since $\epsilon_1 = \frac{\nu_1}{\|\nu_1\|} - {x_1}^{(k)}$ by definition, it follows that $\nu_1 = \|\nu_1\|(\epsilon_1 + {x_1}^{(k)})$. Substituting this expression for ν_1 into Equation (2.36) yields

$$\nu_a = (1 - \omega_{k+2})A^T{x_1}^{(k)} + \|\nu_1\|\omega_{k+2}A^T(\epsilon_1 + {x_1}^{(k)}) \text{ or}$$

$$\nu_a = \kappa_1 A^T{x_1}^{(k)} + \kappa_2 A^T\epsilon_1, \tag{2.37}$$

where $\kappa_1 = 1 + \omega_{k+2}(\|\nu_1\| - 1)$, $\kappa_2 = \omega_{k+2}\|\nu_1\|$, and hence ν_a is a perturbation of a vector in the desired direction $A^T{x_1}^{(k)}$. If this perturbation is suitably small, i.e. $\kappa_2 A^T\epsilon_1 \to 0$, then ϵ_2 could be approximated by

$$\tilde{\epsilon}_2 = \tilde{\nu_a} - {x_2}^{(k+1)}, \tag{2.38}$$

where $\tilde{\nu_a}$ is the normalized version of ν_a. In practice, for typical applications such as information retrieval, the larger singular values σ_i are typically well-separated, and the parameter $\mu = \sigma_2/\sigma_1$ has values in the range $(0.5, 0.8)$ so that, (see Figure 2.5) $\omega_i \approx 1$

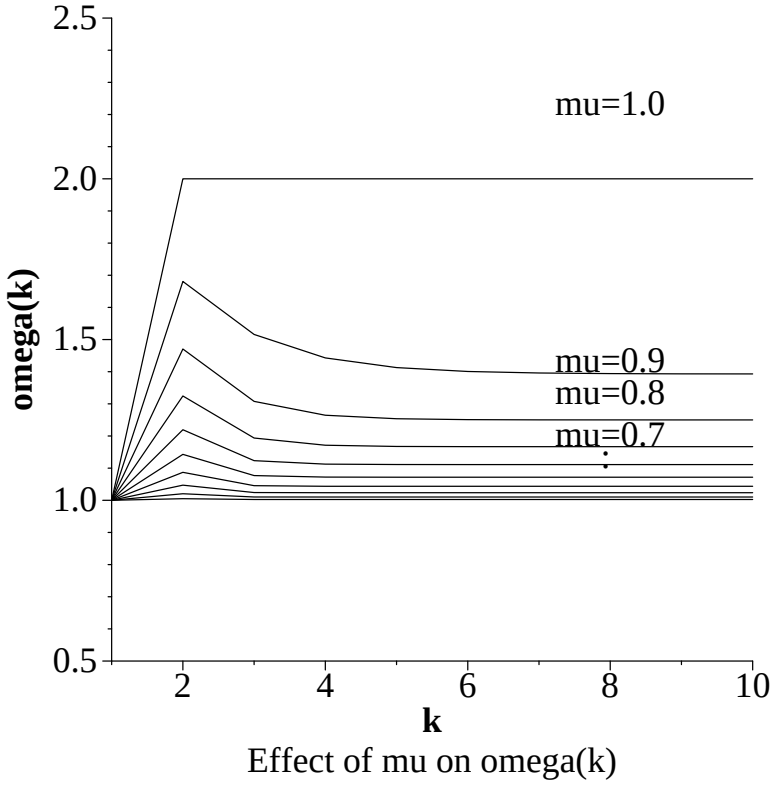


Figure 2.5. Effect of parameter μ

forcing $\kappa_2 \approx 1$. The perturbation ν_a is only as large as ϵ_1, allowing an approximation of ϵ_2 through Equation (2.38). However, when clustered singular values are encountered, the effect of an increased perturbation combined with accumulated round-off render this error estimate unreliable. In this situation, the CSI-MSVD algorithm uses an external refinement scheme such as SYMMLQ to obtain a more accurate estimate of the error in the singular triplet.

A pseudo-code for the two-pass CSI-MSVD algorithm to compute both eigenvalues and eigenvectors with implicit error estimation is listed in Figures 2.6 and 2.7.

Function PASS

Input Starting vector x, scaling parameter δ, damping parameter μ, matrix A, upper bound on number of iterations t_{max}, error-bound on approximated solution tol

Output Approximations to the 3-largest eigenvalues of the 2-cyclic matrix defined in Equation (1.4) λ_1, λ_2 and λ_3, and eigenvector approximation $\left(x^T, y^T\right)^T$.

Compute $\omega(1 : 4t_{max} + 1)$ using Equation (2.13). Initialize λ_1 to δ.

Scale A by δ.

/ This may be implicitly incorporated in the matrix-vector multiplication routine */*

Set $y = A^T x$; Compute $\sigma_{13} \leftarrow \nu_2$ using Equation (2.22).

Compute σ_{22} using Equation (2.25) and set γ_1 to σ_{22}.

for $t = 1$ to $t_{max} - 1$

Save current values of x, y and λ_1 in x_s, y_s and λ_s, respectively.

Compute new iterate x using Equation (2.34), and y using Equation (2.35).

Calculate new moments $\sigma_{1,4t+1}$ and $\sigma_{1,4t+3}$ from x and y using Equations (2.22) and (2.23), respectively.

Using Equation (2.25), compute the antidiagonals $\sigma_{1,4t+1}, \sigma_{2,4t}, \ldots, \sigma_{2t,2t}$ and $\sigma_{1,4t+2}, \sigma_{2,4t+1}, \ldots, \sigma_{2t+1,2t+1}$.

Using Equation (2.26), compute the values γ_{2t} and γ_{2t+1}, and use the bidiagonal QR [DBMS79] iteration to approximate the eigenvalues $\lambda_1, \lambda_2, \lambda_3$ of the updated bidiagonal matrix.

Compute the error estimate $\epsilon = \left\| \begin{pmatrix} \hat{x_s} \\ \hat{y_s} \end{pmatrix} - \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} \right\|$, where $\hat{v}$ denotes the normalized form of the vector v.

If $\|\lambda_1 - \lambda_s\| < tol$, return to the calling program.

endfor

Figure 2.6. Pseudo-code for one PASS of the CSI-MSVD algorithm.

Input $m \times n$ matrix A, estimate of damping parameter μ, estimate of scaling parameter δ, bound on error in computed singular triplet tol, number of desired triplets N.
Output N singular triplets.

Initialize $\mu = \mu_e st$, $startv = rand(m, 1)$.
for $i = 1$ to N
  Call Function **PASS** to obtain singular value approximations λ_1, λ_2 and λ_3, singular triplet $\{x, \lambda_1, y\}$ and error estimate ϵ.
  if $\epsilon < tol$,
    set $A \leftarrow A - \lambda_1 x y^T$.
    goto NEXT.
  else
    Set $\mu = \lambda_2/\lambda_1$, $\delta = \lambda_1$, $startv = x$.
    Call Function **PASS** to obtain singular value approximations λ_1, λ_2 and λ_3, singular triplet $\{x, \lambda_1, y\}$ and error estimate ϵ.
    if $\epsilon < tol$,
      set $A \leftarrow A - \lambda_1 x y^T$.
      goto NEXT.
    else
      Call Function SYMMLQ [PS75] with λ_1 as $shift$, and $\left(x^T, y^T\right)^T$ as starting vector.
    endif
  endif
NEXT: Set $\mu = \lambda_3/\lambda_2, \delta = \lambda_2$.
endfor

Figure 2.7. Pseudo-code for two-pass CSI-MSVD algorithm with implicit error estimation.

Chapter 3

Sparse Matrix Applications

Sparse linear least squares problems naturally arise in many real-world applications in the physical and social sciences. The use of the sparse SVD to solve such problems is of current interest to researchers in these fields [WBSM94], [DDF+90]. A brief description of two such applications, query-based information retrieval and seismic reflection tomography, is provided in this chapter. A description of typical sparse-matrix storage formats used, followed by a discussion of the effect of these storage formats on the performance of iterative methods is also discussed. Parallel implementations of the time-consuming kernels for iterative methods using compressed storage formats for the input sparse matrices are also described in this Chapter.

3.1 Applications for Sparse Singular Value Decomposition

CSI-MSVD was tested with matrices arising from query-based information retrieval applications and seismic reflection tomography. This section describes these two applications, along with the role played by the SVD in each problem domain.

3.1.1 Latent Semantic Indexing

The fundamental goal of information retrieval techniques is to match words of queries provided by a user with words of documents in the database being searched, and thereby extract relevant documents. Attempts to solve this problem by a literal match between words in queries and documents are not always successful because users want to retrieve documents on the basis of conceptual topic or meaning. There are two main sources of noise arising from variability in word usage:

1. many possible in words to express the same concept (*synonymy*), or,
2. multiplicity in meanings of some words (*polysemy*).

Latent Semantic Indexing (LSI) [DDF$^+$90] proposes a solution to this problem by assuming that there is an underlying semantic structure in word usage. Here, the frequency of appearance of terms that could be used as referents to a document is used to set up an $m \times n$ matrix A, whose m rows and n columns correspond to the terms and documents, respectively. Each element $[a_{ij}]$ of the term-document matrix A is a measure of the frequency of appearance of term i in document j. The matrix A is naturally sparse since there are relatively few referent terms for any given document. The closest (in a least squares sense) rank-k approximation to the term-document matrix

$$A_k = \sum_{i=1}^{k} u_i.\sigma_k.{v_i}^T \text{ with } k < r \tag{3.1}$$

is sought so that A_k captures the major associational structure in the matrix and removes the noise. The rank-property of the SVD [GR71] allows the computation of A_k from the matrix A. The model provided by Equation (3.1), usually with $100 \leq k \leq 200$, encodes documents in a reduced space $\mathcal{R}(A_k)$ using the left- and right-singular vectors u_i and v_i. Using A_k as an approximation to the original matrix A allows conceptually-related documents with different referent terms to be mapped into the same vector, ameliorating the effects of synonymy. This clustering of conceptually-related documents in $\mathcal{R}(A_k)$ also causes documents to be described by a consensus of their term meanings, dampening the effects of polysemy. A discussion of the properties and performance of LSI using the sparse SVD can be found in [BD95].

3.1.2 Seismic Reflection Tomography

In this application the sparse SVD problem arises from the solution of nonlinear inverse problems associated with the approximation of acoustic or elastic wave-speed from travel times. Specifically, the travel times $t(r)$ are related to the wave-speed (model parameters) through the relation

$$t(r) = \int_{r(s)} s(x, y, z) dl, \tag{3.2}$$

where $x, y,$ and z are spatial coordinates, dl is the distance (differential) along the ray r and $s(x, y, z) = 1/\mu(x, y, z)$ is the slowness (reciprocal of velocity) at the point (x, y, z). For large two-dimensional problems, the travel times, extracted from the original seismograms, can involve up to $O(10^5)$ rays. The ray path depends on the slowness (unknown) and thus Equation (3.2) must be linearized about some initial or reference slowness (unknown) model. Discretization of the slowness by cells or finite elements within which the slowness is assumed to be constant allows the linearized integral to be approximated as a sum. The resulting over-determined system of linear equations for the unknown slowness perturbation values is

$$D\Delta s = \Delta t, \tag{3.3}$$

where the components of Δt are the differences between the travel times computed for the model and observed times, the components of Δs are the differences between the initial and updated model, and D is the Jacobian matrix whose (i, j) element is the distance the i^{th} ray travels in the j^{th} cell. For two-dimensional models, the matrix D in Equation (3.3) is generally large ($O(10^5)$) and sparse.

For over-determined systems of equations such as Equation (3.3), the SVD is one of the best known methods for obtaining the linear least squares solution using the pseudo-inverse

$$D^{\dagger} = V_k {\Sigma_k}^{-1} {U_k}^T$$

where $k = \text{rank}(D), \Sigma_k = \text{diag}(\sigma_1, \sigma_2, \cdots, \sigma_k), {U_k}^T U_k = {V_k}^t V_k = I$. The smallest singular values and corresponding vectors control model parameters such as the velocity-reflector depth trade-off. Researchers in seismic reflection tomography can assess trade-offs in model parameters using sparse SVD methods to approximate large numbers of singular triplets above specified quantities (noise level).

3.2 Sparse Matrix Storage Formats

The input matrix encountered in the applications described in Section 3.1 is large and sparse. When solving these problems using iterative methods or Krylov subspace methods, a reduction in memory requirements may be achieved by storing only the nonzero elements of A. There are many methods for storing sparse matrices ([Saa90], [Eij92]). Of these formats, the compressed row and compressed column storage formats [B$^+$94] are the most general, i.e., they make no assumptions about the sparsity structure while avoiding the storage of any unnecessary elements. These formats produce an additional reduction in storage by economizing the storage of index information at some additional indirect-addressing overhead during execution. The reduction in storage from using compressed sparse row/column formats is considerable when $min(m, n)$ is much less than nnz, the number of non-zeros in the matrix, as happens with matrices encountered in the applications described in Section 3.1. Without loss of generality, it will be assumed for the rest of this discussion that the matrix is stored in Compressed Column Storage (CCS) format, also known as the Harwell-Boeing format [DGL89].

The CCS format is specified by the three arrays {*val*, *row_ind*, *col_ptr*} where *row_ind(i)* stores the row indices of each nonzero value *val(i)*. Nonzero values in the same column are stored as lists of contiguous elements in the array *val*, with *col_ptr(j)* marking the start of the j^{th} list in *val*. Figure 3.1 shows the values of *col_ptr, row_ind*, and *val* for a matrix with $nrow = 6, ncol = 4$. It can be seen that for $nnz = 12$ non-zeros, the storage requirements are $2nnz + ncol + 1$. This is less than the value $3nnz$ which would be required if the row- and column-index of each non-zero were stored. The disadvantage of this scheme, though, is that it is now necessary to perform more than one memory access to find the column index for any given value.

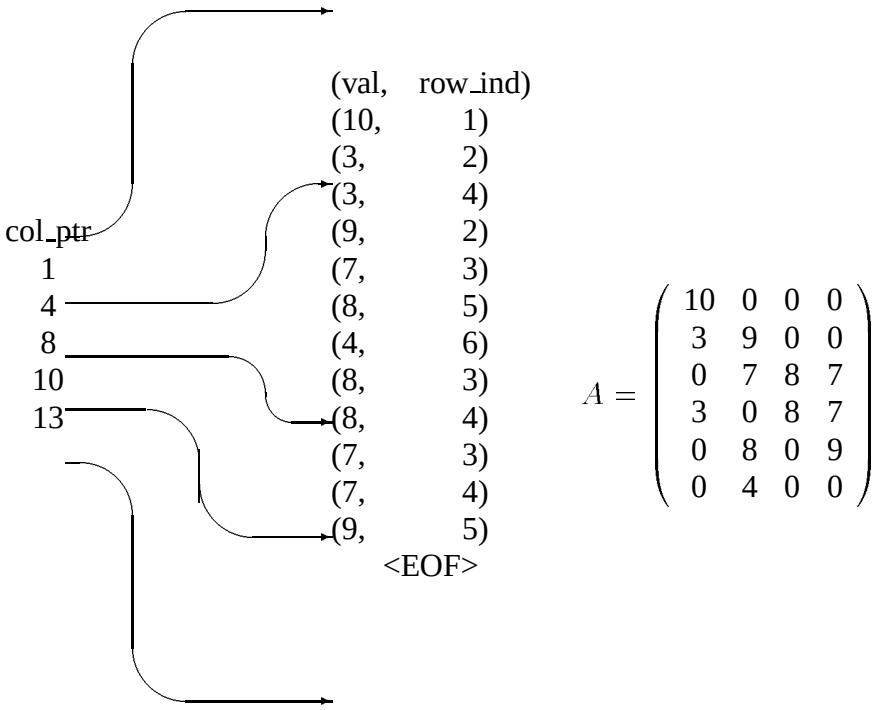


Figure 3.1. Example of the Harwell-Boeing storage format for a 6×4 sparse matrix A.

Figure 3.2 illustrates the pseudocode to perform the operation $y = Ax$ (Algorithm OP) and $y = A^T x$ (Algorithm OPT). It can be seen that both algorithms use indirect addressing, and thus have poor vectorizability properties for any architecture. However, Algorithm OPT has a more favorable memory access pattern in that it reads two vectors (*val()* and *x()*) and writes one scalar. Algorithm OP on the other hand reads elements $x()$ and *val* and performs both reads and writes of the indirectly-addressed elements in $y()$. Thus, unless the machine on which these methods are implemented has three separate memory paths, performance is limited by memory traffic.

