



**An Implementation of  
the Singular Value Decomposition  
on the Connection Machine CM-2**

NHI-ANH CHU

*Signal Processing Branch  
Acoustics Division*

April 11, 1991

# REPORT DOCUMENTATION PAGE

*Form Approved*  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

<b>1. AGENCY USE ONLY (Leave blank)</b>	<b>2. REPORT DATE</b> April 11, 1991	<b>3. REPORT TYPE AND DATES COVERED</b>	
<b>4. TITLE AND SUBTITLE</b> An Implementation of the Singular Value Decomposition on the Connection Machine on CM-2		<b>5. FUNDING NUMBERS</b>	
<b>6. AUTHOR(S)</b> Chu, Nhi-Anh		<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b> NRL Report 9318	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Research Laboratory Washington, DC 20375-5000		<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>		<b>11. SUPPLEMENTARY NOTES</b>	
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution unlimited.		<b>12b. DISTRIBUTION CODE</b>	
<b>13. ABSTRACT (Maximum 200 words)</b>  In modern digital signal processing, the singular value decomposition is increasingly recognized as an important mathematical tool. The true measure of usefulness of such a tool is very much dependent on the ability to compute it at "supercomputer" throughput rates. This report describes an implementation of the singular value decomposition (SVD) on the Connection Machine CM-2 using parallel Fortran. The algorithm is based on Hestenes's, which is a Jacobi iteration in which pairs of rows are rotated to become orthogonal. The Fortran implementation of this algorithm on a full CM-2 is comparable in execution speed to the Linpack implementation on a Convex C220 processor.			
<b>14. SUBJECT TERMS</b> Singular value decomposition      CM-2 SVD    Esprit Connection machine                      Music		<b>15. NUMBER OF PAGES</b>	
		<b>16. PRICE CODE</b>	
<b>17. SECURITY CLASSIFICATION OF REPORT</b> UNCLASSIFIED	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> UNCLASSIFIED	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> UNCLASSIFIED	<b>20. LIMITATION OF ABSTRACT</b> SAR

## CONTENTS

1	Introduction . . . . .	2
2	The Singular Value Decomposition . . . . .	2
3	Previous Work . . . . .	3
4	Implementation of the SVD on the CM-2 . . . . .	3
4.1	Connection machine CM-2 . . . . .	4
4.2	SVD Algorithm . . . . .	4
	One-sided Jacobi rotation . . . . .	5
	Numerical Issues . . . . .	6
	Permutation Scheme . . . . .	7
	Matrix Shape . . . . .	9
4.3	Results . . . . .	9
	REFERENCES . . . . .	10
	APPENDIX A – CM Fortran Code For Subroutine SVD Optimized for $m \simeq n$ . . . . .	13
	APPENDIX B – CM Fortran Code For Subroutine SVD Optimized for $m \gg n$ . . . . .	25
	APPENDIX C – CM Fortran Code For Test Driver . . . . .	37

# AN IMPLEMENTATION OF THE SINGULAR VALUE DECOMPOSITION ON THE CONNECTION MACHINE CM-2

## 1. INTRODUCTION

In recent years, the singular value decomposition (SVD) has become an important tool for modern digital signal processing to find higher resolution and more accurate algorithms to extract underlying signal and system parameters from measurements. The SVD implemented in the LINPACK [1] scientific library was designed for a serial or vector machine and is not directly portable to the Connection Machine, which is of data parallel architecture. A parallel version of the SVD is explored here.

In Section 2, the definition and important properties of the SVD are briefly stated. Section 3 reviews previous implementations of the SVD. Section 4 describes the implementation of the algorithm on the Connection Machine. Details of the algorithm and results are also given.

## 2. THE SINGULAR VALUE DECOMPOSITION

If  $\mathbf{A}$  is a  $m \times n$  matrix of rank  $r$  then there exist real orthogonal matrices  $\mathbf{U} = [\mathbf{u}_1 \mathbf{u}_2 \dots \mathbf{u}_m]$  and  $\mathbf{V} = [\mathbf{v}_1 \mathbf{v}_2 \dots \mathbf{v}_n]$  such that  $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^t$  where

$$\mathbf{\Sigma} = \mathbf{U}^t \mathbf{A} \mathbf{V} = \begin{bmatrix} \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_r) & 0 \\ 0 & 0 \end{bmatrix},$$

$r \leq \min(m, n)$  and  $\sigma_i \geq \sigma_{i+1} > 0$  for  $i = 1, \dots, r-1$ . The  $\sigma_i$  are the singular values of  $\mathbf{A}$  and the corresponding vectors  $\mathbf{u}_i$  and  $\mathbf{v}_i$  are respectively the  $i$ th left and right singular vectors.

The most valuable aspects of the SVD for digital signal processing are in the rank and the dyadic decomposition properties. The rank property says that the singular values can be considered as quantitative measures of the inexact arithmetic measures of the exact mathematic notion of rank. The dyadic decomposition describes a matrix as the sum of  $r$  rank-one matrices of decreasing importance, as measured by the singular values:

Rank property:  $\text{rank}(\mathbf{A}) = r$  where  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$

Dyadic decomposition:  $\mathbf{A} = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^t$

With these properties, the application of the SVD to signal processing and to a wide variety of other systems is often where a linear model is constructed from a sequence of observed data vectors. The complexity of the system is reflected by the rank of the data matrix, and the parameters of the model may often be extracted from certain subspace spanned by singular vectors.

These techniques and their applications to many problems are reviewed in Refs. 2 and 3. For example, the linear least squares method uses the SVD to find a vector of model parameter  $\mathbf{x}$  such that the system output  $\mathbf{A} \mathbf{x}$  is as close to the actual observed output  $\mathbf{b}$  as possible:

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b},$$

the pseudo-inverse  $n \times m$  matrix  $\mathbf{A}^+ = \mathbf{V} \mathbf{\Sigma}^+ \mathbf{U}^t$ , with  $\mathbf{\Sigma}^+ = \begin{bmatrix} \mathbf{\Sigma}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$ .

The SVD and the Generalized SVD [4] serve as the basis for ESPRIT [5], a technique developed by Roy and Kailath for applications such as *direction-of-arrival (DOA)* estimation in which estimates of the spatial location of multiple sources whose radiation is received by an array of sensors are sought. While somewhat less general than the well known MUSIC[6] method, ESPRIT should prove to be more practical because it does not rely on complete knowledge of the antenna gain patterns, and it vastly reduces the amount of calculations. In an example given in Ref. 5, a factor of  $10^5$  reduction of number of multiplications over MUSIC was estimated for a twenty-element sensor array employed to cover 10 signals in an aperture of 2 radians in both elevation and azimuth, with one milliradian resolution.

### 3. PREVIOUS WORK

Theoretically, the SVD may be performed directly following the observation that the singular values  $\sigma_i$  are simply the nonnegative roots of the largest eigenvalues of the matrix  $\mathbf{A} \mathbf{A}^t$ , and the singular vectors  $\mathbf{u}_i$  and  $\mathbf{v}_i$  are the corresponding eigenvectors of  $\mathbf{A} \mathbf{A}^t$  and  $\mathbf{A}^t \mathbf{A}$ . In practice, the loss of numerical precision becomes so severe that smaller singular values are rendered incorrect [7].

The most widely used algorithms used on serial machines are variants of those proposed by Golub and Reinsch [8] and Golub and Kahan [9], in which the given matrix is bidiagonalized, then the QR method is used to compute the singular values of the resultant bidiagonal form. This method is inherently unsuitable for parallel processing [10,11].

The one-sided Jacobi method credited to Hestenes [12,13] and the two-sided variant [13,14] that were superseded by the Golub serial algorithms are apparently suitable to parallel processing because all row-pairs in the matrix may be processed concurrently and each element of each row may also be operated on during the rotations. In the Jacobi iteration process the pair-wise rotations must be done in a particular order for the process to converge. The standard cyclic-by-rows method for Jacobi iteration [15], which involves the sequential processing of row-pairs  $(1, 2), (1, 3), \dots, (1, m), (2, 3), (2, 4), \dots, (2, m), \dots, (m-1, m)$  is not suitable for concurrent processing because of the obvious data dependency. Many other methods to process row-pairs concurrently are reviewed in Ref. 16. The permutation scheme described in this report is akin to the bubble-sort algorithm [17] in which each neighboring pair is transformed by a rotation that leaves the larger (in the norm sense) row on top.

Ewerbring et al. [11] implemented a similar algorithm on the Connection Machine using a parallel variant of Lisp. Their report did not state the execution time. The implementation described here is in Fortran, maps more matrix elements to a processor and uses a different permutation scheme.

## 4. IMPLEMENTATION OF THE SVD ON THE CM-2

The massively parallel computer CM-2 on which the code runs is described in Section 4.1. In Section 4.2, formulae to generate the rotation matrix and the permutation scheme are described in detail. Results are discussed in Section 4.3. The Fortran code is included in the Appendices.

### 4.1. Connection Machine CM-2

Initially, the Connection Machine machine model was a single instruction multiple data (SIMD) array of up to 64K ( $K=1024$ ) bit-serial processors connected by a hypercube bit-serial interconnect network. This paradigm is natural and useful in a number of applications, such as the method of discrete simulation of fluid flow [18] in which each processor is mapped onto a "cell" of the fluid body which interfaces only with a number of neighbor cells.

In the second generation CM machine [19], a 32-bit or 64-bit Weitek floating point arithmetic unit (FPU) was added to each group of 32 processors to provide fast single or double precision floating point capability.

