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

NHI-ANH CHU
Signal Processing Branch
Acoustics Division

April 11, 1991

Approved for public release; distribution unlimited.
An Implementation of the Singular Value Decomposition on the Connection Machine on CM-2

Chu, Nhi-Anh

Naval Research Laboratory
Washington, DC 20375-5000

NRL Report 9318

Approved for public release; distribution unlimited.

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.
## 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 A is a m x n matrix of rank r then there exist real orthogonal matrices $U = [u_1 u_2 ... u_m]$ and $V = [v_1 v_2 ... v_n]$ such that $A = U \Sigma V^t$ where

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

$r \leq \min(m, n)$ and $\sigma_i \geq \sigma_{i+1} > 0$ for $i = 1, ..., r - 1$. The $\sigma_i$ are the singular values of $A$ and the corresponding vectors $u_i$ and $v_i$ 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: $\text{rank}(A) = r$ where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > 0$

Dyadic decomposition: $A = \sum_{i=1}^{r} \sigma_i u_i v_i^t$

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 subspaces 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 \( \mathbf{A}^+ = \mathbf{V} \Sigma^+ \mathbf{U}^* \), with \( \Sigma^+ = \begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix} \).

The SVD and the Generalized SVD 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 \((1,2),(1,3),...,(1,m),(2,3),(2,4),...,(2,m),..., (m-1,m)\) 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.
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 \( A \) 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 \( A \) in \( \mathbb{R}^{m \times n} \) as \( A_{2 \times 2}^{m \times n} \) 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 \( \{ R_k \} \), \( R = R_{2 \times 2}^{m \times n} \). The rotated result is stored in a matrix \( H = H_{2 \times 2}^{m \times n} \).

The product of the rotation matrices is constructed by applying \( \{ R_k \} \) to an initial identity matrix \( I = I_{2 \times 2}^{m \times m} \) during the iterations. The result is kept in matrix \( U \).

\[
H_{2 \times 2}^{m \times n} = [U^t]_{2 \times 2}^{m \times m} A_{2 \times 2}^{m \times n}
\]

where

\[
[U^t]_{2 \times 2}^{m \times m} = \prod_k [R_k]_{2 \times 2}^{m \times m} I_{2 \times 2}^{m \times m}
\] (1)

Note that in Eq.(1), the rotation matrices \( R_k \) are replicated \( m \) times in each row to match the dimensions of \( I \).

After normalizing each row \( i \) of \( H \) by its norm \( \sigma_i \), \( H = [h_1, h_2, ..., h_i, ..., h_m]^t \) may be factored into as a product of a pseudo-diagonal matrix (a diagonal matrix concatenated with null rows) \( \Sigma \) and an orthonormal matrix \( V^t \).

\[
H = [h_1, h_2, ..., h_i, ..., h_m]^t = \Sigma V^t,
\] (2)

where

\[
|h_1| = \sigma_1 \geq |h_2| = \sigma_2 \geq \cdots \geq |h_r| = \sigma_r > 0;
\]

\[
V^t = [v_1, v_2, ..., v_i, ..., v_m]^t,
\]

where

\[
v_i = \begin{cases} h_i/\sigma_i, & i = 1, ..., r \\ 0, & i > r \end{cases}
\]
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:

$$\Sigma V^t = U^t A.$$ 

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

$$
\begin{pmatrix}
X \\
Y
\end{pmatrix} = R
\begin{pmatrix}
x \\
y
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}.
$$

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

$$X^tY = 0.$$ 

Defining

$$
\alpha = x^t y, \\
\beta = |x|^2 - |y|^2, \\
\gamma = (\alpha^2 + \beta^2)^{\frac{1}{2}},
$$

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

$$
\tan 2\theta = \frac{\alpha}{\beta}, \\
\cos 2\theta = \pm \frac{\alpha}{\gamma}, \\
\sin 2\theta = \pm \frac{\alpha}{\gamma} = 2 \sin \theta \cos \theta.
$$

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:

$$
\cos \theta = \pm \left( \frac{1 + \cos 2\theta}{2} \right)^{\frac{1}{2}} = \pm \left( \frac{\gamma + \beta}{2}\right)^{\frac{1}{2}}, \\
\sin \theta = \pm \left( \frac{1 - \cos 2\theta}{2} \right)^{\frac{1}{2}} = \pm \left( \frac{\gamma - \beta}{2}\right)^{\frac{1}{2}}.
$$