Table 3.1 lists the average times to compute the matrix-vector products using Algorithms OP and OPT from Figure 3.2. Here the matrix-vector product was timed repeatedly within a loop, and the average elapsed time for 100 calls was calculated on a single SPARC processor. The datasets used were obtained from information retrieval applications. It can be seen that the time for multiplication by the transpose using Algorithm OPT is consistently less than the corresponding times for multiplication using Algorithm OP.

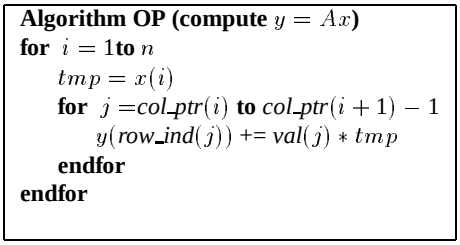


Figure 3.2. Computation of matrix-vector product when the sparse matrix A is stored in the Harwell-Boeing format

Table 3.1

Average (elapsed) time over 100 experiments for computing matrix-vector products, on a SPARCstation 20 - Model 50 (50 MHz SuperSPARC Processor with 256 Mbyte memory).

Dataset	dimensions	non-zeros	time(OP)	time(OPT)
adit	374×82	1343	0.7	0.6
CCE	3054×490	13607	10.2	9.1
oyang2hb	1853×625	3706	2.6	2.4
amoco1	1436×330	35210	18.1	17.1
bellcist	5143×1460	66340	47.3	42.4
knoxns	12615×40140	1780951	1507.6	1266
bellency	25629×56530	2843956	2483.906	2059.494

All times in milli-seconds

3.3 Parallelism

For most iterative and Krylovs subspace methods, the only operations involving the sparse $m \times n$ matrix A are the matrix-vector multiplications. The compressed format used to store the sparse matrix can thus impact single and multiprocessor program performance. [B$^+$94] addresses aspects of parallelism and identifies the basic time-consuming kernels of iterative schemes such as

- inner products,
- vector updates, and
- matrix-vector products, $A\xi^{(i)}$ and $A^T\xi^{(i)}$.

The computation of an inner product of two vectors and vector updates can be easily parallelized (see [B$^+$94]). However, when compressed storage of sparse matrices is used, the larger amount of message-passing involved complicates the parallelization of matrix-vector products.

In the current implementation of the CSI-MSVD algorithm the following scheme (which is also recommended in [B$^+$94]) is used to implment matrix-vector multiplication.

Algorithm P_OP: Compute $y^{m\times 1} = A^{m\times n}x^{n\times 1}$ using p processors

1. Let $n = qp + r$
2. Partition A column-wise, i.e., $A^{m\times n} = [A_1 A_2 \ldots A_r A_{r+1} \ldots A_p]$. Here, $A_i, i = 1, \ldots r$ is an $m \times (q+1)$ matrix containing columns $(i-1)q+i \ldots iq+i+1$ of A, and $A_i, i = r+1 \ldots p$ is an $m \times q$ matrix containing columns $(i-1)q+r+1 \ldots iq+r+1$. The i^{th} processor stores the block A_i.
3. Partition $x^{m\times 1}$ as
$$\begin{pmatrix} x_1 \\ \vdots \\ x_r \\ x_{r+1} \\ \vdots \\ x_p \end{pmatrix},$$
where $x_i \in \mathbb{R}^{(q+1)\times 1}$ when $i \in [1, r]$, and $x_i \in \mathbb{R}^{q\times 1}$ when $i \in [r+1, p]$. The i^{th} processor stores x_i.
4. The matrix vector product Ax can then be written as
$$Ax = \sum_{i=1}^{r} A_i x_i + \sum_{i=r+1}^{p} A_i x_i.$$

Thus, processor i can obtain the matrix-vector product $y_i = A_i x_i$ by applying Algorithm OP of Figure 3.2. The vector $y = Ax$ can then be obtained by doing a global reduction to get $\sum_i y_i$.

Algorithm P_OPT: Compute $y^{n\times 1} = transp(A^{m\times n}) * x^{m\times 1}$ using p processors

1. Let $n = qp + r$
2. Partition A column-wise as in Step 2 of algorithm P_OP, so that

$$A^T = \begin{bmatrix} {A_1}^T \\ {A_2}^T \\ \vdots \\ {A_r}^T \\ {A_{r+1}}^T \\ \vdots \\ {A_p}^T \end{bmatrix}.$$

3. The matrix vector product $A^T x$ can be written as

$$A^T = \begin{bmatrix} {A_1}^T x \\ {A_2}^T x \\ \vdots \\ {A_r}^T x \\ {A_{r+1}}^T x \\ \vdots \\ {A_p}^T x \end{bmatrix}.$$

Thus, processor i can obtain the matrix-vector product $y_i = {A_i}^T x$ by applying Algorithm OPT of Figure 3.2. The elements of the vector $y = A^T x$ are then automatically partitioned across the p processors. Note that, in this case, the vector x is **not** partitioned, and every processor must have all of the elements in x.

It should be noted that for CSI-MSVD, the only information about y that is needed by the remainder of the algorithm is $\|y\|$. By rewriting $\|y\|$ as

$$\sqrt{\sum_{i=1}^{p} {y_i}^2},$$

$\|y\|$ can be computed by performing a global reduction of ${y_i}^2$ across processors, and obtaining the square-root of the resulting scalar.

Chapter 4

Performance Evaluation Methodology

This chapter describes the computational platforms and parameters used to evaluate the performance of a parallel implementation of the CSI-MSVD algorithm.

4.1 Computational Environments

A distributed version of the CSI-MSVD algorithm as described in Section 2.4 was implemented using PVM, the Parallel Virtual Machine [GBD+94]. The objective was to produce a portable implementation that could be used across multiple platforms. PVM was chosen as the software environment because of its modularity, portability and immediate availability. Due to the wide-spread use of PVM in the scientific/engineering community, many vendors of massively-parallel computers provide optimized implementations of PVM for their machines. Portability of the PVM implementation of CSI-MSVD was also studied using one such platform, the CRAY T3D. This section describes the computational environments available through PVM and on the CRAY T3D.

4.1.1 PVM: Parallel Virtual Machine

PVM is a software package developed at the University of Tennesse, Knoxville (UTK) and the Oak Ridge National Laboratory (ORNL) that permits a heterogeneous collection of Unix computers linked together by a network to be used as a single large "virtually-parallel" computer. The PVM system is composed of two parts:

1. The PVM daemon `pvmd` that resides on all the computers making up the virtual machine. The PVM daemon must be started to create the virtual machine.
2. The library of PVM interface routines. This library contains user-callable routines for message-passing, spawning processes, coordinating tasks, and modifying the virtual machine.

The PVM computing model is based on the notion that an application consists of several $tasks$ which can be executed concurrently. **Functional parallelism** is achieved when each task performs a different function, and **data parallelism** is achieved when multiple tasks perform the same operations on different data. The PVM library routines for task-spawning, communication between tasks and virtual machine configuration management provide a portable interface to interprocess communication primitives like sockets.

Program development using PVM on a network of workstations has several advantages. Since the source code for PVM is available at no charge, parallel computing can take place on existing hardware without any additional overhead in investment. Applications developed using PVM are not locked into proprietary interfaces and algorithm development and testing can be done at low cost, using familiar environments and hardware. After a satisfactory implementation is available, the code can be ported to platforms with superior hardware. The virtual computer resources can grow in stages and take advantage of the latest computational and network technologies.

The major disadvantage of parallel programming using PVM on a network of workstations arises from the high latency of typical packet-switched networks and bus technology. The performance of bus-connected systems may degrade rapidly if the data transfer rate on the bus (i.e., bus bandwidth) is not able to deliver data to accommodate the processors. Typically, bus connections are limited to a modest number (> 30) of processors. Also, since each processor in the virtual machine is a workstation, networked computers can have several other users on them, running a variety of jobs, and sharing the bandwidth of the Ethernet bus with the parallel program. Program performance is usually affected by the overhead arising from these factors.

4.1.2 CRAY T3D Hardware/Software Overview

As discussed in the previous section, although PVM on a network of workstations can provide the necessary environment for algorithm development at low cost, the high message-passing overhead makes this model infeasible for the solution of problems in practice.

The CRAY T3D is a massively-parallel platform with superior hardware support that supports message-passing based on the PVM model. A hardware overview of the CRAY T3D is presented below followed by a description of the message-passing environment. Significant differences in the PVM models available on networked-workstations and the MPP are indicated. Further details are available in [CR94b].

CRAY T3D Hardware Overview

The CRAY T3D contains four types of components: processing element (PE) nodes, the interconnect network, I/O gateways, and a clock. Each PE node of the CRAY T3D contains a RISC 64-bit Alpha chip developed by Digital Equipment Corporation, local

memory, and support circuitry. There are two PE's per processing element node. Figure 4.1 illustrates the components of a PE node and the interconnect network.

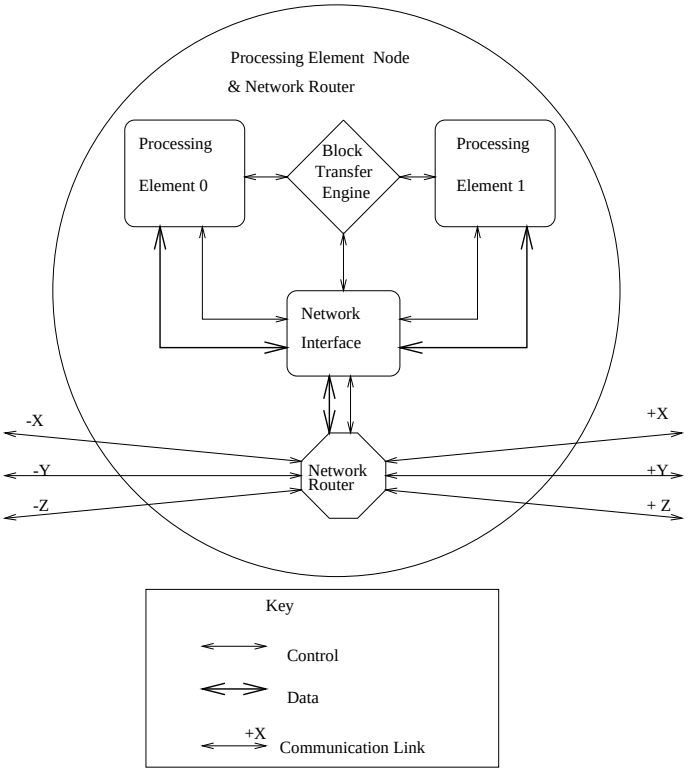


Figure 4.1. Processing element node and interconnect components on the CRAY T3D

The PE's are connected by a fast bidirectional 3-D torus system interconnect network (Figure 4.2). This topology ensures short connection paths and high bisection bandwidth (the maximum rate at which one half of the system can exchange data with the other half). With peak interprocessor communication rates of 300 Mbytes per second in every direction through the torus resulting in up to 76.8 Gbytes per second of bisection bandwidth, this design allows the extremely fast remote memory access critical for efficient MPP system usage.

The local memory within each PE is part of a physically distributed, logically shared memory system. System memory is physically distributed (since each PE contains local memory) and is logically shared since the microprocessor in one PE can access the memory of another PE without involving the microprocessor in that PE.

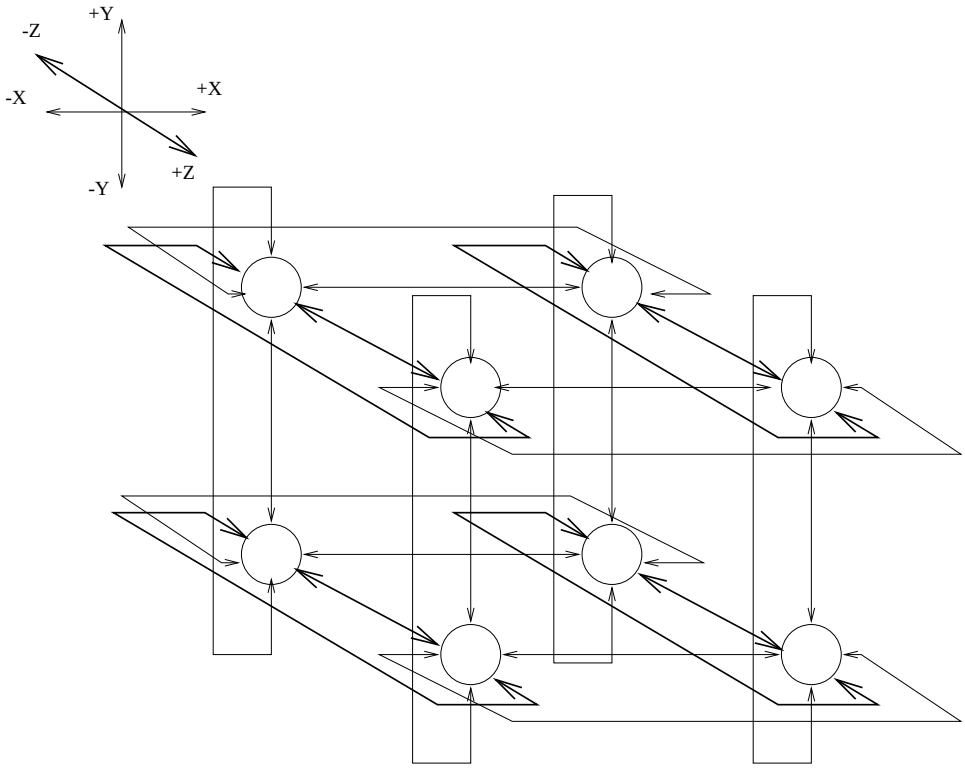


Figure 4.2. Two-dimensional Torus

CRAY T3D Programming Tools

Information about the FORTRAN, C and C++ compilers provided on the CRAY T3D may be found in [CR94b] and [CR94a]. The C compilers were used for this study. The operating system (UNICOS MAX) that supports the CRAY MPP system is a microkernel-based distributed operating system. This operating system runs on a CRAY T3D system and its CRAY Y-MP or C90 host.

All of the three compilers listed above support the message-passing programming methods using PVM. The three release packages of PVM libraries available are the CRAY Network PVM-3.2.0, the CRAY MPP PVM-3.1.0 and the CRAY T3D Emulator 1.1. A brief description of each is listed below.

- The CRAY Network PVM-3.2.0, or the *network version* is basically identical to the software developed and released at ORNL for networked-workstations. This version supports PVM across a heterogeneous network of computers and is based on TCP/IP and UDP data transfers between UNIX processes. In this kind of communication, the network transfer speeds are relatively slow.
- The CRAY MPP PVM-3.1.0, subsequently referred to as either the *CRAY MPP*

version or the *CRAY T3D version* uses the hardware capabilities of the CRAY T3D system to handle communications between the CRAY T3D processor elements and uses TCP/IP and UDP data transfers to handle communications outside the CRAY MPP system. The CRAY MPP version can be used in two modes. In a *stand-alone mode*, it can be used to program an application on the CRAY T3D system, like the message-passing libraries supplied for other MPP systems. In a *distributed* mode, it can be coupled with the network version to let the CRAY T3D applications use message-passing to communicate with processes running on the CRAY Y-MP host or any of the other systems that run PVM.

- The CRAY T3D Emulator 1.1, or the *emulator version*, allows programs to be developed and executed without having to use a CRAY T3D system. PVM is included in the emulator library.

Parallel programming with PVM can involve a combination of any of the above situations to provide a range of different scenarios. The following scenarios are listed roughly in increasing cost of communication:

1. One application running on a CRAY T3D system, requiring only the CRAY MPP version, used in stand-alone mode. High-performance connections are used between the tasks on the PE's in this version.
2. Two processes running on a single CRAY Y-MP computer system, requiring the networking version. Networking capabilities are used for communication
3. One process running on a CRAY Y-MP computer system, and another application running on an associated CRAY T3D system, requiring both the networking and the CRAY MPP version in distributed mode.

Therefore, in order to obtain the best message-passing latencies, it is desirable to use the CRAY MPP in stand-alone mode, with no participation from the Cray Y-MP front end. In the discussion that follows, it will be assumed that Cray MPP is used in this mode.

4.1.3 Differences Between CRAY MPP and NOW Versions of PVM

This section summarizes the salient differences between the MPP version of PVM, and PVM implementations for networks of workstations, henceforth referred to as NOW versions of PVM.