The virtual processor concept allows automatic mapping of problems that require more nodes than are available in the physical machine. In this virtual processor mode, every instruction is executed  $n$  times, where the vp ratio  $n$  is the ratio of number of problem-domain nodes to the actual number of processors. The problem size is thus limited by the amount of memory in each physical processor. At the Naval Research Laboratory Connection Machine Facility, the 16K processor double precision CM-2 has 1 Megabit of memory per processor.

The core of the machine operation is in downloadable microcode. User programming languages include an assembly language called *Paris* and parallel versions of other common high-level languages (HLL): *Lisp*, *C\** and *CM Fortran*. CM Fortran is based on Fortran-8X, which is similar to Fortran-77, augmented with array operations.

The recently introduced slicewise Fortran compiler used for this work employs a different machine model. The machine is presented to the compiler as up to 2048 depth-4 pipelined floating point nodes; each node is a 32-bit or 64-bit processor. For certain problems, this machine model produces compiled codes that are two to three times faster than the fieldwise modeled compiler by streamlining of data in and out of the FPUs, and by using in-place FPU calculations.

The theoretical single precision, peak floating-point performance of a full (64K) CM-2 is 27 GFLOPS, assuming that all of the floating point chips in the machine perform a multiplication and an addition every clock cycle. On a full CM-2 with 32-bit FPU and microcode version 5.0, Levit [20] reported a much lower peak performance of users code without interprocessor communication. This so-called memory-bandwidth-bound peak performance is cited to be 5.17 GFLOPS. For a 16K 64-bit FPU, roughly 800 MFLOPS is expected for the communication-free portions of the code. Grid

communication (between power-of-two interprocessor points) is only 73 MIPS (in terms of number of 32-bit words communicated per second) at vp ratio 1, to 1375 MIPS for adjacent communication at a high vp ratio. The fast Fourier transform (FFT) has been coded and is reported to execute at a sustained rate of more than 1 GFLOPS for very large FFTs on a 64K CM-2 [21].

## 4.2. SVD Algorithm

Consider mapping each element of a real matrix  $\mathbf{A}$  of size  $m \times n$  onto a node on a 2-D array of virtual processors on the CM-2. Transformations of the matrix that require change to the value of each element may take place on all processors simultaneously. The Hestenes one-sided Jacobi iteration algorithm exploits this concurrency.

### *One-Sided Jacobi Rotation*

Denote a matrix  $\mathbf{A}$  in  $\mathfrak{R}^{m \times n}$  as  $\mathbf{A}_2^{\frac{m}{2} \times n}$  to emphasize that 2-element matrix operations are to be performed on the  $\frac{m}{2}$  pairs of the  $n$ -element rows.

In Hestenes's construction of the SVD, two rows of the matrix are rotated to be orthogonal then permuted with other rows to continue the process until all are mutually orthogonal. This is achieved by multiplying each pair of elements from the row pairs by a sequence of rotations  $\{\mathbf{R}_k\}$ ,  $\mathbf{R} = \mathbf{R}_2^{\frac{m}{2} \times 2}$ . The rotated result is stored in a matrix  $\mathbf{H} = \mathbf{H}_2^{\frac{m}{2} \times n}$ .

The product of the rotation matrices is constructed by applying  $\{\mathbf{R}_k\}$  to an initial identity matrix  $\mathbf{I} = \mathbf{I}_2^{\frac{m}{2} \times m}$  during the iterations. The result is kept in matrix  $\mathbf{U}^t$ .

$$\mathbf{H}_2^{\frac{m}{2} \times n} = [\mathbf{U}^t]_2^{\frac{m}{2} \times m} \mathbf{A}_2^{\frac{m}{2} \times n}$$

where

$$[\mathbf{U}^t]_2^{\frac{m}{2} \times m} = \prod_k [\mathbf{R}_k]_{2 \times 2}^{\frac{m}{2} \times m} \mathbf{I}_2^{\frac{m}{2} \times m} \quad (1)$$

Note that in Eq.( 1), the rotation matrices  $\mathbf{R}_k$  are replicated  $m$  times in each row to match the dimensions of  $\mathbf{I}$ .

After normalizing each row  $i$  of  $\mathbf{H}$  by its norm  $\sigma_i$ ,  $\mathbf{H} = [\mathbf{h}_1 \mathbf{h}_2 \dots \mathbf{h}_i \dots \mathbf{h}_m]^t$  may be factored into as a product of a pseudo-diagonal matrix (a diagonal matrix concatenated with null rows)  $\Sigma$  and an orthonormal matrix  $\mathbf{V}^t$ .

$$\mathbf{H} = [\mathbf{h}_1 \mathbf{h}_2 \dots \mathbf{h}_i \dots \mathbf{h}_r \dots \mathbf{h}_m]^t = \Sigma \mathbf{V}^t, \quad (2)$$

where

$$|\mathbf{h}_1| = \sigma_1 \geq |\mathbf{h}_2| = \sigma_2 \dots \geq |\mathbf{h}_r| = \sigma_r > 0;$$

$$\mathbf{V}^t = [\mathbf{v}_1 \mathbf{v}_2 \dots \mathbf{v}_i \dots \mathbf{v}_m]^t,$$

where

$$\mathbf{v}_i = \begin{cases} \mathbf{h}_i / \sigma_i, & i = 1, \dots, r \\ 0, & i > r \end{cases}$$

$$\Sigma = \begin{bmatrix} \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_r) & 0 \\ 0 & 0 \end{bmatrix}.$$

In Eqs. (1) and (2), since  $U^t$  and  $V$  are orthonormal and  $\Sigma$  is pseudo-diagonal, we have the defining equation for the SVD:

$$\Sigma V^t = U^t A. \quad (3)$$

Two rows  $\mathbf{x}$  and  $\mathbf{y}$  are to be rotated into rows  $\mathbf{X}$  and  $\mathbf{Y}$ , respectively by using the premultiplier rotation matrix  $\mathbf{R}$  :

$$\begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix} = \mathbf{R} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}. \quad (4)$$

The first criterion for selecting a value for the rotation angle  $\theta$  is for the resultant rows to be orthogonal:

$$\mathbf{X}^t \mathbf{Y} = 0. \quad (5)$$

Defining

$$\begin{aligned} \alpha &= \mathbf{x}^t \mathbf{y} \\ \beta &= |\mathbf{x}|^2 - |\mathbf{y}|^2 \\ \gamma &= (\alpha^2 + \beta^2)^{\frac{1}{2}}, \end{aligned} \quad (6)$$

and substituting the expansion of the right-hand side of Eq. (4) into Eq. (5), we have

$$\begin{aligned} \tan 2\theta &= \frac{\alpha}{\beta} \\ \cos 2\theta &= \pm \frac{\beta}{\gamma} \\ \sin 2\theta &= \pm \frac{\alpha}{\gamma} = 2 \sin \theta \cos \theta. \end{aligned} \quad (7)$$

The  $\pm$  sign ambiguity corresponds to the  $\frac{\pi}{2}$  ambiguity of  $2\theta$  which can be resolved by an additional constraint that the norms of the rows become more orderly through each rotation in order for the rotation process to converge. In fact, it will soon be shown that the  $+$  sign for  $\cos 2\theta$  and  $\sin 2\theta$  results in a rotation that puts the larger norm on top while the  $-$  sign results in the smaller norm on top.

The rotation matrix coefficients may then be derived from the above using the half-angle trigonometry identities. Thus:

$$\begin{aligned} \cos \theta &= \pm \left( \frac{1 + \cos 2\theta}{2} \right)^{\frac{1}{2}} = \pm \left( \frac{\gamma + \beta}{2\gamma} \right)^{\frac{1}{2}} \\ \sin \theta &= \pm \left( \frac{1 - \cos 2\theta}{2} \right)^{\frac{1}{2}} = \pm \left( \frac{\gamma - \beta}{2\gamma} \right)^{\frac{1}{2}}. \end{aligned} \quad (8)$$

An arbitrary limitation of  $|\theta| \leq \frac{\pi}{4}$  has been found to help in the convergence [22]. In Eq.(8), this limitation is imposed by selecting the positive value for  $\cos \theta$ . The sign of  $\sin \theta$  is the same as that of  $\sin 2\theta$  which is determined by the sign of  $\alpha$ .

To see the significance of the sign for  $\cos 2\theta$  and  $\sin 2\theta$ , calculate the change in norms of the row, say  $\mathbf{x}$ :

$$\begin{aligned} \mathbf{X}^t \mathbf{X} - \mathbf{x}^t \mathbf{x} &= \frac{1}{2} \alpha \sin 2\theta - \beta \sin^2 \theta \\ &= \pm \frac{\alpha^2}{2\gamma} - \beta \frac{\gamma - \beta}{2\gamma} = \frac{\pm \alpha^2 - \gamma \beta + \beta^2}{2\gamma}. \end{aligned} \quad (9)$$

If a + sign is chosen in place of the  $\pm$  in the Eq. ( 9), the change in norm becomes

$$\mathbf{X}^t\mathbf{X} - \mathbf{x}^t\mathbf{x} = \frac{\gamma - \beta}{2}, \quad (10)$$

in which case,  $|\mathbf{x}|$  has increased since the right-hand side of Eq. (10) is greater than zero (The case of  $\gamma = \beta$  is considered separately as discussed below). A similar proof may be carried out for  $\mathbf{y}^t\mathbf{y} - \mathbf{Y}^t\mathbf{Y}$ .

$$\mathbf{X}^t\mathbf{X} - \mathbf{x}^t\mathbf{x} = \mathbf{y}^t\mathbf{y} - \mathbf{Y}^t\mathbf{Y} = \frac{\gamma - \beta}{2} \geq 0. \quad (11)$$

If a - minus is chosen, Eq.( 10) becomes  $-\frac{\gamma+\beta}{2}$ , which is  $\leq 0$ .