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 $x$:

$$
X^tX - x^tx = \frac{1}{2} \alpha \sin 2\theta - \beta \sin^2 \theta \\
= \pm \frac{\alpha^2}{2\gamma} - \beta \frac{\gamma - \beta}{2\gamma} = \pm \frac{\alpha^2 - \gamma \theta + \beta^2}{2\gamma}.
$$
If a + sign is chosen in place of the ± in the Eq. (9), the change in norm becomes

$$X'X - x'x = \frac{\gamma - \beta}{2},$$

in which case, $|x|$ 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 $y'y - Y'y$.

$$X'X - x'x = y'y - Y'y = \frac{\gamma - \beta}{2} \geq 0. \tag{11}$$

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 $R$ is:

1. If $\beta \geq 0$, calculate $\cos \theta = (\frac{\gamma + \beta}{2\gamma})^{\frac{1}{2}}$ then calculate $\sin \theta = \frac{\alpha}{2\gamma \cos \theta}$;
2. If $\beta < 0$, calculate $\sin \theta = \text{sign}(\alpha)(\frac{\gamma - \beta}{2\gamma})^{\frac{1}{2}}$ then calculate $\cos \theta = \frac{\alpha}{2\gamma \sin \theta}$.

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:

$$x'y \leq \delta \min(|x|, |y|) \tag{13}$$

where

$$\delta = \epsilon |A| = \epsilon \left( \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{\frac{1}{2}} \tag{14}$$

The single precision (32-bit) and double precision (64-bit) floating-point machine precisions are $1.17 \times 10^{-7}$ and $2.22 \times 10^{-16}$, 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 $(x_i, y_i), i = 1, ..., m/2$,

$$|x_i| \leq \delta \quad \text{or} \quad |y_i| \leq \delta \quad \text{or} \quad (x_i'y_i) \leq \delta \min(|x_i|, |y_i|). \tag{15}$$

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 $R$ and the current values of the square-norms. The loss of
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, ..., 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. \(|x_1| \geq |x_2| \geq ... \geq |x_m|\). The rows are then exchanged pairwise: (1, 2), (3, 4), (5, 6), ..., (m - 3, m - 2), (m - 1, m) to become (2, 1), (4, 3), (6, 5), ..., (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), ..., (m - 3, m). The iteration repeats even-odd for a total of m cycles (a sweep) after which all pairing of the rows (1, ..., m) have been visited and the norm ordering is reversed, i.e. m, m - 1, ..., 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 $R$ (Eq. (4)) also converges into a bubble-sort transformation because of the ordered-norm conditions described in Eq. (11). For each row-pair $(x, y)$, as $|x|$ keeps increasing and $|y|$ keeps decreasing, eventually $|x| \geq |y|$. 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 $R$. 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 $R$ 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 $A$ is $m \times n$, then $U$ and $V$ 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 $A$ 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 $U$. In this case, $U^T$ may be internally processed sequentially as $n \times n$ 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 $m \approx n$. 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 1K 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 $10^{-16}$.

<table>
<thead>
<tr>
<th>Size</th>
<th>Machine</th>
<th># processors</th>
<th>exec time</th>
<th># sweeps</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>CM-2</td>
<td>16K</td>
<td>280 sec</td>
<td>12</td>
</tr>
<tr>
<td>256</td>
<td>CM-2</td>
<td>16K</td>
<td>39 sec</td>
<td>12</td>
</tr>
<tr>
<td>512</td>
<td>Convex</td>
<td>1</td>
<td>141 sec</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>Convex</td>
<td>1</td>
<td>21 sec</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>Sun-4</td>
<td>1</td>
<td>305 sec</td>
<td></td>
</tr>
</tbody>
</table>

In Table 1, the execution times for $m = n$ are compared against the execution times of the LINPACK $dsyevd$ codes on one processor of a Convex C210 and a Sun/4-280. Both the Convex and Sun/4-280 codes were compiled with optimization; loads were minimal. The codes for the general cases where $m > n$ 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

$$\text{Error} = \max \left| A - USV^T \right|.$$  \hspace{1cm} (16)

Errors in the CM-2 runs were on the order of $10^{-14}$ for double precision and $10^{-5}$ 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 $svdcore$ 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 was included 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.)

$$Q_{FLOP} = 100\left(\frac{m}{2}n\right)I_s,$$  \hspace{1cm} (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).