PE Number

The NOW versions only support the concept of PVM task identifier (`pvm_tid`) to identify a task on the virtual machine. The MPP version allows the use of PE numbers

in place of the pvm_tids. PE numbers are in the range $[0 \ldots npes]$, where $npes$ is the number of PE's in the current partition. On the CRAY T3D, in addition to support for dynamic groups [GBD+94], every task (or PE) is also a member of a static, predefined, *global group*. This group can be accessed through the PVM group manipulation functions defined for NOW versions by using a null name to refer to the global group.

In its current implementation, PVM limits the maximum number of tasks that can join a group. This is done assuming that most groups are small relative to the total number of PE's, and it saves memory for each group. Where possible, it is recommended that programs being ported from NOW versions of PVM to the CRAY MPP use the the predefined global group on the CRAY MPP system. This helps prevent the program from exceeding limits on group-size and also gives better performance when performing synchronization-barriers across the group or broadcasts to the group.

Process Spawning

The NOW versions of PVM allow any task in the virtual machine to spawn any number of images of an arbitrary executable through a call to pvm_spawn. In the MPP version, calls to pvm_spawn may be made only from the Y-MP host. A call to pvm_spawn causes a copy of the spawned program to execute on **each** PE in the current partition. The same executable image must execute on each PE in the partition, and tasks executing on the T3D nodes may not call pvm_spawn.

Virtual Machine Configuration

NOW versions of PVM provide the function pvm_config to provide information about the current configuration of the virtual machine. In the CRAY MPP version, the PVM machine is of fixed size and composition, and all PE's are running the same program. As a result, the full functionality of pvm_config is redundant and is therefore not implemented. The MPP version instead provides the functions pvm_get_PE() to return the PE number of the calling task. Also, the number of PE's in the current partition (i.e, in the virtual machine) may be obtained by determining the size of the global group by the function call

```
n_pes=pvm_gsize(NULL).
```

Word Size

Since all CRAY machines have a 64 bit word-size, and single-precision integers are defined to be 8 bytes, the CRAY MPP and emulator versions provide a constant, INTEGER8, for packing/unpacking functions in the FORTRAN interface to PVM.

Unsupported Functions

Some of the functions supported by the NOW version, but not implemented in the MPP version are `pvm_addhosts, pvm_delhosts`, `pvm_kill`, `pvm_mstat`, `pvm_notify`, `pvm_perror`, `pvm_sendsig`, `pvm_setopt`, `pvm_tasks.` Calling these functions on the CRAY T3D is permitted, but, if called, each will return the error status: `PvmNotImpl`.

`PvmDataInPlace` Semantics

The network and CRAY MPP version differ in the treatment of data buffers packed using `PvmDataInPlace` encoding. In the CRAY MPP version, such data must not be reused until the data has been unpacked by the receiving PE. If possible, down-sizing should be consistent. The caller is responsible for any additional synchronization or communication required to ensure this coordination.

Communicating with the PVM Daemon

PE's on the CRAY T3D system communicate with the daemon using sockets. Because of UNICOS limits in the number of open files per application, not all PE's may be able to communicate with the daemon. By default only PE0 establishes communication. Additional PE's may be allowed to communicate with the daemon by modifying the `PVM_PE_LIST` environment variable.

4.2 Performance Parameters

Two aspects of the performance of the CSI-MSVD algorithm are discussed in this section:

- Performance of the algorithm, measured in terms of convergence rates and magnitude of error.
- Performance of the parallel implementation, measured in terms of execution time, memory requirements and scalability.

The methods used to evaluate each of these aspects will be described in the following sections.

4.2.1 Performance of Algorithm CSI-MSVD

The relative performance of the algorithm is obtained by comparison with other widely used eigenvalue-solvers used to obtain the SVD of sparse matrices. [GL89] points out that Krylov-based methods are typically used to solve large, sparse eigenproblems. As

discussed in Section 3.2, matrix-vector multiplications are the fundamental and most expensive operations for iterative methods (CSI) and Krylov subspace methods (Lanczos, Arnoldi). Thus, the performance of each of these algorithms (CSI-MSVD, Lanczos and Arnoldi's method) can be estimated by computing the number of matrix-vector multiplications required when the same constraints on number of required eigenpairs (singular triplets) tolerance are used. Since the problems under consideration are large and sparse, memory requirements, which could prove to be a constraint, must be minimized. Also the sparsity pattern should not be disturbed, and the benefits of compressed storage should not be lost. Therefore, in this dissertation, cost is measured by evaluating the memory requirements of each method. A comparison of the performance (as measured by the number of matrix-vector multiplications) and the cost (as measured by the memory requirements) is used to assess the theoretical analysis of the two algorithms.

Another aspect of algorithm performance is the magnitude of the error in the singular triplet. When the SVD is obtained from the two-cyclic eigenvalue problem, the error in the singular triplet can be translated to the error in the eigenpair approximants obtained from CSI-MSVD. Following Section 2.6, the error in CSI-MSVD is approximated by

$$\tilde{\epsilon} = \begin{bmatrix} {\epsilon_1}^T \\ \tilde{\epsilon}_2^T \end{bmatrix}, \tag{4.1}$$

where ϵ_1 is defined in Equation (2.33) and $\tilde{\epsilon}_2$ is defined in Equation (2.38). The approximate $\tilde{\epsilon}$ defined in Equation (4.1) is compared to the following two norms:

1. The error in the eigenvector of the iteration matrix M defined in Equation (1.4), i.e.,
$$\| \frac{1}{\|Mv\|} Mv - v \|, \tag{4.2}$$
where $v = [{x_1}^{(k)^T}, {x_2}^{(k+1)^T}]^T$.

2. The error in the eigenpair of the iteration matrix M defined in Equation (1.4), i.e.,
$$\|Mv - \sigma_1 v\|, \tag{4.3}$$
where σ_1 is the current approximation to the largest singular value of the matrix A and $v = [{x_1}^{(k)^T}, {x_2}^{(k+1)^T}]^T$.

Since M is a two-cyclic matrix, the product Mv is of the form

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{pmatrix} {x_1}^{(k)} \\ {x_2}^{(k+1)} \end{pmatrix}$$

so that Equation (4.2) measures the error in the singular vectors, and Equation (4.3) measures the error in the singular triplet $\{{x_1}^{(k)^T}, {\sigma_1}^{(k+1)}, {x_2}^{(k+1)}\}$.

4.2.2 Performance Evaluation of the Parallel Implementation

The performance gain achieved by parallelizing a given application is typically measured by monitoring the speedup [KGGK94]. A related parameter for evaluating a parallel system is the scalability of the system, which is a measure of the capacity of the parallel system to increase the speedup in proportion to the number of processors. Some of the most common factors that prevent a linear increase in speedup with a larger number of processors [Hwa93] are communication overhead, the complexity of interprocess communication or synchronization overhead, and message-passing overhead, which are absent in serial programs. Also for multiprocessing and time-sharing systems, the accurate evaluation of CPU usage and time spent in communication for any one process is difficult, and best-case statistics do not necessarily predict the performance of the system for the average case.

The objective in this dissertation was to evaluate the speed of execution as perceived by the user. For this reason, the wall-clock time for execution was monitored. Although the time taken to start up all the processes and to print the final results were not considered for these experiments, the time spent in loading the matrix into memory was taken into account. This data was obtained by using a network of 24 machines on a LAN (10Mbps Ethernet) isolated by a bridge to avoid interference from external traffic on the Internet, and thus provide an estimate of the best-case performance. Each machine on the LAN was a Sun SPARCstation 5 Model 70 workstation with a clock speed of 70 MHz and 32 MB memory. Wall-clock times for execution using this configuration are listed in Section 5.4.

Although parallel speedup is a popular measure for evaluating parallel program performance, there are some pitfalls to this metric as indicated in [Com93], especially when used across multiple platforms. In this dissertation the speedup is computed to obtain an estimate of the scalability of the algorithm with a variation in problem size, so that for a given class of problems, recommendations can be made for the optimal choice of virtual machine configuration. Speedup is defined as

$$S = \frac{T_1}{T_p} \tag{4.4}$$

where T_1 is taken to be the wall-clock time for CSI-MSVD to execute on one processor without any redundant synchronization/communication operations, and T_p is the wall-clock time with p processors. Although this definition of T_1 does not give the wall-clock time with the best single-processor algorithm, the objective in computing S through Equation (4.4) is to obtain a measure of the speedup over single-processor time, i.e., the scalability of the system.

4.3 Input problems and Test Parameters

For all of the experimental results listed in this dissertation, the wall-clock time for execution was obtained as the difference between the values returned by calls to the UNIX function `gettimeofday()` [Ste92]. The time taken to spawn processes and to print the results was not considered, but the time to read in the matrix A from the disk was included in the wall-clock time. For experiments requiring the monitoring of system parameters like CPU time and page faults the BSD 3.2 function `getrusage()` [Ste92] was used wherever available. Each data point was obtained as the arithmetic mean of 10 samples.

The test problems used were obtained primarily from information retrieval applications and from seismic reflection tomography problems (see Section 3.1). Table 4.1 lists the sizes and sparsity of the matrices used for evaluating the performance of the CSI-MSVD algorithm. Here sparsity is defined by

$$Sparsity = \frac{nnz}{nrows \times ncols},$$

where nnz is the number of non-zeros, $nrows$ is the number of rows, and $ncols$ is the number of columns of the sparse matrix.

In order to study the effect of clustered spectra on the accuracy of the error estimate, synthetic diagonal test matrices having clustered or multiple diagonal elements were also used as input. The diagonal matrices used for these experiments included those described in [Ber94]. In addition, two other matrices CLUS2 and CLUS5 were used to introduce a larger separation between clusters. The diagonal elements were defined as clustered values of a step function chosen to reflect varying numbers of clusters with varying separation between clusters. Formally the diagonal elements were chosen to be a subset of

$$\bigcup_{i,k}(\alpha, \beta, \delta, k, i) = \bigcup_{i,k}(\alpha k + \beta) + \delta i,$$

where $1 \leq i \leq i_{max}, 0 \leq k \leq k_{max}$.

Here, δ defines the separation of elements within a cluster, α defines the separation between consecutive clusters, i_{max} defines the maximum number of distinct elements within a cluster, and k_{max}, α and β are such that for any diagonal element σ_j in the k^{th} cluster C_k, $\lfloor \sigma_j \rfloor = \alpha k + \beta$.

The 11-largest diagonal elements of two such matrices CLUS2 and CLUS5 are shown in Figure 4.3. The parameters for these diagonal test matrices are given in Table 4.2.

Table 4.1

Test matrices used to evaluate the performance of CSI-MSVD.

Matrix	Source	Dimension	Number of non-zeros	Sparsity
amoco1	Amoco Research	1436×330	35210	0.0074
amoco2t	Amoco Research	8754×9855	$1,159,116$	0.0018
belladit	Bellcore Linguistics data	374×82	1343	0.0438
bellcrat	Bellcore Linguistics data	4997×1400	$78,942$	0.0113
bellcist	Bellcore Linguistics data	5143×1460	$66,340$	0.0088
bellency	AA Encyclopedia	25629×56530	$2,843,956$	0.0020
belltect	Bellcore Linguistics data	16637×6535	$327,244$	0.0030
belltrec3	Bellcore Linguistics data	10836×48809	2343775	0.0044
ccelink	Hypertext links from Columbia Condensed Encyclopedia	12025×9778	$33,545$	0.0002
greenh	Greenhouse Effects News	318×100	5772	0.1820
knoxns	Term-document matrix from Knoxville News-Sentinel	12615×40140	$1,780,951$	0.0035

Table 4.2

Parameters for some diagonal test matrices

Matrix	$(\alpha, \beta, \delta, k_{max}, i_{max})$
CLUS1	$(4, 0, 10^{-3}, 12, 4)$
CLUS2	$(1, 0, 10^{-6}, 4, 10)$
CLUS3	$(26, -10^{-8}, 10^{-8}, 1, 25)$
CLUS4	$(10, 1, 0, 4, 10)$
CLUS5	$(10, 8, 10^{-3}, 4, 4)$

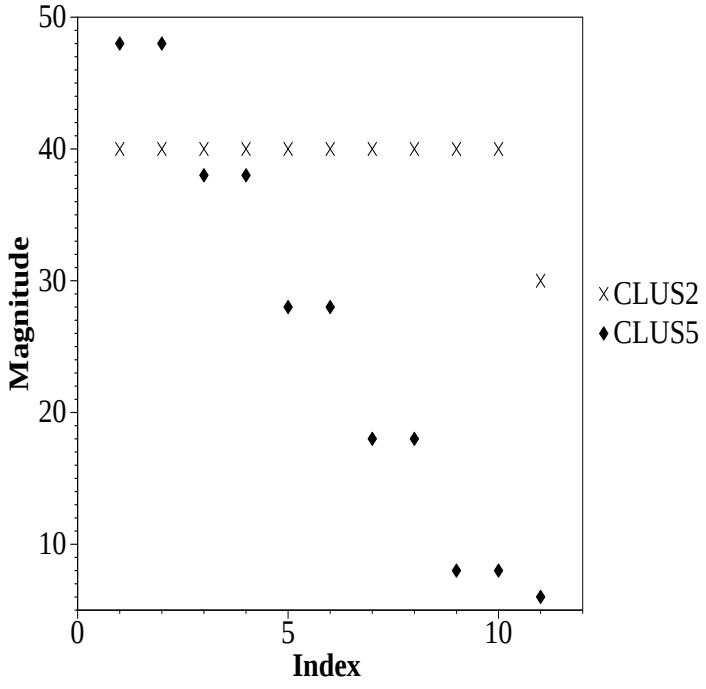


Figure 4.3. Clustered spectra of two 50×50 test matrices.

Chapter 5

Results

Experimental results to demonstrate the performance of PVM implementations of the CSI-MSVD algorithm for two-cyclic matrices are described in this Chapter. First, an evaluation of the theoretical performance of the algorithm is presented through comparisons with Krylov subspace methods like Lanczos and Arnoldi's method in Section 5.1. Section 5.2 presents an analysis of the reliability of the schemes for error estimation described earlier in Section 2.6. The CSI-MSVD algorithm for two-cyclic matrices as defined in Section 2.4.1 was implemented in ANSI C using PVM 3.3.7 for a network of workstations. An overview of the parallel implementation is presented in Section 5.3, and parallel program performance is evaluated by studying parameters such as scalability in Section 5.4. Other aspects of parallel programming such as load balancing are also described in Section 5.5.

As discussed in Chapter 4, vendors of massively-parallel platforms often supply a PVM interface to their message-passing libraries so that PVM applications developed on networked-workstations may be ported easily to these massively parallel computers. The portability of the PVM implementation of the CSI-MSVD algorithm was studied by experiments on the CRAY T3D. The performance of the PVM implementation of the CSI-MSVD algorithm using the Cray MPP implementation of PVM (see Section 4.1.2) is described in Section 5.6 to illustrate the portablity of the current implementation.

5.1 Comparisons with Krylov methods.

It has been pointed out in [Ber90] that the Lanczos algorithm has been demonstrated to be the fastest method among Krylov- and subspace-iteration based methods for computing several of the largest singular values and corresponding vectors of large sparse matrices. Arnoldi's method, a generalization of the Lanczos method, may also be used to obtain the SVD by solving the eigenvalue problems described in Lemma 1.1. However, since the eigenvalue problems under consideration in this dissertation are symmetric eigenvalue problems, Arnoldi's method reduces to the symmetric Lanczos

algorithm.

The implementation was tested on a network of 24 machines, each a Sparc-5 Model 70 (70 MHz) having 32 MBytes internal memory. Cost-performance comparisons between this implementation and library implementations of the Lanczos algorithm in SVDPACK [Ber92] are described in this Section. Experiments using CSI-MSVD as a preconditioner to Arnoldi's method implemented in [LSV94] are also described.

5.1.1 CSI-MSVD versus LAS

The cost and performance of the PVM implementation of CSI-MSVD were compared with LAS1 and LAS2, the Lanczos algorithms implemented in SVDPACK [Ber92]. LAS1 solves the two-cyclic eigenvalue problem (i.e., the eigenvalues of the matrix M defined in Equation (1.4)), and LAS2 solves the eigenvalue problem for the matrix $A^T A$. In order to estimate the cost of each method, the maximum memory requirement for any process in the virtual machine for CSI-MSVD was compared to the corresponding requirements for LAS1 and LAS2.

Table 5.1

Memory requirements for matrix-vector multiplication in CSI-MSVD using 20 processors.