### Numerical Issues

Equations (8) should be used carefully. Specifically, avoid the case when  $(\gamma \pm \beta)$  requires a subtraction that results in a loss of accuracy. An improved algorithm to construct the rotation matrix  $\mathbf{R}$  is:

$$\begin{aligned} \text{If } \beta \geq 0, \text{ calculate } \cos \theta &= \left(\frac{\gamma+\beta}{2\gamma}\right)^{\frac{1}{2}} & \text{then calculate } \sin \theta &= \frac{\alpha}{2\gamma \cos \theta}; \\ \text{If } \beta < 0, \text{ calculate } \sin \theta &= \text{sign}(\alpha)\left(\frac{\gamma-\beta}{2\gamma}\right)^{\frac{1}{2}} & \text{then calculate } \cos \theta &= \frac{\alpha}{2\gamma \sin \theta}. \end{aligned} \quad (12)$$

On a digital computer, the orthogonality condition in Eq. (5) can be satisfied to within a quantity equivalent to the norm of a *null* row. The orthogonality condition (based on Ref. 22) is:

$$\mathbf{x}^t\mathbf{y} \leq \delta \min(|\mathbf{x}|, |\mathbf{y}|) \quad (13)$$

where

$$\begin{aligned} \delta &= \epsilon |\mathbf{A}| \\ &= \epsilon \left( \sum_{i=1}^m \sum_{j=1}^n a_{ij}^2 \right)^{\frac{1}{2}}. \end{aligned} \quad (14)$$

The single precision (32-bit) and double precision (64-bit) floating-point machine precisions are  $1.17e - 7$  and  $2.22e - 16$ , respectively.

When the norm of either vector becomes less than  $\delta$ , the rotation becomes meaningless and could be avoided. On a conventional machine, avoiding these calculations may reduce computation time. On the CM-2, however, no saving is expected because the entire array has to be operated on. Taking note of the occurrences of *null norm* and of *orthogonal row-pairs*, however, serves to establish the stopping criterion of the iteration, namely:

Stop when for all row pairs  $(\mathbf{x}_i, \mathbf{y}_i), i = 1, \dots, \frac{m}{2}$ ,

$$|\mathbf{x}_i| \leq \delta \text{ or } |\mathbf{y}_i| \leq \delta \text{ or } (\mathbf{x}_i^t\mathbf{y}_i) \leq \delta \min(|\mathbf{x}_i|, |\mathbf{y}_i|). \quad (15)$$

The calculation of the norms of the rows in Eq. (6) is expensive in terms of execution time on the CM-2 because of the interprocessor communication involved in adding the square of the row elements, each assigned to a virtual processor. Alternatively, new values may be computed from the elements of the rotation matrix  $\mathbf{R}$  and the current values of the square- norms. The loss of

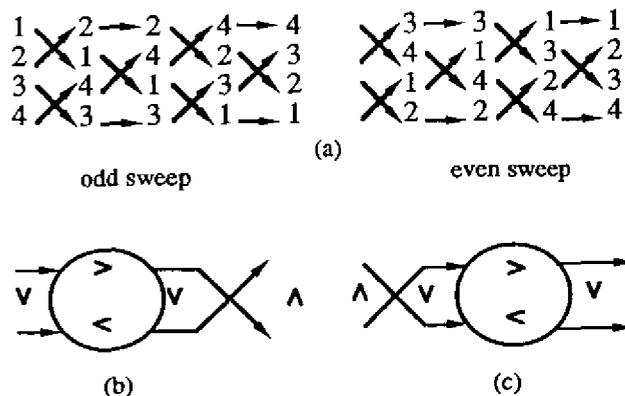


Fig. 1 — Permutation scheme to visit all row pairs in a sweep. (a) The general structure is similar to a bubble-sort using neighbor exchanges. Rotation followed by exchange achieves the required sorting effect that makes the permutation scheme converge to a bubble-sort. (b) The rotation tends to put the larger (norm) row on top, thus in an odd sweep, an exchange is required after rotation to permute the rows. (c) In an even sweep, the inputs must first be exchanged so that the order is enforced after the subsequent rotation.

accuracy in this calculation is sufficiently small for the algorithm to converge—care must be taken to avoid calculating the norm since it would mean taking the square-root of a negative real number. Experiments with the code showed that there was no convergence penalty in terms of number of sweeps.

*Permutation Scheme*

On the Connection Machine, high-speed algorithms must be designed with special care in the assignment of variables that reside on different processors. A general assignment takes on the order of a millisecond to send data between arbitrary processors, while an assignment using specialized communication calls, such as *cshift* to exchange data in a systolic manner, is an order of magnitude faster. Special hardware is used in the high speed execution of a set of specialized communication utilities that includes scan, global, reduce, spread and multispread to implement the broadcast and/or accumulation of values to/from  $n$  processors.

The desirability of the nearest-neighbor systolic communication and the *mesh* layout of the matrix leads naturally to a pairing scheme as illustrated in Fig. 1 (a). For the purpose of illustration, each of the  $m$  rows of the matrix is indicated by an index  $(1, \dots, m)$ — $m$  even. Suppose for a moment that the rows are placed in descending order (from left to right) according to the vector norm, i.e.  $|x_1| \geq |x_2| \geq \dots \geq |x_m|$ . The rows are then exchanged pairwise:  $(1, 2), (3, 4), (5, 6), \dots, (m - 3, m - 2), (m - 1, m)$  to become  $(2, 1), (4, 3), (6, 5), \dots, (m - 2, m - 3), (m, m - 1)$ . In the next iteration, the process is repeated without the first and  $m$ -th rows:  $(2), (1, 4), (3, 6), \dots, (m - 3, m)$ . The iteration repeats even-odd for a total of  $m$  cycles (a sweep) after which all pairing of the rows  $(1, \dots, m)$  have been visited and the norm ordering is reversed, i.e.  $m, m - 1, \dots, 2, 1$ .

If the row norms are not ordered, the same sorting effect described above can also be achieved if each pair is exchanged conditionally on a particular ordering. When the conditional pairwise

exchange is followed by neighbor swap, we have a permutation identical to that in the familiar bubble sort algorithm.

The rotation with matrix  $\mathbf{R}$  (Eq. (4)) also converges into a bubble-sort transformation because of the ordered-norm conditions described in Eq. (11). For each row-pair  $(\mathbf{x}, \mathbf{y})$ , as  $|\mathbf{x}|$  keeps increasing and  $|\mathbf{y}|$  keeps decreasing, eventually  $|\mathbf{x}| \geq |\mathbf{y}|$ . Experience with this SVD algorithm shows that this ordered state is usually achieved in the first couple of sweeps.

Fig. 1(b) illustrates this concept: in each of the  $m$  stages of an odd-numbered sweep, each row-pair is shown to feed into an oval icon representing the premultiplication by matrix  $\mathbf{R}$ . After the rotation, the results are interchanged to realize the bubble-sort.

In Fig. 1(c) which illustrates an even-numbered sweep, each input row-pair is unconditionally exchanged before entering the rotation icon. The necessity of this step is clear if one keeps in mind that the rotation tends (over a few iterations) to make the norm of the upper output larger than that of the lower. Upon entering an even-numbered sweep, if this exchange is not made to put the larger norm row on top, the subsequent rotation would effectively undo the rotation of the preceding odd-numbered sweep. In the even-numbered sweeps, no exchange is required at the output because the output norms ordering is to be reversed from the order produced by the previous rotation (larger norm on top).

An alternative approach is to use a different set of values for  $\mathbf{R}$  such that the rotation will result in a smaller norm on top. This requires a change in Eqs. (12) that involves reversing the sign of  $\beta$  and the order of calculating  $\cos \theta$  and  $\sin \theta$ . The simple mapping of one floating-point processor node per matrix element actually uses only half the resources for computation because the rotation of each row-pair is identical for each of the elements of the pair. The solution used here is to assign a row-pair to one processor-row to make full use of all processor nodes for actual calculations. More significantly, the number of interprocessor communication steps is reduced proportionally; this should significantly reduce execution time. In fact, experiments showed this improved mapping reduced the execution time by an order of magnitude.

### *Matrix Shape*

As indicated at the beginning of Section 4.2., if the matrix  $\mathbf{A}$  is  $m \times n$ , then  $\mathbf{U}$  and  $\mathbf{V}$  are  $m \times m$  and  $n \times n$ , respectively. When  $m \neq n$ , the constructive algorithm described above becomes somewhat cumbersome for the 2-D layout on the CM-2. If  $m$  is slightly more or less than  $n$ , the matrix  $\mathbf{A}$  may be simply padded with  $|m - n|$  null rows or columns. However, if  $m \gg n$ , the algorithm will have to be modified to avoid working directly with the large  $m \times m$  matrix  $\mathbf{U}$ . In this case,  $\mathbf{U}^t$  may be internally processed sequentially as  $\frac{m}{n}$  matrices, each of size  $m \times n$ . See Appendix B for detail.

### 4.3. Results

A CM-Fortran subroutine was written according to the algorithm and requirements presented in the preceding section. Appendix A contains the source codes for the subroutine that is specific for the case of  $m \simeq n$ . Appendix B contains a modified version of Appendix A for the general case of  $m \gg n$ . Appendix C contains source codes for a test driver. The codes were tested on random real matrices. The Connection Machine used was a 16K CM-2 with 64-bit FPU.

Table 1 — Comparing CM-2 execution times with LINPACK. The CM Code was compiled by a slicewise Fortran compiler. The LINPACK codes were run on a very lightly loaded Sun-4/280 and one processor of a Convex C210. The normalized residual error after 12 sweeps was on the order of  $1e - 16$ .