<table>
<thead>
<tr>
<th>Double Precision</th>
<th>Fieldwise Slicewise, unrolled Fieldwise, unrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Fieldwise Slicewise, unrolled Fieldwise, unrolled</td>
</tr>
<tr>
<td>512</td>
<td>35 23 30</td>
</tr>
<tr>
<td>256</td>
<td>7.5 5</td>
</tr>
</tbody>
</table>

Table 2 shows the execution time per sweep in units of seconds for the cases $m = n = 256 \times 2, \times 512$ 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


Appendix A

CM FORTRAN CODE FOR SUBROUTINE SVD OPTIMIZED FOR $m \approx n$

_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.

```fortran
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 V^t_)
C ab is returned as (_U^t A_) where _U_ is product of Jacobi rotation
C matrices on (_A^t A_)
C ub -- 2m x n matrix returned with _U^t_
C vb -- 2m x n matrix returned with _V^t_
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"
C
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)
```

13
interface
    subroutine one2two(a, ap, b, m, n)
    integer m, n
    real a(m, n), ap(m, n), b(2*m, n)
end interface

interface
    subroutine svdcore(a, ap, u, up, m, n, irank, isweep, deltas, eps)
    integer irank, isweep, m, n
    real a(m, n), ap(m, n), u(m, n), up(m, n), eps, deltas
end interface

interface
    subroutine two2one(a, ap, b, m, n)
    integer m, n
    real a(m, n), ap(m, n), b(2*m, n)
end interface

interface
    subroutine evaluatel(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
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)
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)
call CM_timer_start(1)
print*, 'call evaluate'
call CM_timer_stop(1)
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-------------------------------------------------------------
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
forall (i=1:m, j=1:n) row(i,j) = i
C---------------------------------------------------------------
isweep = 0
C
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
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
C---------------------------------------------------------------
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

17
up = c*u + s*up
normsp = c*c*norms + s*s*normsp + alpha*c*s
a = atemp
u = utemp
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
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).and.(row.ne.m))
atemp = ap
utemp = u
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 = 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
\[ \alpha = 2 \text{spread}(\sum(a \cdot a p, 2), 2, n) \]
\[ \beta = \text{normsp} - \text{norms} \]
\[ \gamma = \sqrt{(\alpha \cdot \alpha) + (\beta \cdot \beta)} \]
\[ \text{ortho} = 0.25 \alpha \cdot \alpha - \delta_{\text{n}} \cdot \min(\text{norms}, \text{normsp}) \]
\[ \text{rotate} = (\text{norms} \geq \delta_{\text{t}}) \cdot (\text{norms} \geq \delta_{\text{t}}) \cdot (\text{ortho} \geq 0) \]
\[ \text{where} ((\beta \geq 0) \cdot \text{rotate}) \]
\[ c = \sqrt{\left(\frac{\gamma + \beta}{2 \cdot \gamma}\right)} \quad \text{!cosine term} \]
\[ s = \alpha / (2 \cdot \gamma \cdot c) \quad \text{!sine term} \]
\[ \text{atemp} = s \cdot a + c \cdot a p \]
\[ \text{utemp} = s \cdot u + c \cdot u p \]
\[ \text{ntemp} = s \cdot s \cdot \text{norms} + c \cdot c \cdot \text{normsp} + \alpha \cdot c \cdot s \]
\[ a p = c \cdot a - s \cdot a p \]
\[ u p = c \cdot u - s \cdot u p \]
\[ \text{normsp} = c \cdot c \cdot \text{norms} + s \cdot s \cdot \text{normsp} - \alpha \cdot c \cdot s \]
\[ a = \text{atemp} \]
\[ u = \text{utemp} \]
\[ \text{norms} = \text{ntemp} \]
\[ \text{irot1} = \text{irot1} + 1 \]
\[ \text{endwhere} \]
\[ \text{where} ((\beta < 0) \cdot \text{rotate}) \]
\[ s = \text{sign}\left(\sqrt{\left(\frac{\gamma - \beta}{2 \cdot \gamma}\right)}\right), \alpha \]
\[ c = \alpha / (2 \cdot \gamma \cdot s) \]
\[ \text{atemp} = s \cdot a + c \cdot a p \]
\[ \text{utemp} = s \cdot u + c \cdot u p \]
\[ \text{ntemp} = s \cdot s \cdot \text{norms} + c \cdot c \cdot \text{normsp} + \alpha \cdot c \cdot s \]
\[ a p = c \cdot a - s \cdot a p \]
\[ u p = c \cdot u - s \cdot u p \]
\[ \text{normsp} = c \cdot c \cdot \text{norms} + s \cdot s \cdot \text{normsp} - \alpha \cdot c \cdot s \]
\[ a = \text{atemp} \]
\[ u = \text{utemp} \]
\[ \text{norms} = \text{ntemp} \]
\[ \text{irot1} = \text{irot1} + 1 \]
\[ \text{endwhere} \]
\[ \text{where} ((\beta > 0) \cdot (\text{not rotate})) \]
\[ \text{atemp} = a p \]
\[ \text{utemp} = u p \]
\[ \text{ntemp} = \text{normsp} \]
\[ a p = a \]
\[ u p = u \]
\[ \text{normsp} = \text{norms} \]
\[ a = \text{atemp} \]
\[ u = \text{utemp} \]
\[ \text{norms} = \text{ntemp} \]
\[ \text{endwhere} \]