Matrix	Memory (Mbytes)
amoco	2.458304
bellcist	0.814496
bellcrat	0.799032
bellency	2.708668
belltect	8.019468
knoxns	3.796876

It was found that the dominant memory-cost for CSI-MSVD was for the storage for the matrix A itself. The memory requirements for calculating the triangular matrix σ_{kl} defined in Equation (2.25) and for the extraction of singular values from the resulting bidiagonal matrix in Equation (2.31) are bounded by the number of iterations i required for convergence. As demonstrated by Equations (2.23) and (2.22), the construction of the Jacobi matrix is accelerated by the extraction of moments ν_{2k} and ν_{2k+1} at step $k+1$, and hence the number of iterations i is usually small. In practice, i was found to be of the $O(10)$, and a maximum iteration limit (MAXIT) of 50 was found to be sufficient for all of the matrices used. For this value of MAXIT, the upper-bound on the memory requirements for the calculation of σ_{kl} was found to be 0.191 Kbytes and that for the bidiagonal QR iteration was 0.456 Kbytes. In contrast, the combined memory requirements for the matrix storage and the Chebyshev iteration are shown in Table 5.1.

The memory requirement for storage of the matrix in CSI-MSVD is one that can be controlled by varying the number of processors used. Since this requirement dominates the memory required for the calculations intrinsic to the algorithm itself, there is some control over the cost of CSI-MSVD.

Table 5.2

Memory requirements in MBytes for LAS1 and LAS2. Values, as reported by software from SVDPACK, indicate memory required *in addition to that for the storage of the matrix.*

File Name	LAS1	LAS2
amoco	31.2084	31.6440
bellcist	11.1343	11.2818
bellcrat	10.7899	10.9325
bellency	137.464	139.424
belltect	38.8377	39.3829
knoxns	88.3005	89.5556

The memory requirements for the Lanczos algorithms from SVDPACK are listed in Table 5.2. It can be seen for large matrices, even if the matrix is partitioned across several processors, and matrix-vector multiplication is carried out through data-parallel computation, the memory requirements for the algorithm could prove to be an insurmountable bottle-neck.

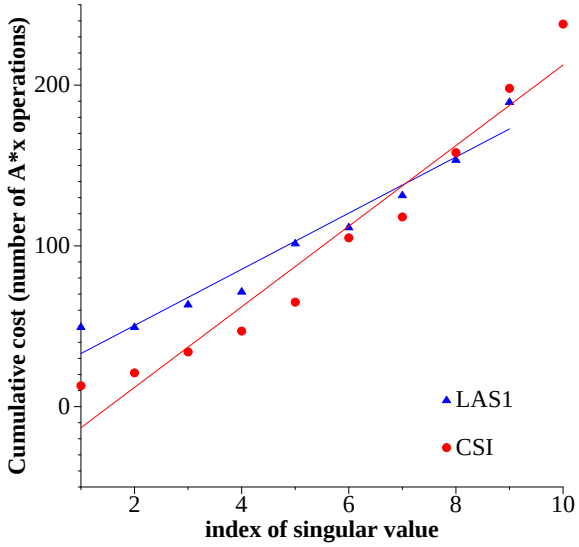
In order to study the performance of each method, the number of matrix-vector multiplications was compared for CSI-MSVD, LAS1 and LAS2. Table 5.3 records the number of Ax operations when calculating the largest 10 singular values and the corresponding vectors for the matrices considered earlier. Here, the maximum iteration limit for the Lanczos codes was set to 100, and to 50 for CSI-MSVD. A tolerance limit of 10^{-6} was requested from both methods.

Although CSI-MSVD takes about 3 times as many matrix-vector multiplications as LAS2, the number of these operations is of the same order of magnitude as LAS1. Better insight into this behavior may be obtained by examining the cumulative overhead for computing singular-triplets. Figure 5.1 shows a comparison of the cumulative overhead for the matrix `bellency`. Here the number of matrix-vector multiplications required

Table 5.3

Performance comparisons, as measured by the number of matrix-vector multiplications required by each method, when calculating the 10-largest singular values and corresponding singular vectors.

File Name	CSI	LAS1	LAS2
amoco	221	164	86
bellcist	289	194	122
bellcrat	263	196	86
bellency	238	192	86
belltect	255	196	118
knoxns	208	120	71



Matrix: BELLENCY (25629 rows, 56530 columns, 2,843,956 nonzeros)

Figure 5.1. **Comparison of cumulative overhead when computing singular triplets by LAS1 and CSI-MSVD.** CSI-MSVD requires fewer matrix-vector multiplications to calculate each triplet, but LAS1 has a lower cumulative count for $k > 8$ because more than one singular triplet can be deflated at each step.

to get the k-largest triplets using LAS1 and CSI-MSVD is plotted with a linear-least squares fit of degree 1 through the data points. For LAS1 the minimum subspace to compute the k-largest triplets to an accuracy of 10^{-6} was used. It can be seen from Figure 5.1 that the performance of CSI-MSVD as measured by the number of matrix-vector multiplications is lower than the corresponding values for LAS1 for the first 8 triplets. For the first triplet, CSI-MSVD takes 13 iterations while LAS1 takes 50 iterations. However, the incremental overhead for calculating each additional triplet for CSI-MSVD is high as indicated by the slope of the interpolating line. This high incremental overhead could be reduced if, at each step, it were possible to accept more than one singular triplet. However, the absence of more than one singular vector approximation precludes this, and the performance of CSI-MSVD deteriorates more rapidly than that of LAS1.

The choice of Wielandt's scheme for deflation provides robustness in CSI-MSVD. The effectiveness of deflation is not as sensitive to the orthogonality of singular-vector approximates as in Lanczos algorithms, and convergence is not affected by loss of orthogonality due to round-off. Thus, vectors can be orthogonalized by one step of total re-orthogonalization after the desired number of triplets are found or as frequently as desired in the application without introducing a synchronization point in the algorithm.

5.1.2 CSI-MSVD versus Arnoldi's method

The feasibility of using CSI-MSVD as a preconditioner to Krylov-based methods like Arnoldi's algorithm was investigated by using Matlab 4.2 implementations of CSI-MSVD in combination with a k-step Arnoldi method (*Arnupd*) as implemented in [LSV94]. *Arnupd* iterates with a subspace of dimension 6, and the number of desired eigenvalues k=1 with $p = 5$ extra vectors calculated at each step to obtain the partial Schur decomposition for the iteration matrix M where

$$MQ \cong QH.$$

The starting vector for Arnoldi's method in *Arnupd* was provided by the singular-vector approximation from CSI-MSVD.

Table 5.4 tabulates the number of Arnoldi iterations required using the iterates generated by the CSI-MSVD procedure as compared to running Arnoldi with a random starting vector for the ADI matrix. There is a clear reduction in the number of Arnoldi iterations so that at most 1 Arnoldi iteration is needed to refine the singular vectors. Also, the residual error is larger when a random starting vector is used with *Arnupd*.

It should be noted that, since *Arnupd* tests for convergence by ensuring that the error in the Rayleigh Quotient $\|Q^T MQ - H\|_2$ is within the user-defined tolerance *tol*, the actual error in the eigenpair is given by $\epsilon = \|Mv_i - \lambda_i v_i\|_2$. Here λ_i is obtained as the eigenvalues of H, and $v_i = Q * y_i$ where y_i is the eigenvector of H corresponding to λ_i. The actual error ϵ is typically larger than *tol*. When the subspace size is

Table 5.4

Refinement by Arnoldi; parameters used for CSI-MSVD were $\eta_1 = 5$, $\eta_2 = 15$ and parameters used with *Arnupd* were k=1 and p=5.

Singular Value	Arnoldi for singular vector refinement			Arnoldi: rand starting values	
	Arnupd iterations for tol: 10^{-3}	tol: 10^{-6}	error	Arnupd iterations tol: 10^{-6}	error
2.078676e+01	1	1	8.348e-06	1	2.780e-02
1.423534e+01	1	1	7.702e-05	3	8.427e-03
1.295953e+01	1	1	3.935e-05	2	1.248e-02
1.056388e+01	1	1	6.652e-05	3	1.083e-02
9.614439e+00	1	1	2.618e-04	5	7.000e-03
8.907366e+00	1	1	8.133e-04	9	8.748e-03
8.660762e+00	1	1	5.374e-04	6	1.367e-02
8.356867e+00	1	1	8.394e-04	6	6.465e-03
8.157926e+00	1	1	7.586e-04	8	7.376e-03

kept constant for a given user-defined tolerance tol, starting vectors generated by the CSI-MSVD algorithm consistently yield improved errors in the eigenpairs computed by Arnoldi's method. In addition, as indicated by Figure 5.1, since the CSI-MSVD algorithm typically requires $O(10)$ matrix-vector multiplications to converge to each eigenvalue, a considerable improvement in the accuracy in the solution obtained from $Arnupd$ can be obtained at very little cost.

5.2 Error estimation

In order to study the magnitude and validity of the error estimate, several matrices from information retrieval applications and synthetically generated test matrices were used. A description of these matrices is provided in Section 4.3. The 10-largest singular values and the corresponding singular vectors were calculated.

For typical term-document matrices obtained from information retrieval applications, it was found that in spite of the perturbation $\kappa_2 A^T \epsilon_1$ defined in Equation (2.37), the error estimate defined by Equation (2.32) provides an upper bound on the actual error in the vector for up to 8 of the largest triplets (see Figure 5.2). Even when the estimate is not a good upper bound on the actual error, a good approximation to the actual error is available from the estimate, especially for well-separated singular values. Thus the error estimate can be used as a reasonable indicator of the acceptability of the current eigenvector approximation, even though it may not always be an upper bound on the actual error in the eigenvector, and the explicit computation of the exact error may be

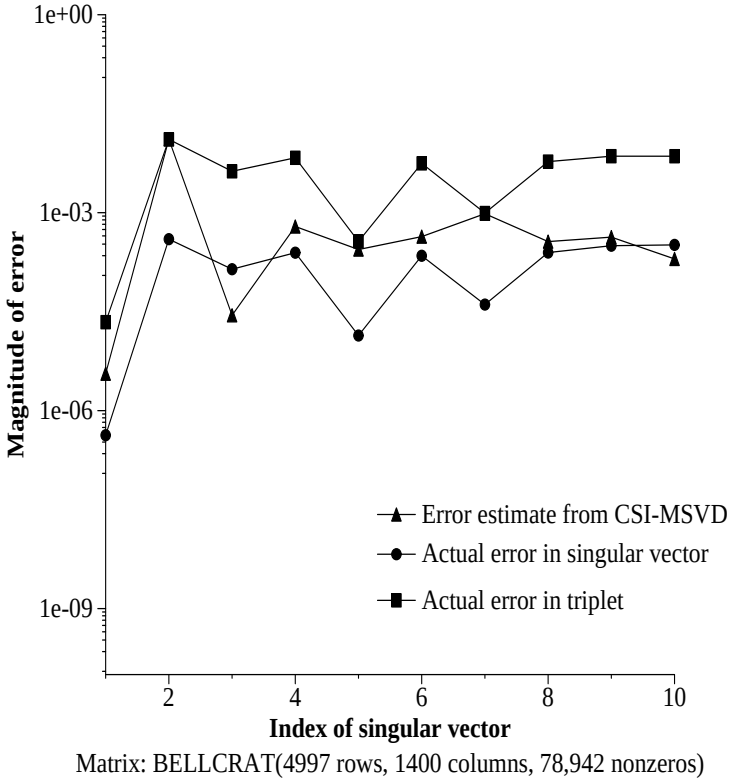


Figure 5.2. Variation in the reliability of the error estimate as an upper-bound for clustered singular values.

avoided through this scheme.

Figure 5.3 shows the error estimate, and the actual error measures defined in Equations (4.2) and (4.3) for two of the diagonal test matrices described in Table 4.2. For the matrix CLUS2, whose 10-largest singular values are clustered around 40, the error estimate cannot provide a good bound on the actual error for the fourth to seventh singular triplets in the 10-element cluster around the singular value 40. The effect of clustering on the error estimate is more obvious for the matrix CLUS5. Here, for the 11-largest triplets, each cluster has at most 2 values, with a large separation ($\alpha = 10$) between consecutive clusters. When the gap between the values of the largest and second-largest singular-values of the deflated matrix is large, $\mu = \sigma_2/\sigma_1$ is relatively small and a good approximation to the actual error is provided by Equation (4.1). Figure 5.3(b) illustrates that when σ_2 and σ_1 are not well-separated, the validity of the error estimate as a reliable upper-bound becomes more questionable ($\sigma_j, j \in \{1, 3, 5, 7, 9\}$). However, the CSI-MSVD algorithm invokes an external refinement procedure SYMMLQ when clustered singular values are encountered. The error estimate provided by the SYMMLQ overrides the estimate given by Equation (4.1) so that an accurate error estimate is obtained regardless of the clustering in the singular values.

5.3 Parallel Implementations

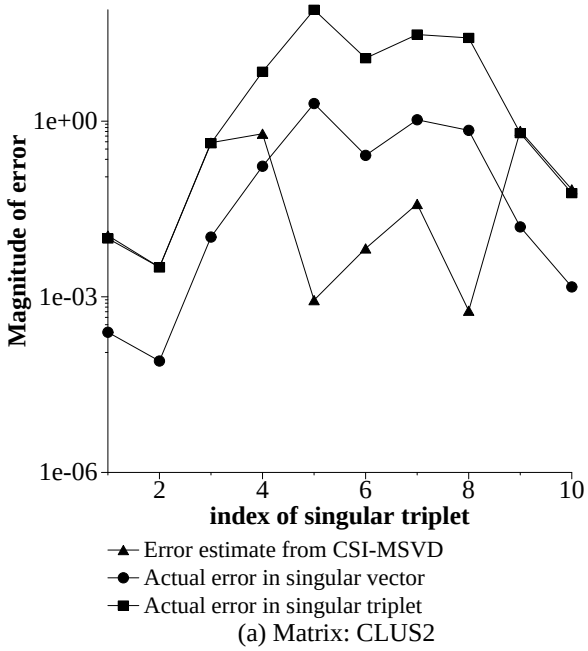
The components of the parallel implementation and the parameters controlling communication and computation overhead will first be described. Then the parallel program performance will be examined, using the methods described in Section 4.2.2.

The major functional components of CSI-MSVD (see Figure 2.3) are

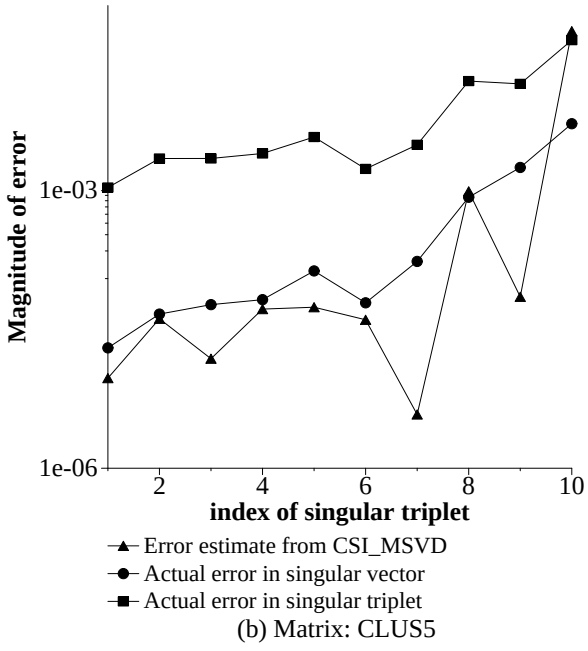
- MATVEC(i) — When the matrix has been partitioned across p processors, this indicates the i-th process participating in matrix-vector multiplication, and thus, the Chebyshev semi-iterative method defined by Equation (2.12). Processes MATVEC(i), $0 \leq i < p$ are enrolled in a logical, dynamic PVM group, MATVEC.
- SIGMA — Process that updates the array σ_{kl} defined in Equation (2.25).
- GAMMA — Process that performs the bidiagonal QR-iteration to approximate the singular values of the current bidiagonal matrix B_k in Equation (2.31)
- MAIN — Driver program that initializes parameters for CSI-MSVD and keeps track of deflation.

The processes MATVEC, SIGMA and GAMMA are pipelined so that MATVEC initiates the pipeline by sending moments to SIGMA. SIGMA then updates the bidiagonal matrix, and GAMMA, the third process in the pipeline, performs the QR-iteration with the updated bidiagonal matrix.