Double Precision				
Size	Machine	# processors	exec time	# sweeps
512	CM-2	16K	280 sec	12
256	CM-2	16K	39 sec	12
512	Convex	1	141 sec	
256	Convex	1	21 sec	
256	Sun-4	1	305 sec	

In Table 1, the execution times for  $m = n$  are compared against the execution times of the LINPACK *dsudc* codes on one processor of a Convex C210 and a Sun/4-280. Both the Convex and Sun/4-280 codes were compiled by using the respective Fortran compiler with optimization; loads were minimal. The codes for the general cases where  $m \gg n$  were slightly slower.

Table 1 shows that the LINPACK implementation on one processor of a Convex C210 is 2 times faster than the CM-Fortran implementation on the 16K CM-2 at  $m = n = 256$  and 512. A full (64K) CM-2 is expected to run between 3 to 4 times faster than a 16K CM-2. It is reasonable to conclude that the CM-2 and Convex implementations are comparable in execution times.

For a great majority of runs on random matrices, the number of row rotations begins to drop off at the end of 8 sweeps, and down to 0 by the end of 12 sweeps. Accuracy was checked by computing

$$Error = \max |A - USV^t|. \quad (16)$$

Errors in the CM-2 runs were on the order of  $1e-14$  for double precision and  $1e-5$  for single precision. After normalizing by the F-norm of  $A$ , these errors were on the order of the respective machine precisions. To gauge the efficiency in the usage of main hardware components (the FPUs), the number of floating point operations in the innerloop of the Fortran code (subroutine *sudcore* in Appendix A) was counted. By using a weight of 4, 2, and 1 for square-root, division, and multiplication/addition/logical respectively, the FLOP count  $Q = 100$  per virtual processor per loop per sweep. This included 20 for the calculation of  $\beta$ ,  $\gamma$  and the conditions for rotation (Eq. (6) and (15)), 40 each for  $\beta \geq 0$  and  $\beta < 0$  for the calculation of  $\cos \theta$ ,  $\sin \theta$  and the subsequent rotations according to Eqs (4) and (12). (On the CM-2, *either-or* code segments are sequentially executed and thus must be counted towards the FPU usage.)

$$Q_{FLOP} = 100 \left( \frac{m}{2} n \right) m I_s \quad (17)$$

where  $I_s$  is the number of sweeps. By using Eq. (17) and the results of Table 1, the throughput rates for the double precision runs are 258 and 288 MFLOPS for matrix size 256x256 and 512x512, respectively, on the 16K CM-2.

Interprocessor communication associated with the calculation of the dot product of the row pairs ( $\alpha$  in Eq. (6)) and the systolic communication steps was timed to be 30% and 22% of the total execution time for  $n = 256$  and 512, respectively. After accounting for the communication time, the performance shown in Table 1 is within a factor of 2 to 2.5 of the peak-memory-bound-performance of the machine.

Table 2 — Execution time per sweep (seconds). The prereleased slicewise Fortran compiler with some unrolling of codes to streamline the inner loop improved execution times by 1.7 and 1.3 times for the smaller vp ratios ( $n = 256$  and 512 respectively).

Double Precision			
Size	Fieldwise	Slicewise, unrolled	Fieldwise, unrolled
512	35	23	30
256	7.5	3	5

Table 2 shows the execution time per sweep in units of seconds for the cases  $m = n = 256$  and 512 by three versions of the CM Fortran code on the 16K CM-2. The best performance was achieved when the matrix was laid out in slicewise mode and the inner loop was unrolled to remove conditionals that fragmented the code.

## 5. REFERENCES

1. J.J. Dongarra et. al. ed., *LINPACK Users' Guide*, 7th printing, (SIAM, Philadelphia, PA 1979).
2. F. Deprettere, *SVD and Signal Processing, Algorithms, Applications and Architectures*, (North Holland 1988).
3. Joos Vandewalle and Bart De Moor, "A Variety of Applications of Singular Value Decomposition in Identification and Signal Processing," in Ref. 2, pp. 43-91.
4. G.H. Golub and C. F. Van Loan, *Matrix Computations*, (John Hopkins University Press, Baltimore, MD, 1989).
5. R. Roy and T. Kailath. "ESPRIT — Estimation of Signal Parameters via Rotational Invariance Techniques," in Ref. 2, pp. 233-265.
6. R. O. Schmidt, "A Signal Subspace Approach to Multiple Emitter Location and Spectral Estimation," PhD thesis, Stanford University, CA, 1981.
7. Ben Noble and James W. Daniel, *Applied Linear Algebra*. (Prentice Hall, 2nd Ed. 1977.)
8. G.H. Golub and C. Reinsh, "Singular Value Decomposition and Least Squares Solutions," *Numer. Math*, 14, pp. 403-420, (1970).
9. G.H. Golub and W. Kahan, "Calculating the Singular Values and Pseudo-inverse of a Matrix," *J. SIAM Ser. B: Numer. Anal.*, 2, pp. 205-224, (1965).
10. Franklin T. Luk, "Computing the Singular-Value Decomposition on the ILLIAC IV," *ACM Transactions on Mathematical Software*, 6 (4), pp. 524-639, (1980).
11. L. Magnus Ewerbring, Franklin T. Luk and Alan H. Ruttenberg, "SVD Computation on the Connection Machine," *IEEE* (1988).
12. Magnus R. Hestenes, "Inversion of Matrices by Biorthogonalization and Related Results," *J. Soc. Indust. Appl. Math.*, 6(1), pp. 51 -91, (1958).

13. J.C. Nash, "A One-sided Transformation Method for the Singular Value Decomposition and Algebraic Eigenproblem," *Compu. J.*, **18**, pp. 74-76, (1975).
14. E.G. Kogbetliantz, "Solution of Linear Equations by Diagonalization of Coefficients Matrix," *Quart. Appl. Math.*, **13**, pp. 123-132, (1955).
15. J.H. Wilkinson, "Note on the Quadratic Convergence of the Cyclic Jacobi Process," *Numerische Mathematik*, **4**, pp. 296-300, (1962).
16. Uwe Schwiegelshohn and Lothar Thiele, "A Systolic Array for Cyclic-by-Rows Jacobi Algorithms," *J. Parallel and Distributed Computing*, **4**, pp. 334-340, (1987).
17. Donald E. Knuth, *The Art of Computer Programming, Vol.3/ Sorting and Searching*, (Addison-Wesley Publishing Co. 1973).
18. "Introduction to Data Level Parallelism," Thinking Machines Corp Technical Report 86.14, April 1986.
19. "Connection Machine Model CM-2 Technical Summary," Thinking Machine Corp. Technical Report HA87-4, April 1987.
20. Creon Levit, "Grid Communication on the Connection Machine: analysis, performance, and improvements," Proc. Conf. Scientific Applications of the Connection Machine, Horst D. Simon, ed. World Scientific, Sep 1988.
21. S. L. Johnsson, R. L. Krawitz, R. Frye and D. MacDonald, "A Radix-2 FFT on the Connection Machine," Proc. Supercomputing '89, pp. 809-819, (1989).
22. Christian Hansen, "Reducing the Number of Sweeps in Hestenes' Method," in Ref. [2] pp. 357-368.

## Appendix A

### CM FORTRAN CODE FOR SUBROUTINE SVD OPTIMIZED FOR $m \simeq n$

*N.B.*—The main subroutine `svd` contains 5 units: `svdcore` contains the main Jacobi rotation codes; `two2one` allocates arrays on the CM two row-elements per processor while `one2two` performs the reverse process; `evaluate1` calculates the S and V matrices; `evaluate2` calculates the residual error. In this *unrolled* version, `svdcore` similar chunks of codes are sequentially repeated 4 times, one slightly different from the others. This is to avoid invoking *if-then* clauses that would otherwise fragment the resulting codes.

```
      subroutine svd (ab,ub,vb,sv,m,n,irank,isweep,eps)
C      Author: Nhi-Anh Chu
C      Connection Machine Facility
C      Code 5153, Naval Research Lab
C      Nov 9 1990
C      Revised Jan 3 1991
C      ab -- 2m x n matrix A, to be decomposed into singular values
C           sv = diag (S) such that A = (U S Vt)
C           ab is returned as (Ut A) where Ut is product of Jacobi rotation
C           matrices on (At A)
C      ub -- 2m x n matrix returned with Ut
C      vb -- 2m x n matrix returned with Vt
C      sv -- 2m-vector returned with diag(S), the singular values of A
C      irank -- integer returned with the rank of A
C      isweep -- integer returned with number of sweeps of the rotations;
C              each sweep orthogonalize every row-pair combinations of A
C      eps -- real number specifying the machine precision, used to determine
C            a "zero"
      integer m, n, irank, isweep
      real ab(2*m, n), ub(2*m, n), vb(2*m, n), sv(2*m)
      real eps, deltas
      real a(m, n), ap(m,n), u(m,n), up(m,n), v(m,n), vp(m,n)
      real a_original(2*m,n)
cmf$ layout a(:news, :news), ap(:news, :news), u(:news, :news)
cmf$ layout up(:news, :news), v(:news, :news), vp(:news, :news)
cmf$ layout ab(:news, :news), ub(:news, :news), vb(:news, :news)
```

```

cmf$ layout sv(:news), a_original(:news, :news)

interface
  subroutine one2two(a,ap,b, m, n)
  integer m, n
  real a(m,n), ap(m,n), b(2*m,n)
cmf$ layout a(:news, :news), ap(:news, :news), b(:news, :news)
end interface

interface
  subroutine svdcore (a, ap, u, up, m, n, irank, isweep, deltas, eps)
  integer irank, isweep, m, n
  real a(m,n), u(m,n),ap(m,n), up(m,n), eps,deltas
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout ap(:news,:news), up(:news, :news)
end interface

interface
  subroutine two2one(a,ap,b,m,n)
  integer m, n
  real a(m,n), ap(m,n), b(2*m,n)
cmf$ layout a(:news, :news), ap(:news, :news), b(:news, :news)
end interface

interface
  subroutine evaluate1 (a,u,v,sv,irank,deltas,m,n)
  integer m, n, irank
  real a(m,n), v(m,n), u(m,n), sv(m), deltas
cmf$ layout a(:news,:news), u(:news,:news), v(:news,:news)
cmf$ layout sv(:news)
end interface

interface
  subroutine evaluate2 (a,u,a_original,m,n)
  integer m, n
  real a(m,n), u(m,n), a_original(m,n)
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout a_original(:news, :news)
end interface
-----C-----
a_original = ab
print*, 'call one2two'
call CM_timer_clear(1)
call CM_timer_start (1)
call one2two(a, ap, ab, m, n)
call one2two(u, up, ub, m, n)
call CM_timer_stop(1)
call CM_timer_print(1)