C communicate (a, ap) to/from processors aligned with odd rows
\[ \text{atemp} = a p \]
\[ \text{utemp} = u p \]
\[ \text{ntemp} = \text{normsp} \]
ap = cshift(a, 1, sdown)
up = cshift(u, 1, sdown)
normsp = cshift(norms, 1, sdown)
a = atemp
u = utemp
norms = ntemp

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

endwhere

communicate (a, ap) to/from processors aligned with odd rows
atemp = a
utemp = u
ntemp = norms
a = cshift(ap, 1, sup)
= cshift(up, 1, sup)
norms = cshift(normsp, 1, sup)
ap = atemp
up = utemp
normsp = ntemp
endo ! 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)

oned = sum(a*a,2)
sv = sqrt(oned)
where (oned.gt.deltas)
one = 1
elsewhere
one = 0
sv = 0.0
endwhere
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 VtV off diagonal ',
   maxval(abs(t3(1:n,1:n)), mask=(row.ne.col))
C evaluate UUt
   t1 = 0.0
   t2 = 0.0
   t2 = transpose(tl)
   t3 = 0.0
   t3 = matmul (t1, t2)
   print*, 'maxval UUt of f 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( USVt - 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 = (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)

25
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(pm,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
  end interface

26
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

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

28
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
enddo
where (rotate)
 utemp(k,:,:) = -s*u(k,:,:) + c*up(k,:,:)
up(k,:,:) = c*u(k,:,:) + s*up(k,:,:)
u(k,:,:) = utemp(k,:,:)
endwhere
endo
where ((beta.gt.0).and.(.not.rotate))
 atemp = ap
ntemp = normsp
ap = a
normsp = norms
a = atemp
norms = ntemp
endwhere
doi = 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
doi = 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
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

30
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).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,:,:)
endwhere

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,:,:)
up(k,:,:) = cshift(up(k,:,:), 1, sup)
up(k,:,:) = utemp(k,:,:)
endo
endo ! 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,:,:)
endo
!
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
endo
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,:,:)
endwhere
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
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
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)
calculate singular values and rank
\[
t_3 = a(1:n, 1:n) \\
sv = \sqrt{\text{sum}(t_3*t_3, 2)} \\
\text{irank} = \text{count}(sv gt \sqrt{\text{deltas}})
\]
calculate v
\[
t_2 = \text{spread}(sv, 2, n) \\
v = 0.0
\]
where \((t_2.gt.0) v = t_3/t_2\)
debug codes beyond the next statement
return
detailed check
print*, 'max min of v', maxval(v), minval(v)
evaluate VtV
\[
t_2 = \text{matmul}(\text{transpose}(v), v)
\]
forall \((i=1:m, j=1:m) \text{row}(i, j) = i\)
forall \((i=1:m, j=1:m) \text{col}(i, j) = j\)
print*, 'maxval VVt off diagonal', maxval(abs(t2)),
\text{mask}=(\text{row}(1:n, 1:n).ne.\text{col}(1:n, 1:n))
evaluate UUt
\[
t_1 = \text{matmul}(\text{transpose}(u), u)
\]
print*, 'maxval UUt off diagonal',
\text{maxval}(abs(t1)), \text{mask}=(\text{row}.ne.\text{col})
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 \((i=2:2*m:2, j=1:n) b(i,j) = a(1+((i-1)/2), j)\)
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)
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)
  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 evaluate2 (a, u, a_original, m, n)
  integer m, n, irank
  real a(m,n), u(m,m), a_original(m,n)
  real ti(m,n)
  C evaluate USVt
  C (SVt) is already in matrix a
  ti = matmul(transpose(u), a) - a_original
  print*,’error = max(abs( U S Vt - A )) is ’, maxval(abs(ti))
  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)
N. A. CHU

print*, 'exit svd routine'
stop
end