As noted in Section 5.1.1, due to the accelerated computation of moments and the



(a) Matrix: CLUS2



(b) Matrix: CLUS5

Figure 5.3. Error estimates for computing the 10-largest singular triplets for 50×50 diagonal test matrices.

simplifications possible for the two-cyclic iteration (Section 2.4.1), the sizes of the matrix σ_{kl} defined by Equation (2.25) and the bidiagonal matrix are independent of the problem size in practice. Hence the computational workload in SIGMA and GAMMA is not large enough to necessitate the parallelization of these components.

The load on processors involved in the MATVEC group, on the other hand, is dependent on the size of the input sparse matrix. It is possible that a single processor may not be able to satisfy the memory requirements of some large input matrices. Even if the matrix could be accommodated in a single processor, it is desirable to partition a large matrix across multiple processors to obtain the benefits of distributed computation of the Chebyshev iteration.

5.3.1 Data Parallel implementation of CSI

One method to obtain a data-parallel computation of the Chebyshev semi-iterative method using only those operations defined by PVM will now be described.

Let the iterate at the k^{th} step, $\xi^{(k)}$ be partitioned so that $\xi^{(k)}[1 : m] = x^{(k)}$ and $\xi^{(k)}[m + 1 : n] = y^{(k)}$ where m, n are the number of rows and columns of the input matrix A. Further, assume that the sparse matrix A, and the elements of $x^{(k)}$ and $y^{(k)}$ are distributed across p processors as described in Section 3.3 so that processor i has A_i, $x^{(k)}$ and ${y_i}^{(k)}$ stored in its local memory. The Chebyshev semi-iterative method now proceeds as follows:

1. Calculate $A_i {y_i}^{(k)}$ on the i^{th} processor
2. Sum the result of Step 1 across all processors (global reduction), and store result in processor l_1. Note that since $A_i {y_i}^{(k)}$ is an m-element vector, the complexity of this step is $O(m)$.
3. On processor l_1 compute $x^{(k+1)}$ using Equation (2.27), with the equivalences $x_1 \leftrightarrow x$, $x_2 \leftrightarrow y$, $m \leftrightarrow k$.
4. The scalar $\langle y^{(k)}, y^{(k)} \rangle$ computed in iteration $k - 1$ at Step 7 may be sent to the process SIGMA for computation of the new anti-diagonal as illustrated in Figure 2.3.
5. On completion of Step 3, processor l_1 must broadcast the vector $x^{(k+1)}$ to all other processors in the group, in preparation for the computation of $A^T x^{(k+1)}$. Note that this is necessary because, as discussed in Section 3.3, every processor must have all of the elements of $x^{(k+1)}$. Thus, an unavoidable synchronization step is required so that the value of $x^{(k+1)}$ is consistent across all processors. Since $x^{(k+1)}$ is an m-element vector, the communication overhead involved is $O(m)$
6. On processor i, compute ${y_i}^{(k+1)}$ using Equation (2.28).
7. Globally reduce $\langle {y_i}^{(k+1)}, {y_i}^{(k+1)} \rangle$ to store the result in processor l_2.

8. The value: $\langle x^{(k+1)}, x^{(k+1)} \rangle$ may be sent to the process SIGMA for computation of the new anti-diagonal.

It can be seen that the length of the largest messages (and thus the complexity of message-passing) are determined by m, the number of rows in the input matrix. For rectangular matrices, when $m \neq n$, it is therefore desirable to have $m < n$, and, if necessary, use the transposed matrix to achieve this effect.

For example, consider the matrix BELLTECT described in Table 4.1 with 16,637 rows and 6,535 column. Figure 5.4 illustrate the wall-clock times for computing the 10-largest singular triplets of BELLTECT using CSI-MSVD with varying numbers of processors on the CRAY T3D. The corresponding times for the transposed matrix BELLTECH (6,535 rows, 16,637 columns) are also shown in Figure 5.4. Although BELLTECH and BELLTECT have almost identical execution times when the size of

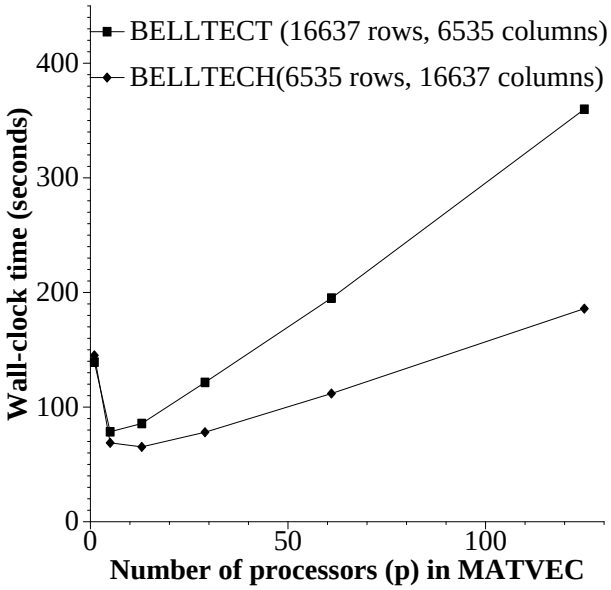


Figure 5.4. Effect of m, the number of rows, on wall-clock times in seconds obtained on the CRAY T3D. BELLTECT is the transpose of the BELLTECH. The communication-overhead is proportional to the number of rows in the matrix, which causes higher wall-clock times for BELLTECT.

the MATVEC group, $p \leq 2$, these times are much larger for BELLTECT when $p > 2$, indicating the phenomenon of higher communication overhead from the larger number of rows. Thus, it is preferable to use BELLTECH as the input matrix to CSI-MSVD.

5.4 Scalability

The wall-clock times for execution of CSI-MSVD on a network of SPARCstation 5 machines on a LAN isolated by a bridge were monitored as described in Section 4.2.2.

For small matrices ($\leq 500,000$ non-zeros) the benefits obtained by partitioning the

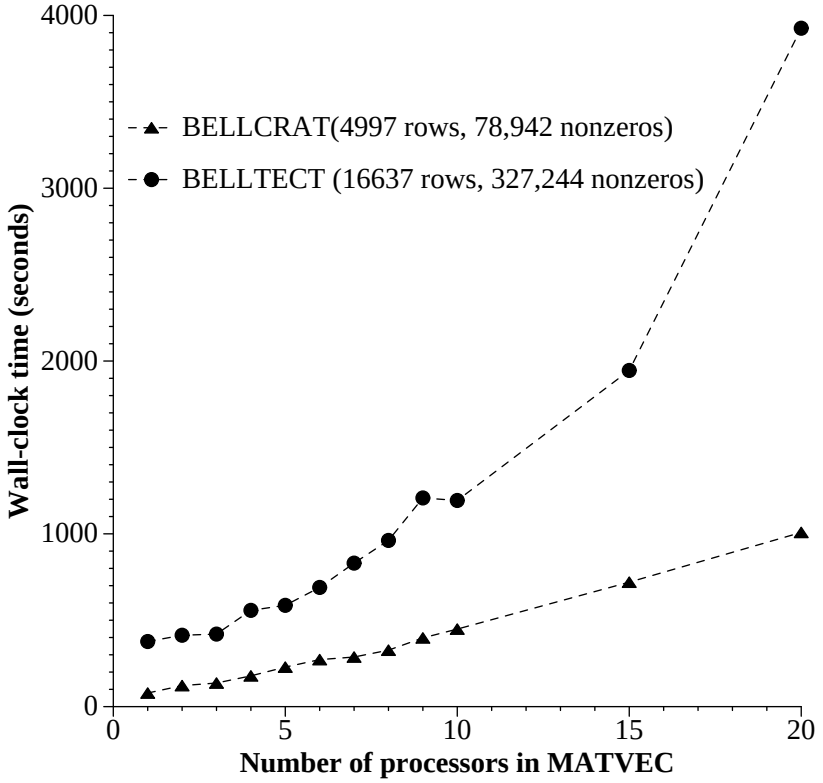


Figure 5.5. CSI-MSVD: wall-clock times (seconds) for execution for matrices with $500,000$ or fewer non-zeros

matrix across multiple processors are overshadowed by the communication overhead so that partitioning the matrix over more than one processor results in a slow-down. Figure 5.5 illustrates this effect for the matrices BELLCRAT and BELLTECT. The values of the number of non-zeros and the number of rows (m) for BELLTECT are approximately 4 times the values for BELLCRAT, and this is reflected in the corresponding wall-clock times.

The detrimental effect of communication overhead can be observed more clearly in Figure 5.6. Here the two matrices AMOCO1 and CCELINK were used in experiments on the CRAY T3D. Wall-clock times for execution for computing 10 singular triplets to 10^{-6} accuracy. AMOCO1 is a $1,436 \times 330$ matrix with $35,210$ non-zeros, while CCELINK is a $12,025 \times 9,778$ matrix with $33,545$ non-zeros. Thus, although both matrices have approximately the same number of non-zeros, they have widely different number of rows and columns and thus very different sparsity. As expected, the CSI-MSVD algorithm encounters larger communication overhead with CCELINK, which is also the sparse matrix, so that larger execution times are observed in this case.

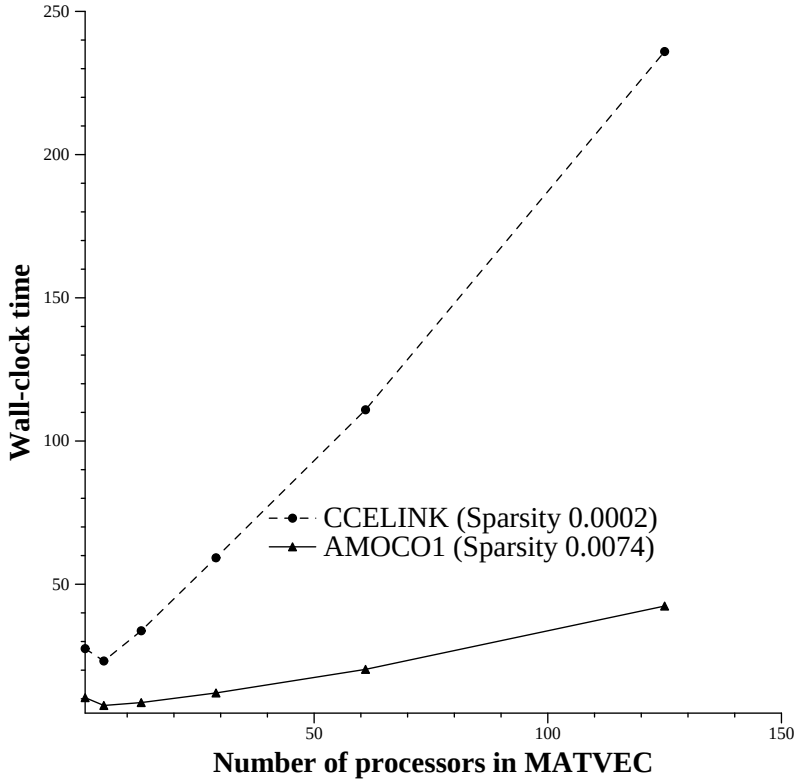


Figure 5.6. Effect of communication overhead and sparsity on the parallel program performance. Elapsed wall-clock times on the CRAY T3D were recorded.

While the slow-down exhibited in Figure 5.5 with increasing number of processors might appear to detract from any advantages of parallelism, it is also encountered when traditional sequential methods like Lanczos are used to solve large eigenvalue problems by partitioning the matrix across multiple processors.

Figure 5.6 also illustrates that the number of non-zeros is not the only indicator of the complexity of computing the eigenvalues using CSI-MSVD for the matrices AMOCO1 and CCELINK, and factors such as difference in sparsity and communication overhead must be considered in order to construct run-time performance models. The factors that affect wall-clock execution time, listed in relative order of importance are:

- number of rows
- number of non-zeros
- sparsity

For large matrices ($> 10^6$ non-zeros), however, the time to calculate the matrix-vector product by storing the entire matrix on one processor is much larger than the

corresponding time when the matrix is partitioned across multiple processors. The parallel program has an additional advantage: there are fewer page faults per process for the partitioned matrix so that a super-linear speedup can be observed in some cases (Figure 5.7). Figures 5.7 and 5.8 show the trend in wall-clock times with varying numbers of non-zeros when the number of rows (and thus, the communication overhead) are of the same order of magnitude for all the matrices being considered. The optimal choice for p, the number of processors in MATVEC at which wall-clock times are mimimized, appears to be approximately 5. While this implies that the algorithm cannot exploit the parallelism afforded by a larger number of processors, it also indicates that relatively few resources are required to obtain optimal performance.

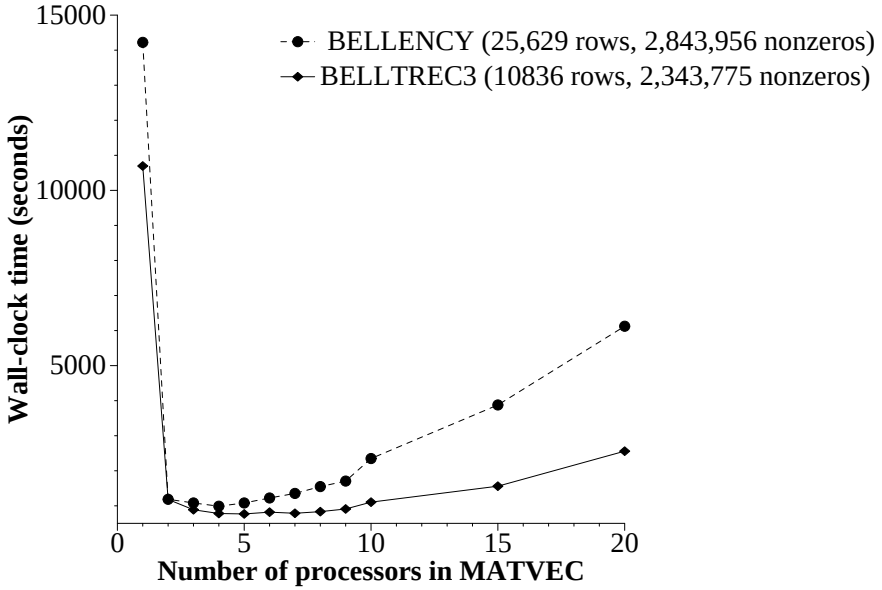


Figure 5.7. CSI-MSVD wall-clock times (seconds) for execution for matrices with $\approx 2 \times 10^6$ non-zeros on a network of SUN workstations.

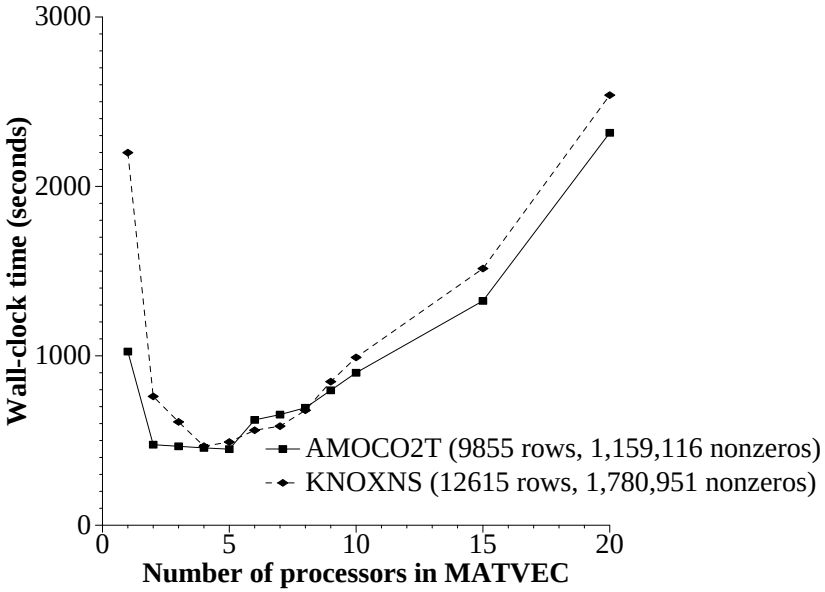


Figure 5.8. CSI-MSVD wall-clock times (seconds) for execution for matrices with $\approx 10^6$ non-zeros on a network of SUN workstations.