```

```

print*, 'call svdcore'
call CM_timer_start(1)
call svdcore (a, ap, u, up, m, n, irank, isweep, deltas, eps)
call CM_timer_stop(1)
call CM_timer_print(1)
call CM_timer_start(1)
print*, 'call two2one'
call two2one(a, ap, ab, m, n)
call two2one(u, up, ub, m, n)
call CM_timer_stop(1)
call CM_timer_print(1)
print*, 'call evaluate'
call CM_timer_start(1)
call evaluate1 (ab,ub,vb,sv,irank,deltas,2*m,n)
call evaluate2 (ab,ub,a_original,2*m,n)
call CM_timer_print(1)
print*, '...done svd'
call CM_timer_stop(1)
return
end subroutine svd

subroutine svdcore (a, ap, u, up, m, n, irank, isweep, deltas, eps)
integer m, n, irank, isweep
real a(m,n), u(m,n), ap(m,n), up(m,n), eps, deltas
C Main locals
real alpha (m,n), beta(m,n), gamma(m,n), c(m, n), s(m, n)
real norms(m,n), normsp(m,n)
C scalars to compute convergence criterion
real epss, Fnorms
C temporaries
real atemp(m,n), utemp(m,n), ntemp(m,n), ortho(m,n)
integer row(m,n), col(m,n), irot1(m,n), irot2(m,n)
logical rotate(m,n)
C loop control variables
integer m2, index, i, j, numswEEP, numrotate
C constant
integer sup, sdown
C layout on the connection machine
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout ap(:news,:news), up(:news, :news)
cmf$ layout norms(:news,:news), normsp(:news,:news)
cmf$ layout alpha(:news, :news), beta(:news,:news), gamma(:news,:news)
cmf$ layout utemp(:news,:news), atemp(:news,:news), ntemp(:news,:news)
cmf$ layout c(:news,:news), s(:news,:news)
cmf$ layout row(:news,:news), col(:news,:news)
cmf$ layout irot1(:news, :news), irot2(:news, :news)
cmf$ layout ortho(:news,:news), rotate(:news, :news)
C-----

```

```

C      initialize
      m2 = 2*m
      numswEEP = isweep
      epss = eps*eps
      sdown= +1 !a(k)<---a(k+1)
      sup   = -1 !a(k+1)<-a(k)
      norms = spread (sum(a*a,2),2,n)
      normsp = spread (sum(ap*ap,2),2,n)
      Fnorms  = sum(norms(:,1),1) + sum(normsp(:,1),1)
      deltas = epss*Fnorms
      print*, 'Frobenius norm squared = ', Fnorms
      print*, 'Square of machine precision * Fnorm = ', deltas
      forall (i=1:m, j=1:n) col(i,j) = j
      u = 0.0
      up = 0.0
      forall (i=1:m, j=1:n) row(i,j) = 2*i-1
      where (row.eq.col) u= 1.0
      forall (i=1:m, j=1:n) row(i,j) = 2*i
      where (row.eq.col) up =1.0
      forall (i=1:m, j=1:n) row(i,j) = i
-----
      isweep = 0
100    continue
C      start odd sweep
      isweep = isweep +1
      norms = spread(sum(a*a,2),2,n)
      normsp = spread(sum(ap*ap,2),2,n)
C      unroll loop by 2
      do index = 1, m2, 2
C          start odd index
          alpha = 2*spread(sum(a*ap,2),2,n)
          beta = norms -normsp
          gamma = sqrt((alpha*alpha)+(beta*beta))
          ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
          rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
          where ((beta.ge.0).and.rotate)
              c = sqrt((gamma+beta)/(2.0*gamma))
              s = alpha / (2*gamma*c)
              atemp = -s*a + c*ap
              utemp = -s*u + c*up
              ntemp = s*s*norms + c*c*normsp - alpha*c*s
              ap    = c*a + s*ap
              up    = c*u + s*up
              normsp= c*c*norms + s*s*normsp + alpha*c*s
              a     = atemp
              u     = utemp
              norms = ntemp
          endwhile

```

```

where ((beta.lt.0).and.rotate)
  s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
  c = alpha / (2.0*gamma*s)
  atemp = -s*a + c*ap
  utemp = -s*u + c*up
  ntemp = s*s*norms + c*c*normsp - alpha*c*s
  ap    = c*a + s*ap
  up    = c*u + s*up
  normsp= c*c*norms + s*s*normsp + alpha*c*s
  a     = atemp
  u     = utemp
  norms = ntemp
endwhere
where ((beta.gt.0).and.(.not.rotate))
  atemp = ap
  utemp = up
  ntemp = normsp
  ap    = a
  up    = u
  normsp = norms
  a     = atemp
  u     = utemp
  norms = ntemp
endwhere
C
communicate (a, ap) to/from processors aligned with odd rows
atemp = ap
utemp = up
ntemp = normsp
ap =cshift(a, 1, sdown)
up =cshift(u, 1, sdown)
normsp =cshift(norms, 1, sdown)
a = atemp
u = utemp
norms = ntemp
C
start even index
alpha = 2*spread(sum(a*ap,2),2,n)
beta = norms -normsp
gamma = sqrt((alpha*alpha)+(beta*beta))
ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
        .and.(row.ne.m)
where ((beta.ge.0).and.rotate)
  c = sqrt((gamma+beta)/(2.0*gamma))      !cosine term
  s = alpha / (2*gamma*c)                !sine term
  atemp = -s*a + c*ap
  utemp = -s*u + c*up
  ntemp = s*s*norms + c*c*normsp - alpha*c*s
  ap    = c*a + s*ap

```

```

        up      = c*u + s*up
        normsp= c*c*norms + s*s*normsp + alpha*c*s
        a       = atemp
        u       = utemp
        norms   = ntemp
    endwhile
    where ((beta.lt.0).and.rotate)
        s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
        c = alpha / (2.0*gamma*s)
        atemp = -s*a + c*ap
        utemp = -s*u + c*up
        ntemp = s*s*norms + c*c*normsp - alpha*c*s
        ap    = c*a + s*ap
        up    = c*u + s*up
        normsp= c*c*norms + s*s*normsp + alpha*c*s
        a     = atemp
        u     = utemp
        norms = ntemp
    endwhile
    where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))
        atemp = ap
        utemp = up
        ntemp  = normsp
        ap    = a
        up    = u
        normsp = norms
        a     = atemp
        u     = utemp
        norms = ntemp
    endwhile
C   communicate (a, ap) to/from processors aligned with odd rows
    atemp = a
    utemp = u
    ntemp = norms
    a = cshift(ap, 1, sup)
    u = cshift(up, 1, sup)
    norms = cshift(normsp, 1, sup)
    ap = atemp
    up = utemp
    normsp = ntemp
enddo ! end odd sweep
C   start even sweep
C   number of rotations kept in irot1 and irot2
    isweep = isweep + 1
    irot1 = 0
    irot2 = 0
    do index = 1, m2, 2
C       start odd index

```

```

alpha = 2*spread(sum(a*ap,2),2,n)
beta = normsp - norms
gamma = sqrt((alpha*alpha)+(beta*beta))
ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
where ((beta.ge.0).and.rotate)
    c = sqrt((gamma+beta)/(2.0*gamma))           !cosine term
    s = alpha / (2*gamma*c)                     !sine term
    atemp = s*a + c*ap
    utemp = s*u + c*up
    ntemp = s*s*norms + c*c*normsp + alpha*c*s
    ap    = c*a - s*ap
    up    = c*u - s*up
    normsp= c*c*norms + s*s*normsp - alpha*c*s
    a     = atemp
    u     = utemp
    norms = ntemp
    irot1 = irot1 +1
endwhere
where ((beta.lt.0).and.rotate)
    s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
    c = alpha / (2.0*gamma*s)
    atemp = s*a + c*ap
    utemp = s*u + c*up
    ntemp = s*s*norms + c*c*normsp + alpha*c*s
    ap    = c*a - s*ap
    up    = c*u - s*up
    normsp= c*c*norms + s*s*normsp - alpha*c*s
    a     = atemp
    u     = utemp
    norms = ntemp
    irot1 = irot1 +1
endwhere
where ((beta.gt.0).and.(.not.rotate))
    atemp = ap
    utemp = up
    ntemp = normsp
    ap    = a
    up    = u
    normsp= norms
    a     = atemp
    u     = utemp
    norms = ntemp
endwhere
communicate (a, ap) to/from processors aligned with odd rows
atemp = ap
utemp = up
ntemp = normsp