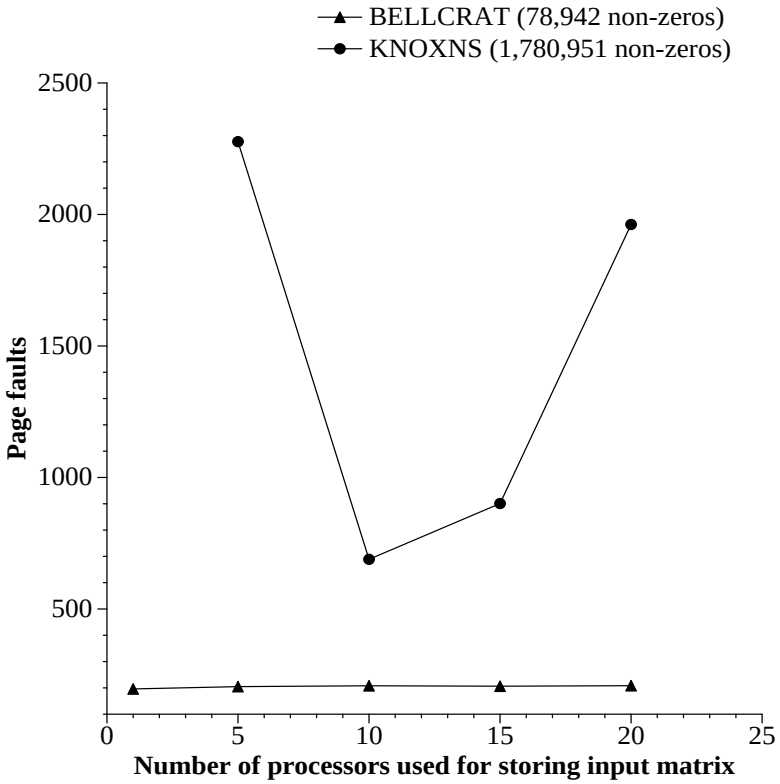


Figure 5.9. Average number of page faults per processor, with a variation in the number of processors, for 2 matrices of different sizes

To further investigate the effect of variation in page faults with varying numbers of processors, the number of page-faults per process involved in matrix-vector multiplication was monitored within the program through calls to `getrusage()` ([Ste92]). Figure 5.9 illustrates the variation in the number of page-faults with a varying number of processors. As expected, for smaller matrices like BELLCRAT ($4997 \times 1400, 78942$ non-zeros), the number of page-faults does not vary substantially with the number of processors over which the matrix is partitioned. However, for large matrices such as KNOXNS ($12,615 \times 40,140, 1,780,951$ non-zeros, the average number of page faults per process is affected by factors such as number of columns of the matrix assigned to the process, the distribution of non-zeros in these columns, and hardware characteristics such as cache size.

Figure 5.10 summarizes the speedups for four matrices AMOCO2T, KNOXNS, BELLTREC3 and BELLENCY, where the speedup is defined by Equation (4.4). Since the number of rows for AMOCO2T, KNOXNS and BELLTREC3 are all about $10,000$, the communication overhead for these three matrices is approximately the same. AMOCO2T, with $1,159,116$ non-zeros gives rise to the smallest computational

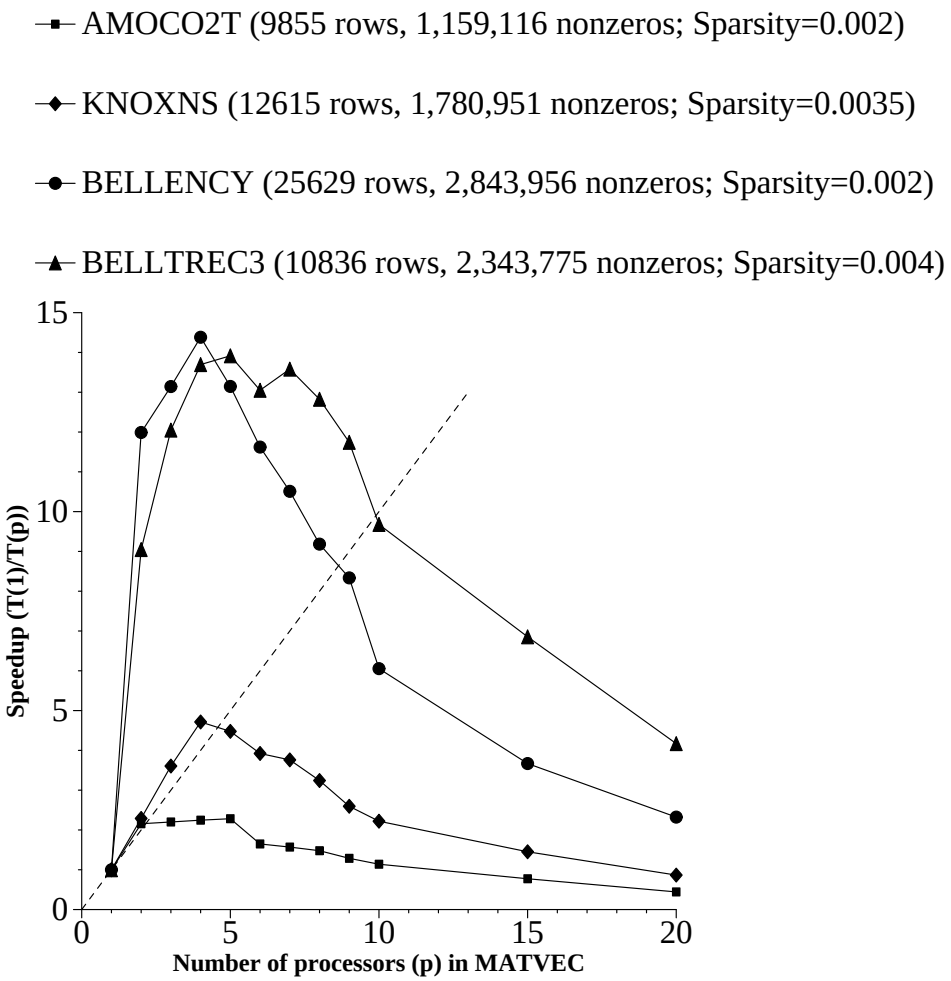


Figure 5.10. Summary of speedups attained using PVM on a network of workstations. The dashed line indicates the theoretical linear speedup.

load with the CSI-MSVD algorithm, and BELLTREC3, with $2,343,775$ non-zeros, produces the largest computational load Thus as expected, the smallest speedups are observed with the matrix AMOCO2T, and the largest are observed with BELLTREC3, with a clear trend within the set $\{$AMOCO2T, KNOXNS, BELLTREC3$\}$.

An interesting variation is observed when BELLENCY is considered. Although BELLENCY has more non-zeros than BELLTREC3, it is the sparser of the two matrices (see Table 4.1). Low sparsity produces a detrimental memory-access pattern which is aggravated by the large value for the number of non-zeros, so that when BELLENCY is stored on one processor, the elapsed wall-clock time T(1) is larger than the corresponding value for BELLTREC3. This causes the speedup T(1)/T(p) for BELLENCY to be larger than the value for BELLTREC3 when $p \leq 5$.

However, since the number of rows in BELLENCY (25,629) is approximately twice that for BELLTREC3 (10,836), the communication overhead for BELLENCY is correspondingly higher. Thus, as p increases, the larger communication overhead has a deteriorating effect on speedup, and BELLTREC3 experiences the higher speedup for $p > 5$. Since speedup is sensitive to the choice of T(1), it cannot be used as an absolute estimate of the degree of parallelization achieved. However, Figure 5.10 provides a heuristic for estimating the upper bound on the number of processors in the virtual machine for this class of matrices.

5.5 Load Balancing

From a performance point of view, the computational load is not fully balanced across the three functional components in the pipeline described in Section 5.3. The computational load on SIGMA and GAMMA is much less than the load on MATVEC for large sparse matrices since the process MATVEC involves the manipulation of large vectors through indirect addressing. Attempts to balance this load dynamically would necessitate the sharing of the matrix across these processors as well and further decrease the granularity of the parallel program. Ideally, it is desirable to have CSI-MSVD distributed across a heterogeneous network of processors on a LAN with the processes involved in MATVEC executing on the fastest processors. However, such a heterogeneous LAN without interference from general Internet traffic was not available for this study, and all experiments were performed on a network of homogeneous processors.

Some scope for static load balancing is possible in the data-parallel computation of the matrix-vector multiplication. In order to obtain an estimate of the balance of load achieved in the current implementation, calls to `getrusage()` [KR92] were made in each process to monitor the system time used. The system time is defined to be the time attributed to the kernel when it executes on behalf of the process [Ste92].

Table 5.5 shows the typical distribution of system time for the processes involved in matrix-vector multiplication. It can be observed that the system time for P_0, the 0^{th} process in MATVEC is much larger than the system time for the other processes. This load imbalance arises from program design. All PVM group operations such as vector reduction require a *root* to be defined in the call so that the result of the reduction operation can be accumulated in the root. In the current version, processor P_0 is defined as the root for all reductions.

A small degree of static load-balancing may be achieved by assigning a different root at each reduction. However, this would impose constraints on the minimum number of processes that need to be involved in the matrix-vector multiplication. As discussed earlier in Section 5.4, the number of processors *p* required to achieve optimal performance is dependent on the size of the matrix. The implementation of such a static load-balancing scheme is thus justifiable only if a substantial improvement in system performance could be attained through the scheme that would compensate for the loss

Table 5.5

System time for each process involved in matrix-vector multiplication when the matrix BELLCRAT partitioned across 10 processors.

Process	System Time
MATVEC(0)	22.38
MATVEC(1)	3.710
MATVEC(2)	3.480
MATVEC(3)	3.820
MATVEC(4)	3.920
MATVEC(5)	3.930
MATVEC(6)	3.960
MATVEC(7)	4.010
MATVEC(8)	4.870
MATVEC(9)	4.110

of flexibility introduced by the constraints on the minimum number of processors.

In order to study the scope for static load balancing in the Chebyshev semi-iteration described in Section 5.3.1, the steps described there will now be examined. It is possible to overlap Step 3 with Step 4 on processor l_2, as well as Steps 6 and Step 7 with Step 8. Step 4 involves a call to `pvm_send()` which is an asynchronous operation (computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor). Step 3 involves a `daxpy` [GL89] which can also be computed efficiently. Thus, the overlap of Steps 3 and 4 produces a slight, though not remarkable, speedup. Similarly the call to `pvm_send()` [GBD+94] involved in Step 8 is asynchronous, and thus is not expected to be a time-critical operation. On the other hand, barriers exist at two points:

- The broadcast of the m-element vector $x^{(k+1)}$ at Step 5 which must be completed before Step 6, and after Step 3. Note that the size of the vector is independent of the number of processors used.
- Step 7 must be completed on processor l_2 so that l_2 is available for the next iteration through Steps 1 and 2.

Tables 5.6 and 5.7 show the times for group communications within PVM, as listed in [SGDM94] where it has been pointed out ([GBD+94] [SGDM94]) that group functions in PVM have been designed to be robust at some cost in efficiency.

Consider a matrix like BELLADIT, with 374 rows, and 82 columns. As has been demonstrated in Section 5.3, the most expensive message-passing operations involve messages of length m, where m is the number of rows of the matrix under consideration.

Table 5.6

Data transfer times (milliseconds)

Network Type	Message Length							
	0	128	512	1K	4K	16K	64K	1M
Ethernet	1.2	1.5	2.1	3.2	7.2	24.5	82.3	1211.2
FDDI	1.2	1.5	1.9	2.5	5.9	16.1	60.3	665.7

Table 5.7

Group operation times (milliseconds). Message size: 1K

Operation Type	Number of Processors				
	2	4	8	16	32
Barrier	2.2	10.5	28.1	53.2	107.2
Broadcast	3.2	5.5	15.9	28.5	65.9
Opt. Bcast	1.2	3.2	11.5	18.2	35.1

Even if calculations were done with single-precision accuracy, the message-passing length in bytes would be $374 \times 4 = 1,496$ bytes. Then, from Table 5.6 the Ethernet data transfer time per m-element message is at least 3.2 milliseconds. Comparing this with the time for matrix-vector multiplication for BELLADIT listed in Table 3.1, it can be seen that the communication overhead is about four to five times larger than the time for the more computationally complex matrix-vector multiplications. Thus the barriers described earlier could be bottlenecks that cannot be circumvented by static load-balancing.

Table 5.8

Distribution of system time (seconds) for all 10 processes involved in matrix-vector multiplication for CSI-MSVD. Ten processors were used for storing the input matrix and to compute the 10 largest singular values and corresponding singular vectors of the matrix BELLCRAT.

Process	Computation	Communication
MATVEC	20.75	381.51
SIGMA	0.04	164.40
GAMMA	0.23	201.78

Table 5.8 lists the distribution of the time spent by the different processes in computation and communication for the matrix BELLCRAT with the matrix divided across 10 processors. This includes the CPU usage for computation in the pipelined processes MATVEC, SIGMA and GAMMA, and the time spent in message-passing for the three processes. Since the sending of messages is designed to be a non-blocking call in PVM, the time spent on waiting for message arrival was found to account for most of the communication overhead. For the processes SIGMA and GAMMA, this arose from waiting on the arrival of moment information from MATVEC. For processes in the MATVEC group the slowest message-passing operations were those involving reduction operations for the group, where the result of the operation had to be accumulated on the root (the 0^{th} processor) and then broadcast to the members of the group. Examples of this type of operation include computing the norm of a vector and Step 4 of Algorithm P_OP.

Table 5.8 indicates that even though the computational load of MATVEC(0) is higher than that of SIGMA and GAMMA, the amount of time spent in communication is much larger than that spent in computation for the three processes so that even when static load-balancing was implemented for matrix-vector multiplication, no significant performance gain was achieved. It did not appear justifiable to impose constraints on the minimum number of processors, and the load-balancing scheme was therefore abandoned.

5.6 Results on the CRAY T3D

The implementation of CSI-MSVD for NOW versions of PVM was ported successfully to the CRAY MPP. Due to the differences in the PVM implementations on the CRAY MPP (see Section 4.1.3), some syntactic modifications had to be made. Only the minimal changes required to successfully port the NOW version were attempted. The emphasis was on developing a portable, modular implementation of Algorithm CSI-MSVD that could be used across multiple platforms. This section tabulates wall-clock times on a 256 node CRAY T3D at the Advanced Computing Laboratory, Los Alamos National Laboratory.

The C compiler used for these experiments was `Cray Standard C Version 4.0.3.2`, and the loader used was `MPPLDR version 10.x`. Compiler optimization for aggressive vectorization, suppression of redundant symbol-tables, and usage of branches instead of jumps to external functions were used.

Table 5.9 summarizes the modifications that were made to port the NOW implementation of the CSI-MSVD algorithm to the CRAY T3D. Since the CRAY MPP does not support `pvm_spawn()` and therefore requires all nodes to run the same executable program, heterogeneity was achieved by using $myid$ in a driver program to determine if the node under consideration would participate in the computations involving MATVEC, SIGMA or GAMMA. The driver program to achieve this is shown in Appendix B.

The wall-clock times for computing the 10-largest singular values and corresponding

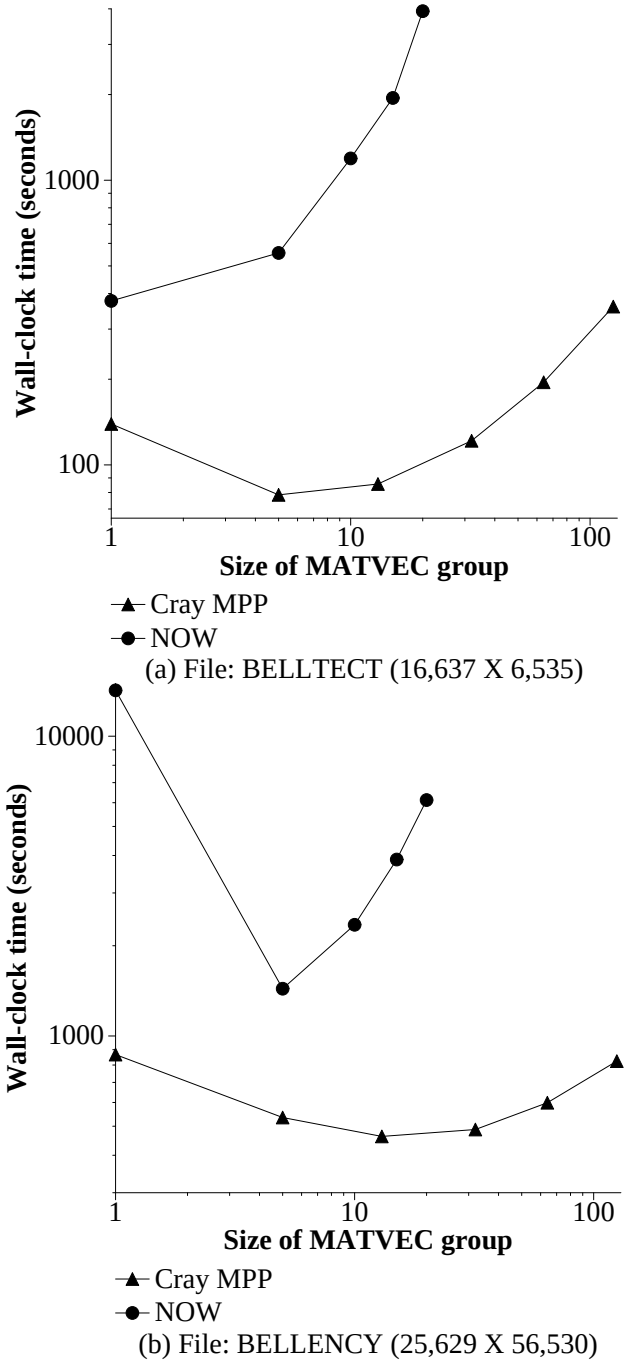


Figure 5.11. Wall-clock times for execution using CRAY T3D's MPP version of PVM, compared with times using PVM on a network of workstations. The 10-largest singular values and corresponding vectors were computed to 10^{-6} accuracy.