```

C

```

ap    = cshift(a, 1, sdown)
up    = cshift(u, 1, sdown)
normsp= cshift(norms, 1, sdown)
a     = atemp
u     = utemp
norms = ntemp
C     end odd index of even sweep
C     start even index
alpha = 2*spread(sum(a*ap,2),2,n)
beta  = normsp - norms
gamma = sqrt((alpha*alpha)+(beta*beta))
ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
        .and.(row.ne.m)

where ((beta.ge.0).and.rotate)
  c = sqrt((gamma+beta)/(2.0*gamma))      !cosine term
  s = alpha / (2*gamma*c)                !sine term
  atemp = s*a + c*ap
  utemp = s*u + c*up
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap    = c*a - s*ap
  up    = c*u - s*up
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a     = atemp
  u     = utemp
  norms = ntemp
  irot2 = irot2 +1
endwhere

where ((beta.lt.0).and.rotate)
  s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
  c = alpha / (2.0*gamma*s)
  atemp = s*a + c*ap
  utemp = s*u + c*up
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap    = c*a - s*ap
  up    = c*u - s*up
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a     = atemp
  u     = utemp
  norms = ntemp
  irot2 = irot2 +1
endwhere

where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))
  atemp = ap
  utemp = up
  ntemp = normsp
  ap    = a
  up    = u

```

```

        normsp= norms
        a      = atemp
        u      = utemp
        norms = ntemp
    endwhile
C    communicate (a, ap) to/from processors aligned with odd rows
    atemp = a
    utemp = u
    ntemp = norms
    a      = cshift(ap, 1, sup)
    u      = cshift(up, 1, sup)
    norms = cshift(normsp, 1, sup)
    ap     = atemp
    up     = utemp
    normsp= ntemp
    enddo ! end even sweep
    irot1 = irot1 + irot2
    numrotate = sum(irot1(1:m,1),1)
    print*, ' sweep ', isweep, ' ', numrotate, ' rotations'
    if (numrotate.eq.0) goto 300
    if (isweep.eq.numswEEP) goto 300
    goto 100
300  continue
    print*, 'done rotation...calculating singular values...'
    return
end subroutine svdcore

```

```

subroutine evaluate1 (a,u,v,sv,irank,deltas,m,n)
integer m, n, irank
real a(m,n), v(m,n), u(m,n), sv(m), deltas
integer row(m,n), col(m,n), one(m), ier, i, j
real oned(m), t1(m,m), t2(m,m), t3(m,m)
cmf$ layout a(:news,:news), u(:news,:news), v(:news,:news)
cmf$ layout sv(:news)
cmf$ layout row(:news,:news), col(:news,:news)
cmf$ layout one(:news), oned(:news)
cmf$ layout t1(:news,:news), t2(:news,:news), t3(:news,:news)
C    calculate singular values and rank
    oned = sum(a*a,2)
    sv = sqrt(oned)
    where (oned.gt.deltas)
        one =1
    elsewhere
        one =0
        sv = 0.0
    endwhile
    irank = sum (one(1:m))

```

```

C      calculate v
      t3 = spread (sv,2,m)
      v = 0.0
      where (t3(1:m,1:n).gt.0) v(1:m, 1:n) = a(1:m, 1:n)/t3(1:m, 1:n)
C      debug codes beyond the next statement
      return
C      detailed check
      print*, 'max min of v'
      print *, maxval(v(1:m, 1:n)), minval(v(1:m, 1:n))
C      evaluate VtV
      t1 = 0.0
      t2 = 0.0
      t1(1:m, 1:n) = v(1:m, 1:n)
      t2 = transpose(t1)
      t3 = 0.0
      t3 = matmul (t1, t2)
      forall (i=1:m, j=1:n) row(i,j)= i
      forall (i=1:m, j=1:n) col(i,j)= j
      print*, 'maxval VVt off diagonal ',
              maxval(abs(t3(1:n,1:n)), mask=(row.ne.col))
C      evaluate UUt
      t1 = 0.0
      t2 = 0.0
      t1(1:m, 1:n) = u(1:m, 1:n)
      t2 = transpose(t1)
      t3 = 0.0
      t3 = matmul (t1, t2)
      print*, 'maxval UUt off diagonal ',
              maxval(abs(t3(1:n,1:n)), mask=(row.ne.col))
100   return
      end subroutine evaluate1

      subroutine one2two(a,ap,b,m,n)
      integer m, n
      real a(m,n), ap(m,n), b(2*m,n)
cmf$  layout a(:news, :news), ap(:news, :news), b(:news, :news)
      forall (i=1:m, j=1:n) a(i,j)= b(2*(i-1) +1, j)
      forall (i=1:m, j=1:n) ap(i,j)= b(2*i, j)
      return
      end

      subroutine two2one(a,ap,b,m,n)
      integer m, n
      real a(m,n), ap(m,n), b(2*m,n)
cmf$  layout a(:news, :news), ap(:news, :news), b(:news, :news)
      forall (i=1:2*m-1:2, j=1:n) b(i,j)= a(1+((i-1)/2), j)
      forall (i=2:2*m:2, j=1:n) b(i,j)= ap(1+((i-1)/2),j)

```

```

return
end

subroutine evaluate2 (a,u,a_original,m,n)
integer m, n, irank
real a(m,n), u(m,n), a_original(m,n)
real t1(m,m), t2(m,m), t3(m,m)
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout a_original(:news, :news)
cmf$ layout t1(:news,:news), t2(:news,:news), t3(:news,:news)
C
C evaluate USVt
t1 = 0.0
t2 = 0.0
t2 (1:m,1:n) = a
t1(1:m, 1:n) = u(1:m, 1:n)
t1 = transpose(t1)
t3 = 0.0
t3 = matmul (t1, t2)
t3(1:m,1:n) = t3(1:m, 1:n) - a_original
print*, 'error = max(abs( U S Vt - A )) is ', maxval(abs(t3))
return
end subroutine evaluate2

```

## Appendix B

### CM FORTRAN CODE FOR SUBROUTINE SVD OPTIMIZED FOR THE CASE

$$m \gg n$$

*N.B.*—The main subroutine `svd` contains 7 units: `svdcore` contains the main Jacobi rotation codes; `two2one` allocates arrays on the CM two row-elements per processor while `one2two` performs the reverse process; `evaluate1` calculates the **S** and **V** matrices; `evaluate2` calculates the residual error; `p2one` allocates an  $m \times m$  array into  $\frac{m}{n}$  arrays, each  $\frac{m}{2} \times n$ , while `one2p` performs the reverse. In this *unrolled* version of `svdcore` similar chunks of codes are repeated 4 times, each slightly different from the other. This is to avoid invoking *if-then* clauses that would otherwise fragment the resulting codes. Further, calculations involving the matrix **U** is carried out in a  $\frac{m}{n}$ -times do loop.

```
subroutine svd (ab,ub,vb,sv,p,m,n,irank,isweep,eps)
C   Author: Nhi-Anh Chu
C   Connection Machine Facility
C   Code 5153, Naval Research Lab
C   Nov 9 1990
C   Revised Jan 3 1991
C   p = 2*int(m/n)
C   ab -- 2m x n matrix A, to be decomposed into singular values
C         sv = diag (S) such that A = (U S Vt)
C         ab is returned as (Ut A) where Ut is product of Jacobi rotation
C         matrices on (At A)
C   ub -- 2m x 2m matrix returned with Ut
C   vb -- 2m x n matrix returned with Vt
C   sv -- 2m-vector returned with diag(S), the singular values of A
C   irank -- integer returned with the rank of A
C   isweep -- integer returned with number of sweeps of the rotations;
C             each sweep orthogonalize every row-pair combinations of A
C   eps -- real number specifying the machine precision, used to determine
C          a "zero"
integer p, m, n, irank, isweep
real ab(2*m, n), ub(2*m, p*n), vb(2*m, n), sv(2*m)
real eps, deltas
real a(m, n), ap(m,n), u(p,m,n), up(p,m,n), v(m,n), vp(m,n)
```

```

real a_original(2*m,n)
cmf$  layout a(:news, :news), ap(:news, :news), u(:serial, :news, :news)
cmf$  layout up(:serial, :news, :news), v(:news, :news), vp(:news, :news)
cmf$  layout ab(:news, :news), ub(:news, :news), vb(:news, :news)
cmf$  layout sv(:news), a_original(:news, :news)

interface
  subroutine one2two(a,ap,b, m, n)
    integer m, n
    real a(m,n), ap(m,n), b(2*m,n)
cmf$  layout a(:news, :news), ap(:news, :news), b(:news, :news)
  end interface

interface
  subroutine svdcore (a, ap, u, up, p, m, n, irank, isweep, deltas, eps)
    integer irank, isweep, p, m, n
    real a(m,n), u(p,m,n),ap(m,n), up(p,m,n), eps,deltas
cmf$  layout a(:news,:news), u(:serial, :news,:news)
cmf$  layout ap(:news,:news), up(:serial, :news, :news)
  end interface

interface
  subroutine two2one(a,ap,b,m,n)
    integer m, n
    real a(m,n), ap(m,n), b(2*m,n)
cmf$  layout a(:news, :news), ap(:news, :news), b(:news, :news)
  end interface

interface
  subroutine p2one(u,up,ub,p,m,n)
    integer p, m, n
    real u(p,m,n), up(p,m,n), ub(2*m,p*n)
cmf$  layout u(:serial, :news, :news), up(:serial, :news, :news)
cmf$  layout ub(:news, :news)
  end interface

interface
  subroutine one2p(u,up,ub,p,m,n)
    integer p, m, n, k
    real u(p,m,n), up(p,m,n), ub(2*m,p*n)
cmf$  layout u(:serial, :news, :news), up(:serial, :news, :news)
cmf$  layout ub(:news, :news)
  end interface

interface
  subroutine evaluate1 (a,u,v,sv,irank,deltas,m,n)
    integer m, n, irank
    real a(m,n), v(m,n), u(m,m), sv(n), deltas