Table 5.9

List of modifications made to port the CSI-MSVD algorithm to the CRAY T3D

Variable	purpose	Previous Evaluation:	replaced by:
$nprocs$	number of processes in virtual machine	Read from parameters file	nprocs=pvm_gsize(NULL)
$tids[i]$	array of tids[] of processes in virtual machine	Parent broadcasts array to child processes	tids[i]=pvm_gettid(NULL,i)
$myid$	logical number of process	index within logical group	myid=pvm_get_PE(mytid)

singular vectors on the CRAY T3D are shown in Figure 5.11 along with the corresponding times for execution on a network of workstations. The benefits of improved connectivity can be clearly seen. For the matrix BELLTECT ($16,637 \times 6,535$, with $327,244$ non-zeros) the improvement in execution time through an increase in parallelism is no longer damped by the communication overhead, and a pattern similar to that exhibited in Tables 5.7 and 5.8 can now be seen. Further, for both the matrices BELLTECT and BELLENCY, the rate of increase in communication latency is much lower with the CRAY T3D, so that when the matrix is partitioned across 125 processors, the execution time is less than the execution time on a network of workstations with the matrix stored on only one processor. Also, the minimum execution time is still observed when the size of the MATVEC group is approximately 5, confirming the heuristic established by Figure 5.10. A comparison of this minimum execution time between the networked- and MPP versions of PVM is shown in Table 5.10. It can be seen that the MPP implementation is about 2 to 10 times faster than the networked implementation. The largest differences in execution time are observed for matrices like BELLCRAT and BELLCIST which have the smallest number of non-zeros, indicating that these matrices are most critically affected by communication overhead in NOW environments.

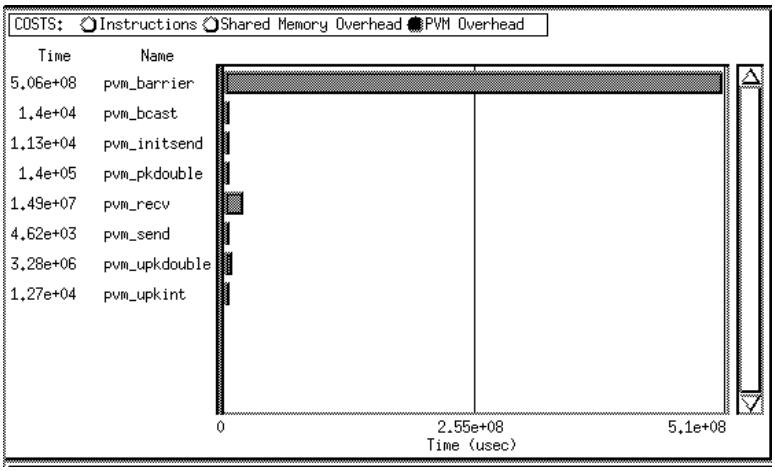
Table 5.10

Comparison of elapsed wall-clock times to compute 10 singular-triplets to 10^{-6} accuracy using CSI-MSVD. Cray MPP and networked versions of PVM were used . The times reported here were obtained with the PVM configurations that result in the minimum execution time for the respective platforms.

File	MPP	NOW
amoco2t	165.831	449.127
bellency	462.322	1437.603
belltrec3	334.928	768.5795
knoxns	266.414	466.568
bellcist	18.934	270.135
belltect	78.478	586.219
bellcrat	19.571	228.691

All times in seconds.

Program performance was also profiled using the Cray MPP Apprentice tool, a window-based performance analysis tool available on Cray MPP systems. The Apprentice tool can be configured to report time spent in each subroutine in performing tasks such as parallel computations, I/O, and communications. One example output of the Apprentice tool is shown in Figures 5.12 and 5.13. Figure 5.12 shows the PVM overhead associated with matrix-vector multiplication when the matrix KNOXNS is distributed across 5 processors, and 10 singular triplets are requested from CSI-MSVD to an accuracy of 10^{-6}. From Figure 5.14 it can be seen that the 47% of the total time



MATVEC

Figure 5.12. PVM overhead within MATVEC as evaluated by Apprentice for CSI-MSVD when computing the singular triplets of KNOXNS using 5 processors for matrix storage.

taken by the program is accounted for by `pvm_recv` alone. The Apprentice tool reports that 179,028,341 μsec (5.08%) are spent in executing "work" instructions, 344,496,344 μsec (9.77%) in loading instruction and data caches 2183,117,126 μsec (61.92%) in waiting on PVM communication and 819,286,910 μsec (23.24%) are spent in executing uninstrumented functions.

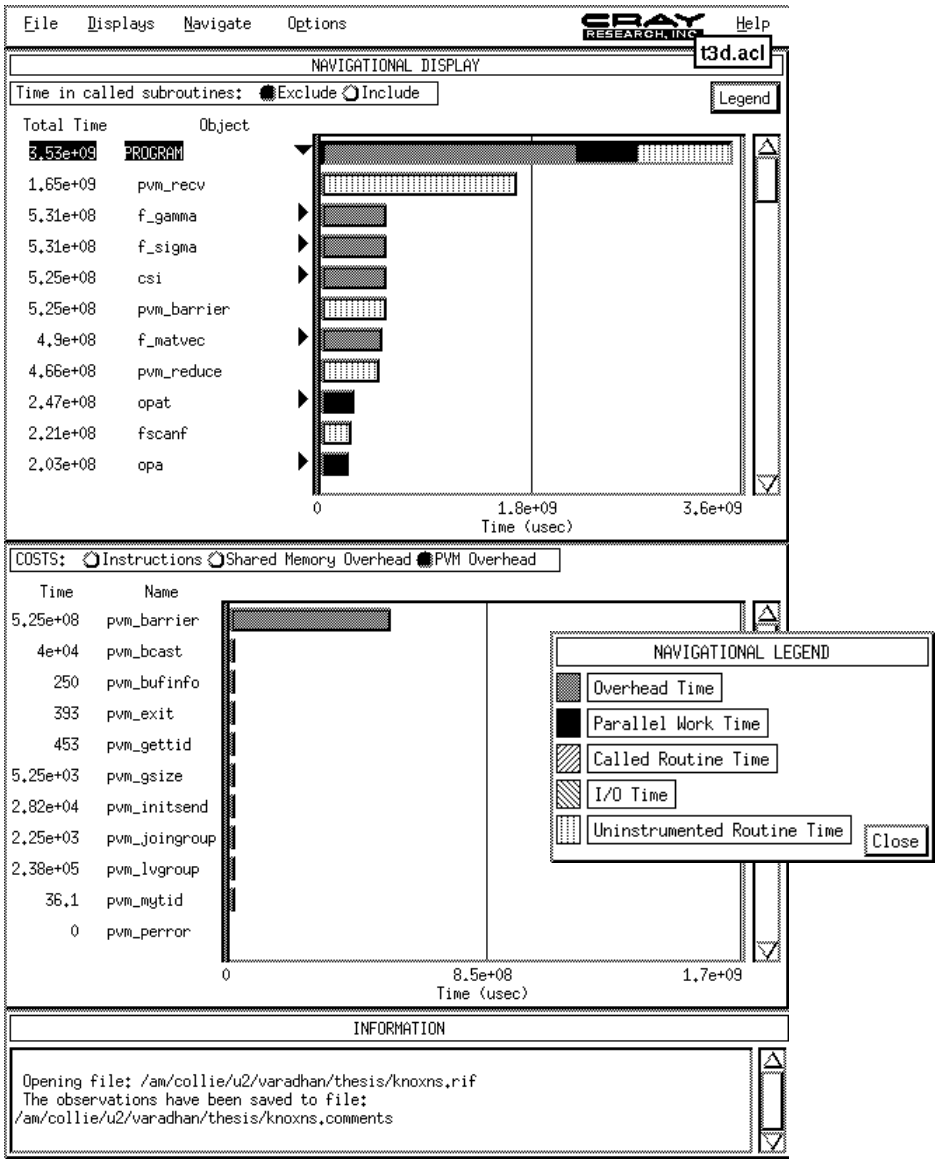
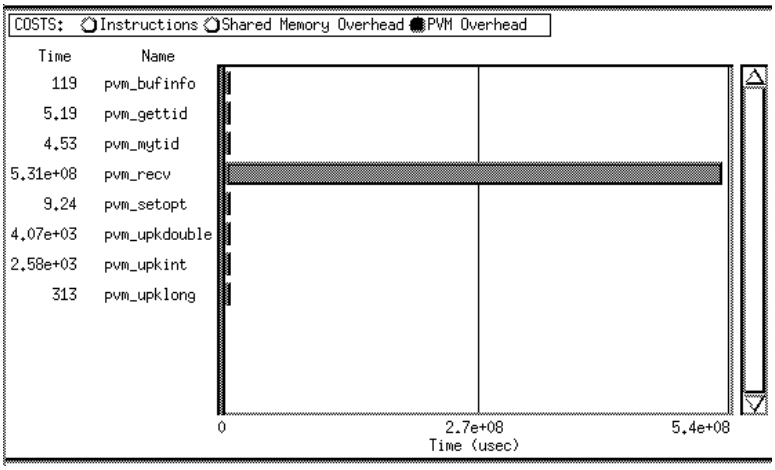


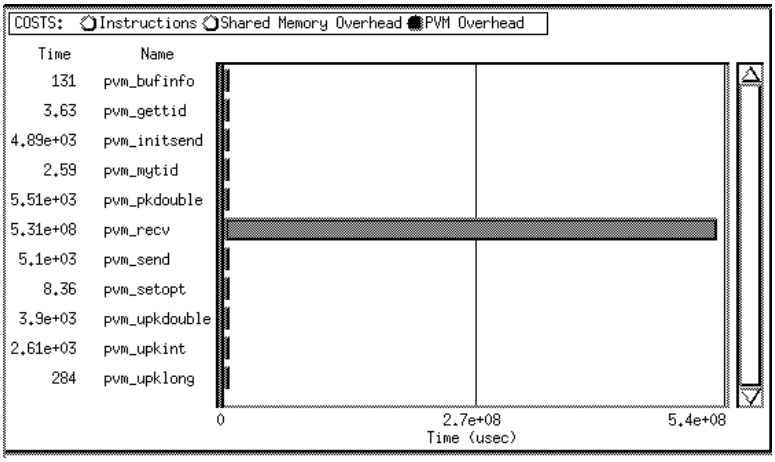
Figure 5.13. Output of Apprentice for CSI-MSVD when computing singular triplets of KNOXNS using 5 processors in the MATVEC group.

An examination of the PVM overhead for each of the functional components MATVEC, SIGMA and GAMMA, is shown in Figures 5.12 and 5.14. The pattern exhibited is similar to the behavior observed for performance on the network of workstations (see Section 5.5). For the processes SIGMA and GAMMA, the time spent in `pvm_recv` (i.e., from waiting for information about moments) accounts almost entirely for the PVM overhead. The cause for this overhead can be understood by examining the process MATVEC which initiates the pipeline described in Section 5.3. The processes involved in the Chebyshev semi-iterative method are all associated with a PVM group, and the operations for CSI are synchronized through calls to `pvm_barrier`. As been pointed out in Section 4.1.3, on the CRAY T3D, operations on groups other than the global group may be expected to be inefficient. Specifically, synchronizations and broadcasts to arbitrary groups are not optimally implemented, and `pvm_barrier` is seen to be the most expensive PVM operation in Figure 5.12. The time spent in `pvm_barrier` (5.06×10^8 μsec) is almost the same as the time spent by SIGMA and GAMMA on `pvm_recv`.

The explicit usage of PVM groups may be avoided by implementing the same functionality directly in between the tasks involved in the Chebyshev semi-iterative method. This may not be desirable since PVM groups are implemented with portability given higher priority than efficiency. Several problems are encountered with attempts to implement group-operations (e.g. broadcasts) efficiently for every architecture. Refinement of PVM group operations is being investigated, and it is possible that future implementations may be able to achieve a better balance between portability and efficiency. For example, PVM 3.3.8 contains several optimized implementations of native group operations. By isolating operations such as vector-reduction, synchronizations and broadcasts to the PVM group functions, it is possible to use vendor-specific implementations for these operations. In the interest of the modularity provided by isolating these logical operations to the class of PVM-defined group operations, no attempts were made to circumvent the usage of PVM groups.



(a) GAMMA



(b) SIGMA

Figure 5.14. PVM overhead within SIGMA and GAMMA as evaluated by Apprentice for computing singular triplets of KNOXNS using 5 processors in MATVEC.

Chapter 6

Conclusions

A procedure, CSI-MSVD, for approximating the singular values and vectors of large, sparse matrices has been presented. When compared to popular single-processor algorithms such as the Lanczos method, CSI-MSVD has the attractive properties of reduced memory requirements, and accelerated convergence to the larger singular value. In addition, the CSI-MSVD algorithm allows both functional and data parallelism, making it suitable for heterogeneous computing environments.

Since CSI-MSVD obtains the SVD by providing a fast algorithm to obtain highly accurate approximations to the eigenvalues of the equivalent eigensystems, it can be used to accelerate the convergence of other eigenvalue solvers. The Chebyshev semi-iterative method involves the computation of matrix-vector products which are also required by Krylov subspace methods. It is thus possible to interleave the computations of CSI-MSVD with other Krylov subspace methods such as block Lanczos methods so that the eigenvalue estimates from CSI-MSVD could be used within a Krylov subspace method to solve a *shifted* eigenvalue problem. The approximations to the eigenvalues provided by CSI-MSVD could also be used within LSI applications to obtain an estimate of the error in the low-rank approximations to the term-document matrix.

The biggest challenge encountered with the CSI-MSVD algorithm is the absence of a relationship between the moments of the iterative method and the eigenvectors of the iteration matrix. The absence of this relationship necessitates the use of external refinement schemes like SYMMLQ when a high accuracy in the singular vector estimates is desired. It has been pointed out in [Saa92] and [Bre80] that the Lanczos procedure is the Stieltjes algorithm for computing a sequence of orthogonal polynomials with respect to the inner product. Deriving the relationship between the Lanczos vectors and the Stieltjes moments could provide a better understanding between the eigenvectors and the moments arising from quadrature formulae, so that convergence to the current eigenvalue and eigenvector could be obtained simultaneously, and more efficient deflation schemes could then be used.

Some software modifications that could be made to improve the performance of the

PVM implementation include optimized implementations of the PVM group functions like broadcast and barrier functions that take into account the topology of the platform available. The message-passing paradigm provided by MPI [DHHW93] is emerging as an efficient, portable standard and future research will include a performance study of MPI implementations of the CSI-MSVD algorithm. It should be noted that the CRAY MPP implementation used for the experiments listed in this dissertation was not optimized. Further improvement in performance is possible by using T3D-specific features like the channeled send/receive, and the shared memory library (`shmem_get()` and `shmem_put()` etc.).

Bibliography

Bibliography

[B+94] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.

[BD95] M. W. Berry and S. T. Dumais. Using linear algebra for intelligent information retrieval. *SIAM Review*, 1995.

[Ber90] M. Berry. *Multiprocessor sparse SVD algorithms and applications*. PhD thesis, The Univeristy of Illinois at Urbana-Champaign, 1990.

[Ber92] M. W. Berry. Large scale singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, 1992.

[Ber94] M. W. Berry. Computing the sparse singular value decomposition via svdpack. In G. Golub, A. Greenbaum, and M. Luskin, editors, *IMA Volumes in Mathematics and its Applications 60*, pages 13–29, 1994.

[BF81] R. L. Burden and J. D. Faires. *Numerical Analysis*. PWS Publishers, Boston, Massachusetts, 1981.

[BG91] M. Berry and G. Golub. Estimating the Largest Singular Values of Large Sparse Matrices via Modified Moments. *Numerical Algorithms*, 1:353–374, 1991.

[Bre80] Claude Brezinski. *Padé-type approximation and general orthogonal polynomials*. Birkhäuser Verlag, 1980.

[Com93] PARKBENCH Committee. Public international benchmarks for parallel computers. Technical Report CS 93-213, University of Tennessee, Knoxville, November 1993.

[CR94a] Inc. Cray Research. *CF90 Fortran Language Reference Manual, publication SR-3902 1.0*. Cray Research, Inc., 655F Lone Oak Drive, Eagan, MN 55121, 5.0 edition, October 1994.

[CR94b] Inc. Cray Research. *UNICOS C Library Reference Manual, publication SR-2080*. Cray Research, Inc., 655F Lone Oak Drive, Eagan, MN 55121, 5.0 edition, October 1994.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, 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, 1990.

[DGL89] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[DHHW93] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level message-passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Lab, February 1993.

[Eij92] V. Eijkhout. Lapack working note 50: Distributed sparse data structures for linear algebra operations. Technical Report CS 92-169, Computer Science Dept, University of Tennessee, Knoxville, TN, 1992.

[Gau82] W. Gautschi. On Generating Orthogonal Polynomials. *SIAM Journal of Statistical and Scientific Computing*, 3(3):289–317, 1982.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[GK89] G. Golub and M. D. Kent. Estimating Eigenvalues for Iterative Methods. *Mathematics of Computation*, 53(188):619–626, 1989.

[GL89] G. Golub and C. Van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, second edition, 1989.

[GLO81] G. Golub, F. T. Luk, and M. L. Overton. A block lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Transactions on Mathematical Software*, 7(2):149–169, 1981.

[Gol74] G. H. Golub. Some uses of the lanczos algorithm in numerical linear algebra. In J. Miller, editor, *Topics in Numerical Analysis*, 1974.

[GR71] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. In *Handbook for automatic computation II, linear algebra*. Springer-Verlag, New York, 1971.

[GV61] G. Golub and R. S. Varga. Chebyshev Semi-iterative Methods, Successive Overrelaxation Iterative methods, and Second Order Richardson Iterative Methods. *Numerische Mathematik*, 3:147–156, 1961.

[Hou64] A. S. Householder. *The Theory of Matrices in Numerical Analysis*. Blaisdell, New York, 1964.

[Hwa93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw Hill, Inc., San Fransisco, 1993.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[KR92] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1992.

[Lan50] C. Lanczos. An iteration method for the solution of the eigenvalue problem. *J. Res. Nat. Bur. Standards*, 45(4):255–282, 1950.

[LSV94] R. B. Lehoucq, D. C. Sorensen, and P. Vu. ARPACK: An implementation of the Implicitly Re-started Arnoldi Iteration that computes some of the eigenvalues and eigenvectors of a large sparse matrix. (*Available from netlib@ornl.gov under the directory Scalapack*), 1994.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice Hall, Englewood Cliffs, NJ, 1980.

[PS75] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal of Numerical Analysis*, 12(4):617–629, 1975.

[Saa90] Y. Saad. Sparskit: A basic tool kit for sparse matrix computation. Technical Report CSRD TR 1029, University of Illinois, Urbana, IL, 1990.

[Saa92] Y. Saad. *Numerical Methods For Large Eigenvalue Problems*. Manchester University Press, Manchester, UK, 1992.

[SGDM94] V. Sunderam, A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–547, 1994.

[Ste73] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[Ste92] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Publishing Company, Massachusetts, 1992.

[Var62] Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1962.

[WBSM94] C. Wu, M. Berry, S. Shivakumar, and J. McLarty. Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition. *Machine Learning*, 1994. To appear.

[Wil62] H. S. Wilf. *Mathematics for the Physical Sciences.* J. Wiley, New York, 1962.

Appendices

Appendix A

Selected Derivations/Proofs

A.1 Error in the Chebyshev Iterates

Consider the i^{th} iterate in a semi-iterative method

$$\begin{array}{c} y^{(m)} \equiv \sum_{j=0}^{m} \nu_j(m) x^{(j)}, \\ \text{where } \sum_{j=0}^{m} \nu_j(m) = 1, \text{ for } m \geq 0. \end{array} \tag{A.1}$$

By definition, the error in the iterate is

$$\begin{aligned} \tilde{\epsilon}^{(m)} &= y^{(m)} - x \\ &= \sum_{j=0}^{m} \nu_j(m) x^{(j)} - x. \end{aligned}$$

The above expression can be rewritten as

$$\sum_{j=0}^{m} \nu_j(m)(x^{(j)} - x) + \sum_{j=0}^{m} \nu_j(m) x - x.$$

Due to the constraints on ν_j imposed by Equation (A.1), this is equivalent to

$$\sum_{j=0}^{m} \nu_j(m)(x^{(j)} - x).$$

By definition, $\epsilon^{(j)} = (x^{(j)} - x)$, i.e., the error in the vector $x^{(j)}$, so that

$$\begin{aligned} \tilde{\epsilon}^{(m)} &= \sum_{j=0}^{m} \nu_j(m) \epsilon^{(j)} \\ &= \sum_{j=0}^{m} \nu_j(m) M^j \epsilon^{(0)}. \end{aligned}$$

Thus, it can be inferred that

$$\tilde{\epsilon}^{(m)} = p_m(M) \epsilon^{(0)}.$$

A.2 Recurrence Relations for $\tilde{p}_m(t)$

The recurrences for Chebyshev polynomials are defined as

$$\begin{array}{c} C_0(t) = 1, C_1(t) = t, \\ C_{m+1}(t) = 2tC_m(t) - C_{m-1}(t), \text{ for } m \geq 1. \end{array} \tag{A.2}$$

Using these recurrences in the definition for the polynomial $\tilde{p}_m(t)$ defined by

$$\tilde{p}_m(t) = \frac{C_m\left(\frac{2t-(b+a)}{b-a}\right)}{C_m\left(\frac{2-(b+a)}{b-a}\right)}, \tag{A.3}$$

it follows that

$$\tilde{p}_{m+1}(t) = \frac{2\frac{2t-(b+a)}{b-a}C_m(\frac{2t-(b+a)}{b-a}) - C_{m-1}(\frac{2t-(b+a)}{b-a})}{C_m(\frac{2-(b+a)}{b-a})}.$$

With $b \equiv \rho(M) \equiv -a$, then $b + a = 0, b - a = 2\rho$, and

$$\tilde{p}_{m+1}(t) = \frac{\frac{2t}{\rho}C_m\left(\frac{t}{\rho}\right) - C_{m-1}\left(\frac{t}{\rho}\right)}{C_{m+1}\left(\frac{1}{\rho}\right)}, \text{ so that}$$

$$C_{m+1}\left(\frac{1}{\rho}\right)\tilde{p}_{m+1}(t) = \frac{2t}{\rho}C_m\left(\frac{t}{\rho}\right) - C_{m-1}\left(\frac{t}{\rho}\right). \tag{A.4}$$

Also, from (A.3)

$$\tilde{p}_m(t) = \frac{C_m(\frac{2t}{2\rho})}{C_m(\frac{2}{2\rho})}, \tag{A.5}$$

which yields

$$C_m\left(\frac{t}{\rho}\right) = C_m\left(\frac{1}{\rho}\right)\tilde{p}_m(t). \tag{A.6}$$

Using Equation (A.6) in Equation (A.4) results in the recurrence

$$C_{m+1}\left(\frac{1}{\rho}\right)\tilde{p}_{m+1}(t) = \frac{2t}{\rho}C_m\left(\frac{1}{\rho}\right)\tilde{p}_m(t) - C_{m-1}\left(\frac{1}{\rho}\right)\tilde{p}_{m-1}(t), \text{ for } m \geq 1.$$

A.3 Obtaining the Moments from the Iterates of the Chebyshev Semi-Iterative Method

The following recurrences for Chebyshev polynomials may be inferred from the definition of Chebyshev polynomials, and basic trigonometric identities [Var62].

$$C_{2k} = 2{C_k}^2 - C_0, \text{ and} \tag{A.7}$$

$$C_{2k+1} = 2C_kC_{k+1} - C_1. \tag{A.8}$$

Then, from Equations (A.7) and (A.5),

$$\tilde{p}_{2k}(x) = \frac{C_{2k}(\frac{x}{\rho})}{C_{2k}(\frac{1}{\rho})} = \frac{2{C_k}^2(\frac{x}{\rho}) - C_0(\frac{x}{\rho})}{C_{2k}(\frac{1}{\rho})} = \frac{2\tilde{p}_k^2(x){C_k}^2(\frac{1}{\rho}) - C_0(\frac{x}{\rho})}{C_{2k}(\frac{1}{\rho})}$$

$$= \frac{\tilde{p}_k^2(x)\{C_{2k}(\frac{1}{\rho}) + C_0(\frac{1}{\rho})\} - C_0(\frac{x}{\rho})}{C_{2k}(\frac{1}{\rho})} \text{ from Equation (A.7).}$$

Since $C_0(\frac{x}{\rho}) = 1$ by definition (Equation (A.2) it follows that

$$\tilde{p}_{2k}(x) = \tilde{p}_k^2(x) + \frac{(\tilde{p}_k^2(x) - 1)}{C_{2k}(\frac{1}{\rho})}. \tag{A.9}$$

From the theory of orthogonal polynomials, and the properties of the Chebyshev semi-iterative method one may write

$$\langle \xi^{(k)}, \xi^{(l)} \rangle = \int \pi_k(\lambda)\pi_l(\lambda)d\alpha(\lambda). \tag{A.10}$$

Hence it follows that

$$\nu_{2k} = \langle \xi^{(k)}, \xi^{(0)} \rangle = \int \tilde{p}_{2k}(\lambda)d\alpha(\lambda),$$

where the polynomials π_k have been chosen to be the polynomials $\tilde{p}_k$ defined in Equation (A.5). From Equation (A.9)

$$\nu_{2k} = \int \tilde{p}_k^2(\lambda)d\alpha(\lambda) + \frac{1}{C_{2k}(\frac{1}{\rho})}\{\int \tilde{p}_k^2(\rho)d\alpha(\rho) - \int d\alpha(\rho)\}$$

$$= \langle \xi^{(k)}, \xi^{(k)} \rangle + \frac{1}{C_{2k}(\frac{1}{\rho})}\{\langle \xi^{(k)}, \xi^{(k)} \rangle - \langle \xi^{(0)}, \xi^{(0)} \rangle\}.$$

Substituting $l = k$ in Equation (A.10) yields

$$\nu_{2k} = \langle \xi^{(k)}, \xi^{(k)} \rangle + \frac{1}{C_{2k}(\frac{1}{\rho})}\{\langle \xi^{(k)}, \xi^{(k)} \rangle - \nu_0\}.$$

Similarly, using Equations (A.8) and (A.5),

$$\begin{aligned}\tilde{p}_{2k+1}(x) = \frac{C_{2k+1}(\frac{x}{\rho})}{C_{2k+1}(\frac{1}{\rho})} &= \frac{2C_k(\frac{x}{\rho})C_{k+1}(\frac{x}{\rho}) - C_1(\frac{x}{\rho})}{C_{2k+1}(\frac{1}{\rho})} \\ &= \frac{2\tilde{p}_k(x)C_k(\frac{1}{\rho})\tilde{p}_{k+1}(x)C_{k+1}(\frac{x}{\rho}) - C_1(\frac{x}{\rho})}{C_{2k+1}(\frac{1}{\rho})} \\ &= \frac{\tilde{p}_k(x)\tilde{p}_{k+1}(x)\{C_{2k+1}(\frac{1}{\rho}) + C_1(\frac{1}{\rho})\} - C_1(\frac{x}{\rho})}{C_{2k+1}(\frac{1}{\rho})}.\end{aligned}$$

By the definition of Chebyshev polynomials given by Equation (A.2), $C_1(x) = x$, so that

$$\tilde{p}_{2k+1}(x) = \tilde{p}_k(x)\tilde{p}_{k+1}(x) + \frac{\tilde{p}_k(x)\tilde{p}_{k+1}(x)}{C_{2k+1}(\frac{1}{\rho})}\frac{1}{\rho} - \frac{x}{\rho}\frac{1}{C_{2k+1}(\frac{1}{\rho})}. \tag{A.11}$$

By definition, the $(k+1)^{th}$ moment is

$$\nu_{2k+1} = \int \tilde{p}_{2k+1}(\lambda)d\alpha(\lambda).$$

Using the expression for $\tilde{p}_{2k+1}$ derived in Equation (A.11) in the above definition for ν_{2k+1}, it follows that

$$\begin{aligned}\nu_{2k+1} = \quad & \int \tilde{p}_k(\lambda)\tilde{p}_{k+1}(\lambda)d\alpha(\lambda) + \frac{1}{\rho C_{2k+1}(\frac{1}{\rho})}\int \tilde{p}_k(\lambda)\tilde{p}_{k+1}(\lambda)d\alpha(\lambda) \\ & -\frac{1}{\rho C_{2k+1}(\frac{1}{\rho})}\int \lambda d\alpha(\lambda).\end{aligned} \tag{A.12}$$

Also,

$$\langle \xi^{(0)}, \xi^{(1)} \rangle = \int \tilde{p}_1(\lambda)d\alpha(\lambda) = \int \frac{C_1(\frac{\lambda}{\rho})}{C_1(\frac{1}{\rho})}d\alpha(\lambda) = \int \lambda d\alpha(\lambda).$$

Using this result to substitute for $\int \lambda d\alpha(\lambda)$ and using Equation (A.10) to substitute for $\int \pi_k(\lambda)\pi_l(\lambda)d\alpha(\lambda)$, Equation (A.12) yields

$$\nu_{2k+1} = \langle \xi^{(k)}, \xi^{(k+1)} \rangle + \frac{1}{\rho C_{2k+1}(\frac{1}{\rho})}\{\langle \xi^{(k)}, \xi^{(k+1)} \rangle - \langle \xi^{(0)}, \xi^{(1)} \rangle\}.$$

Appendix B

Driver program for CRAY MPP version

The NOW version of PVM is composed of four distinct executable programs:

1. `main`, the driver routine, that spawns off the other 3 codes. This executable performs adjustment of the parameters μ and δ at the end of PASS1 and PASS2 defined in Figure 2.4, and determines if the the external refinement procedure should be invoked.
2. `matvec`, the routines involved in the Chebyshev semi-iterative method, defined in Figure 2.3.
3. `sigma`, the routines involved in the computation of σ_{kl}, defined in Figure 2.3.
4. `gamma`, the routines involved in the computation of γ_k, defined in Figure 2.3.

Since the CRAY MPP version requires that all nodes run the same executable program the heterogeneity in the NOW design was simulated through a wrapper program, that uses the logical PE number to determine the functionality of the invoking node. The code to achieve this is shown in Figure B.1.

```
main()
{
 int myid= _my_pe(); /* Use Cray macro to obtain logical PE number */
 char *filename="/usr/tmp/varadhan/params";

  nprocs=pvm_gsize(NULL);
  if (nprocs<4)
      /* Print error message and quit */
  if ((myid>0) && (myid<nprocs-2)){
      /* I am going to work on the Chebyshev iteration*/
      if ((MyAddress = pvm_joingroup("MATVEC"))<0){
      /* Could not join group. Print error message and quit */
      }
  }
  pvm_barrier(NULL,0);

  if (myid==0) f_main(filename);  /* I am the driver program */
  else if (myid==nprocs-2) f_sigma(filename); /* SIGMA */
  else if (myid==nprocs-1) f_gamma(filename); /* GAMMA */
  else f_matvec(filename);

  pvm_barrier(NULL,0);
  pvm_exit();
  exit(1);
}
```

Figure B.1. Wrapper used in Cray MPP implementation of CSI-MSVD

Vita

Sowmini Varadhan was born in Madras, India, July 29, 1969. She received the Bachelor of Technology in Mechanical Engineering at the Indian Institute of Technology, Madras, in August 1990, and the Master of Science in Computer Science from the University of Alabama, Birmingham in December 1991. In January 1992, she entered the Computer Science program at the University of Tenneeessee, and was awarded the Doctor of Philosophy degree in Computer Science in December 1995.