```

```
cmf$ layout a(:news,:news), u(:news,:news), v(:news,:news)
cmf$ layout sv(:news)
end interface
```

```
interface
  subroutine evaluate2 (a,u,a_original,m,n)
    integer m, n
    real a(m,n), u(m,n), a_original(m,n)
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout a_original(:news, :news)
end interface
```

C-----

```
C make sure that p is 2*m/n
if (p.ne.(2*m/n)) stop 'p must be equal to m/n'
a_original = ab
print*, 'call one2two'
call CM_timer_clear(1)
call CM_timer_start (1)
call one2two(a, ap, ab, m, n)
call one2p(u, up, ub, p, m, n)
call CM_timer_stop(1)
call CM_timer_print(1)
print*, 'call svdcore'
call CM_timer_start(1)
call svdcore (a, ap, u, up, p, m, n, irank, isweep, deltas, eps)
call CM_timer_stop(1)
call CM_timer_print(1)
call CM_timer_start(1)
print*, 'call two2one'
call two2one(a, ap, ab, m, n)
call p2one(u, up, ub, p, m, n)
call CM_timer_stop(1)
call CM_timer_print(1)
print*, 'call evaluate'
call CM_timer_start(1)
call evaluate1 (ab,ub,vb,sv,irank,deltas,2*m,n)
call evaluate2 (ab,ub,a_original,2*m,n)
call CM_timer_print(1)
print*, '...done svd'
call CM_timer_stop(1)
return
end subroutine svd

subroutine svdcore (a, ap, u, up, p, m, n, irank, isweep, deltas, eps)
  integer p, m, n, irank, isweep
  real a(m,n), u(p,m,n), ap(m,n), up(p,m,n), eps, deltas
C Main locals
  real alpha (m,n), beta(m,n), gamma(m,n), c(m, n), s(m, n)
```

```

      real norms(m,n), normsp(m,n)
C scalars to compute convergence criterion
      real epss, Fnorms
C temporaries
      real atemp(m,n), utemp(p, m,n), ntemp(m,n), ortho(m,n)
      integer row(m,n), col(m,n), irot1(m,n), irot2(m,n)
      logical rotate(m,n)
C loop control variables
      integer m2, index, k, i, j, numsweep, numrotate
C constant
      integer sup, sdown
C layout on the connection machine
cmf$ layout a(:news,:news), u(:serial,:news,:news)
cmf$ layout ap(:news,:news), up(:serial, :news, :news)
cmf$ layout norms(:news,:news), normsp(:news,:news)
cmf$ layout alpha(:news, :news), beta(:news,:news), gamma(:news,:news)
cmf$ layout utemp(:serial, :news,:news)
cmf$ layout atemp(:news,:news), ntemp(:news,:news)
cmf$ layout c(:news,:news), s(:news,:news)
cmf$ layout row(:news,:news), col(:news,:news)
cmf$ layout irot1(:news, :news), irot2(:news, :news)
cmf$ layout ortho(:news,:news), rotate(:news, :news)
C-----
C initialize
m2 = 2*m
numsweep = isweep
if (m2.lt.n) then
  print*, 'There must be no more columns than rows. '
  print*, 'Transpose the matrix'
  irank = 0
  stop
end if
epss = eps*eps
sdown= +1 !a(k)<---a(k+1)
sup = -1 !a(k+1)<-a(k)
norms = spread (sum(a*a,2),2,n)
normsp = spread (sum(ap*ap,2),2,n)
Fnorms = sum(norms(:,1),1) + sum(normsp(:,1),1)
deltas = epss*Fnorms
print*, 'Frobenius norm squared = ', Fnorms
print*, 'Square of machine precision * Fnorm = ', deltas
u = 0.0
up = 0.0
do k = 1, p
  forall (i=1:m, j=1:n) col(i,j) = j+ (k-1)*n
  forall (i=1:m, j=1:n) row(i,j) = 2*i-1
  where (row.eq.col) u(k, :, :) = 1.0
  forall (i=1:m, j=1:n) row(i,j) = 2*i

```

```

      where (row.eq.col) up(k,::) =1.0
    enddo
    forall (i=1:m, j=1:n) row(i,j) = i
C-----
      isweep = 0
100  continue
C start odd sweep
      isweep = isweep +1
      norms = spread(sum(a*a,2),2,n)
      normsp = spread(sum(ap*ap,2),2,n)
C unroll loop by 2
      do index = 1, m2, 2
C start odd index
      alpha = 2*spread(sum(a*ap,2),2,n)
      beta = norms -normsp
      gamma = sqrt((alpha*alpha)+(beta*beta))
      ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
      rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
      where ((beta.ge.0).and.rotate)
        c = sqrt((gamma+beta)/(2.0*gamma))
        s = alpha / (2*gamma*c)
        atemp = -s*a + c*ap
        ntemp = s*s*norms + c*c*normsp - alpha*c*s
        ap = c*a + s*ap
        normsp= c*c*norms + s*s*normsp + alpha*c*s
        a = atemp
        norms = ntemp
      endwhere
      where ((beta.lt.0).and.rotate)
        s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
        c = alpha / (2.0*gamma*s)
        atemp = -s*a + c*ap
        ntemp = s*s*norms + c*c*normsp - alpha*c*s
        ap = c*a + s*ap
        normsp= c*c*norms + s*s*normsp + alpha*c*s
        a = atemp
        norms = ntemp
      endwhere
      do k=1,p
      where (rotate)
        utemp(k,::) = -s*u(k,::) + c*up(k,::)
        up(k,::) = c*u(k,::) + s*up(k,::)
        u(k,::) = utemp(k,::)
      endwhere
      enddo
      where ((beta.gt.0).and.(.not.rotate))
        atemp = ap
        ntemp = normsp

```

```

        ap    = a
        normsp = norms
        a      = atemp
        norms  = ntemp
    endwhile
    do k = 1, p
        where((beta.gt.0).and.(.not.rotate))
            utemp(k,:,:) = up(k,:,:)
            up(k,:,:)    = u(k,:,:)
            u(k,:,:)     = utemp(k,:,:)
        endwhile
    enddo
C   communicate (a, ap) to/from processors aligned with odd rows
    atemp = ap
    ntemp = normsp
    ap = cshift(a, 1, sdown)
    normsp = cshift(norms, 1, sdown)
    a = atemp
    norms = ntemp
    do k = 1, p
        utemp(k,:,:) = up(k,:,:)
        up(k,:,:) = cshift(u(k,:,:), 1, sdown)
        u(k,:,:) = utemp(k,:,:)
    enddo
C   start even index
    alpha = 2*spread(sum(a*ap,2),2,n)
    beta = norms - normsp
    gamma = sqrt((alpha*alpha)+(beta*beta))
    ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
    rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
            .and.(row.ne.m)
    where ((beta.ge.0).and.rotate)
        c = sqrt((gamma+beta)/(2.0*gamma))           !cosine term
        s = alpha / (2*gamma*c)                       !sine term
        atemp = -s*a + c*ap
        ntemp = s*s*norms + c*c*normsp - alpha*c*s
        ap    = c*a + s*ap
        normsp = c*c*norms + s*s*normsp + alpha*c*s
        a      = atemp
        norms  = ntemp
    endwhile
    where ((beta.lt.0).and.rotate)
        s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
        c = alpha / (2.0*gamma*s)
        atemp = -s*a + c*ap
        ntemp = s*s*norms + c*c*normsp - alpha*c*s
        ap    = c*a + s*ap
        normsp = c*c*norms + s*s*normsp + alpha*c*s

```

```

        a      = atemp
        norms = ntemp
    endwhile
do k=1,p
    where (rotate)
        utemp(k,:,:) = -s*u(k,:,:) + c*up(k,:,:)
        up(k,:,:)    = c*u(k,:,:) + s*up(k,:,:)
        u(k,:,:)     = utemp(k,:,:)
    endwhile
enddo
where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))
    atemp = ap
    ntemp = normsp
    ap    = a
    normsp = norms
    a     = atemp
    norms = ntemp
endwhere
do k=1,p
    where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))
        utemp(k,:,:) = up(k,:,:)
        up(k,:,:)    = u(k,:,:)
        u(k,:,:)     = utemp(k,:,:)
    endwhile
enddo
C    communicate (a, ap) to/from processors aligned with odd rows
    atemp = a
    ntemp = norms
    a = cshift(ap, 1, sup)
    norms = cshift(normsp, 1, sup)
    ap = atemp
    normsp = ntemp
do k=1,p
    utemp(k,:,:) = u(k,:,:)
    u(k,:,:) = cshift(up(k,:,:), 1, sup)
    up(k,:,:) = utemp(k,:,:)
enddo
enddo ! end odd sweep
C    start even sweep
C    number of rotations kept in irot1 and irot2
    isweep = isweep + 1
    irot1 = 0
    irot2 = 0
do index = 1, m2, 2
C    start odd index
    alpha = 2*spread(sum(a*ap,2),2,n)
    beta = normsp - norms
    gamma = sqrt((alpha*alpha)+(beta*beta))

```

```

ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
where ((beta.ge.0).and.rotate)
  c = sqrt((gamma+beta)/(2.0*gamma))           !cosine term
  s = alpha / (2*gamma*c)                     !sine term
  atemp = s*a + c*ap
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap     = c*a - s*ap
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a      = atemp
  norms  = ntemp
  irot1  = irot1 +1
endwhere
where ((beta.lt.0).and.rotate)
  s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
  c = alpha / (2.0*gamma*s)
  atemp = s*a + c*ap
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap     = c*a - s*ap
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a      = atemp
  norms  = ntemp
  irot1  = irot1 +1
endwhere
do k=1,p
  where (rotate)
    utemp(k,::) = s*u(k,::) + c*up(k,::)
    up(k,::)    = c*u(k,::) - s*up(k,::)
    u(k,::)     = utemp(k,::)
  endwhere
enddo
where ((beta.gt.0).and.(.not.rotate))
  atemp = ap
  ntemp = normsp
  ap     = a
  normsp= norms
  a      = atemp
  norms  = ntemp
endwhere
do k =1,p
  where ((beta.gt.0).and.(.not.rotate))
    utemp(k,::) = up(k,::)
    up(k,::)    = u(k,::)
    u(k,::)     = utemp(k,::)
  endwhere
enddo
C communicate (a, ap) to/from processors aligned with odd rows
atemp = ap

```

```

ntemp = normsp
ap    = cshift(a, 1, sdown)
normsp= cshift(norms, 1, sdown)
a     = atemp
norms = ntemp
do k=1,p
  utemp(k,::) = up(k,::)
  up(k,::) = cshift(u(k,::), 1, sdown)
  u(k,::) = utemp(k,::)
enddo
C   end odd index of even sweep
C   start even index
alpha = 2*spread(sum(a*ap,2),2,n)
beta  = normsp - norms
gamma = sqrt((alpha*alpha)+(beta*beta))
ortho = 0.25*alpha*alpha - deltas*min(norms, normsp)
rotate = (norms.ge.deltas).and.(normsp.ge.deltas).and.(ortho.ge.0)
      .and.(row.ne.m)

where ((beta.ge.0).and.rotate)
  c = sqrt((gamma+beta)/(2.0*gamma))      !cosine term
  s = alpha / (2*gamma*c)                !sine term
  atemp = s*a + c*ap
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap    = c*a - s*ap
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a     = atemp
  norms = ntemp
  irot2 = irot2 +1
endwhere
where ((beta.lt.0).and.rotate)
  s = sign(sqrt((gamma-beta)/(2.0*gamma)) , alpha)
  c = alpha / (2.0*gamma*s)
  atemp = s*a + c*ap
  ntemp = s*s*norms + c*c*normsp + alpha*c*s
  ap    = c*a - s*ap
  normsp= c*c*norms + s*s*normsp - alpha*c*s
  a     = atemp
  norms = ntemp
  irot2 = irot2 +1
endwhere
do k=1,p
  where (rotate)
    utemp(k,::) = s*u(k,::) + c*up(k,::)
    up(k,::)    = c*u(k,::) - s*up(k,::)
    u(k,::)     = utemp(k,::)
  endwhere
enddo
where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))

```

```

        atemp = ap
        ntemp = normsp
        ap     = a
        normsp= norms
        a      = atemp
        norms  = ntemp
    endwhile
    do k=1,p
        where ((beta.gt.0).and.(.not.rotate).and.(row.ne.m))
            utemp(k,:,:) = up(k,:,:)
            up(k,:,:)    = u(k,:,:)
            u(k,:,:)     = utemp(k,:,:)
        endwhile
    enddo
C   communicate (a, ap) to/from processors aligned with odd rows
    atemp = a
    ntemp = norms
    a     = cshift(ap, 1, sup)
    norms = cshift(normsp, 1, sup)
    ap    = atemp
    normsp= ntemp
    do k=1,p
        utemp(k,:,:) = u(k,:,:)
        u(k,:,:)     = cshift(up(k,:,:), 1, sup)
        up(k,:,:)    = utemp(k,:,:)
    enddo
    enddo ! end even sweep
    irot1 = irot1 + irot2
    numrotate = sum(irot1(1:m,1),1)
    print*, ' sweep ', isweep, ' ', numrotate, ' rotations'
    if (numrotate.eq.0) goto 300
    if (isweep.eq.numswep) goto 300
    goto 100
300 continue
    print*, 'done rotation...calculating singular values...'
    return
end subroutine svdcore

subroutine evaluate1 (a,u,v,sv,irank,deltas,m,n)
integer m, n, irank
real a(m,n), v(n,n), u(m,m), sv(n), deltas
integer row(m,m), col(m,m), i, j
real t1(m,m), t2(n,n), t3(n,n)
cmf$ layout a(:news,:news), u(:news,:news), v(:news,:news)
cmf$ layout sv(:news)
cmf$ layout row(:news,:news), col(:news,:news)
cmf$ layout t1(:news,:news), t2(:news,:news), t3(:news,:news)

```

```

C      calculate singular values and rank
      t3 = a(1:n, 1:n)
      sv = sqrt(sum(t3*t3,2))
      irank = count(sv.gt.sqrt(deltas))
C      calculate v
      t2 = spread (sv,2,n)
      v = 0.0
      where (t2.gt.0) v = t3/t2
C      debug codes beyond the next statement
C      return
C      detailed check
      print*, 'max min of v' ,maxval(v), minval(v)
C      evaluate VtV
      t2 = matmul (transpose(v), v)
      forall (i=1:m, j=1:m) row(i,j)= i
      forall (i=1:m, j=1:m) col(i,j)= j
      print*, 'maxval VVt off diagonal ', maxval(abs(t2),
      .           mask=(row(1:n,1:n).ne.col(1:n,1:n)))
C      evaluate UUt
      t1 = matmul (transpose(u), u)
      print*, 'maxval UUt off diagonal ',
      .           maxval(abs(t1), mask=(row.ne.col))
100      return
      end subroutine evaluate1

      subroutine one2two(a,ap,b,m,n)
      integer m, n
      real a(m,n), ap(m,n), b(2*m,n)
cmf$      layout a(:news, :news), ap(:news, :news), b(:news, :news)
      forall (i=1:m, j=1:n) a(i,j)= b(2*(i-1) +1, j)
      forall (i=1:m, j=1:n) ap(i,j)= b(2*i, j)
      return
      end

      subroutine two2one(a,ap,b,m,n)
      integer m, n
      real a(m,n), ap(m,n), b(2*m,n)
cmf$      layout a(:news, :news), ap(:news, :news), b(:news, :news)
      forall (i=1:2*m-1:2, j=1:n) b(i,j)= a(1+((i-1)/2), j)
      forall (i=2:2*m:2, j=1:n) b(i,j)= ap(1+((i-1)/2),j)
      return
      end

      subroutine one2p(u,up,ub,p,m,n)
      integer p, m, n, k
      real u(p,m,n), up(p,m,n), ub(2*m,p*n)

```

```

cmf$ layout u(:serial, :news, :news), up(:serial, :news, :news)
cmf$ layout ub(:news, :news)
do k =1,p
  forall (i=1:2*m-1:2, j=k:(k+n-1)) ub(i,j)= u(k,1+((i-1)/2), j)
  forall (i=2:2*m:2, j=k:(k+n-1)) ub(i,j)= up(k,1+((i-1)/2),j)
enddo
return
end

subroutine p2one(u,up,ub,p,m,n)
integer p, m, n, k
real u(p,m,n), up(p,m,n), ub(2*m,p*n)
cmf$ layout u(:serial, :news, :news), up(:serial, :news, :news)
cmf$ layout ub(:news, :news)
do k =1,p
  forall (i=1:2*m-1:2, j=1:n) ub(i,j+(k-1)*n)= u(k,1+((i-1)/2), j)
  forall (i=2:2*m:2, j=1:n) ub(i,j+(k-1)*n)= up(k,1+((i-1)/2),j)
enddo
return
end

subroutine evaluate2 (a,u,a_original,m,n)
integer m, n, irank
real a(m,n), u(m,m), a_original(m,n)
real t1(m,n)
cmf$ layout a(:news,:news), u(:news,:news)
cmf$ layout a_original(:news, :news)
cmf$ layout t1(:news,:news)
C evaluate USVt
C (SVt) is already in matrix a
t1 = matmul(transpose(u), a) - a_original
print*, 'error = max(abs( U S Vt - A )) is ', maxval(abs(t1))
return
end subroutine evaluate2

```

## Appendix C

### CM FORTRAN CODE FOR TEST DRIVER

```
program svdtest
C test program for singular value decomposition (svd) subroutine
integer, parameter:: mm=512, nn=512
integer m ,n ,irank, isweep, b(mm,mm), ans, i
real a(mm, nn), u(mm,nn), v(mm,nn), sv(mm)
real eps, error
cmf$ layout a(:news, :news), b(:news, :news), u(:news,:news)
cmf$ layout v(:news, :news), sv(:news)
interface
    subroutine svd (ab,ub,vb,sv,m,n,irank,isweep,eps)
        integer m,n,irank,isweep
        real ab(2*m,n), ub(2*m,n), vb(2*m,n), sv(2*m),eps
cmf$ layout ab(:news, :news), ub(:news, :news), vb(:news, :news)
cmf$ layout sv(:news)
    end interface

call CM_set_safety_mode(0)
print*, 'eps (default to 2.22e-16)'
read*,eps
if (eps.le.0) eps = 2.22e-16
print*,eps
a = 0.0
print*, 'm, n of matrix ?'
read*, m,n
print*,m,n
print*, 'max number of sweep ?'
read*, isweep
print*, isweep
print*, 'creating a random matrix'
call cmf_random (a(1:m,1:n))
print*, 'maxval matrix =', maxval(a(1:m,1:n))
print*, 'call svd routine'
call svd(a(1:m,1:n),u(1:m,1:n), v(1:m,1:n), sv(1:m),
        m/2, n, irank, isweep, eps)
```

```
print*, 'exit svd routine'  
stop
```

```
end
```