COMPUTATION OF THE SINGULAR VALUE DECOMPOSITION USING MESH-CONNECTED PROCESSORS

Richard P. Brent†
Franklin T. Luk*
Charles Van Loan*

TR 82-528
March 1983

†Department of Computer Science
Australian National University
Canberra, A.C.T. 2600 Australia

*Department of Computer Science
Cornell University
Ithaca, New York 14853

Computation of the Singular Value Decomposition Using Mesh-Connected Processors

Richard P. Brent

Department of Computer Science
Australian National University
Canberra, A.C.T. 2600
Australia

Franklin T. Luk

Department of Computer Science
Cornell University
Ithaca, New York 14853
U.S.A.

Charles Van Loan

Department of Computer Science
Cornell University
Ithaca, New York 14853
U.S.A.

ABSTRACT

A cyclic Jacobi method for computing the singular value decomposition of an $m \times n$ matrix $(m \geq n)$ using systolic arrays is proposed. The algorithm requires $O(n^2)$ processors and $O(m + n\log n)$ units of time.

Keywords and Phrases: Systolic arrays, singular value decomposition, cyclic Jacobi method, real-time computation, VLSI.

1. Introduction

The singular value decomposition (SVD) of a matrix $A \in \mathbf{R}^{m \times n}$,where $m \geq n$, is given by

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

where $U \in \mathbf{R}^{m \times m}$ and $V \in \mathbf{R}^{n \times n}$ are each orthogonal and

$$\Sigma = diag(\sigma_1, \cdots, \sigma_n) \in \mathbf{R}^{m \times n},$$

with $\sigma_1 \geq \cdots \geq \sigma_n \geq 0$. The σ_i are referred to as the singular values of A, and the columns of U and V are respectively known as the left and right singular vectors. The decomposition has many important scientific and engineering applications (cf. [10]).

The standard method for computing (1.1) is the Golub-Kahan-Reinsch SVD algorithm ([9] and [11]) that is implemented in both EISPACK [7] and LINPACK [4]. However, the advent of massively parallel computer architectures has aroused much interest in parallel SVD procedures, e.g., [1], [5], [13], [16], [17] and [20]. Such architectures may turn out to be indispensable in settings where real-time computation of the SVD is required (cf. [21] and [22]). Speiser and Whitehouse [21] survey parallel processing architectures and conclude that systolic architectures offer the best combination of characteristics for utilizing VLSI/VHSIC technology to do real-time signal processing. (See also [15] and [22].) The most effective SVD algorithm that has been proposed for systolic array implementation is apparently the Jacobi-like algorithm given by Brent and Luk [1]. This is not surprising because Jacobi-type procedures are very amenable to parallel computations (cf. [2], [17] and [19]).

In this report we present a modification of the two-sided Jacobi SVD method for square matrices detailed in Forsythe and Henrici [6] and show how it can be implemented on a systolic array. The array is very similar to the one proposed in Brent and Luk [2] that uses $O(n^2)$ processors to solve n-by-n symmetric eigenproblems in $O(n \log n)$ time. However, to handle m-by-n SVD problems, our algorithm requires a "pre-processing" step in which the QR-factorization $A = QR$ is computed. This can be done in $O(m)$ time using the systolic array described in [8]. Our modified Brent-Luk array then computes the SVD of R in $O(n \log n)$ time. $O(n^2)$ processors are required for the overall algorithm.

The paper is organized as follows. In Section 2 basic concepts and calculations are discussed by reviewing the Jacobi algorithm for the symmetric eigenproblem. Our modified Forsythe-Henrici scheme is then presented in Sections 3 and 4 where we discuss two-by-two SVD problems and "parallel orderings" respectively. In Section 5 we describe a systolic array for implementing the scheme. How this array can be used to solve rectangular SVD problems is discussed in Section 6. In that final section we also describe a block Jacobi method that might be of interest when our array is too small to handle a given SVD problem.

2. Background

The classical method of Jacobi uses a sequence of plane rotations to diagonalize a symmetric matrix $A \in \mathbf{R}^{n\times n}$. We denote a Jacobi rotation of an angle θ in the (p,q) plane by $J(p,q,\theta) \equiv J$, where $p<q$. The matrix J is same as the identity matrix except for four strategic elements:

$$J_{pp} = c \ , \ J_{pq} = s \ ,$$
$$J_{qp} = -s \ , \ J_{qq} = c \ ,$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. Letting $B = J^T AJ$, we get

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}.$$

If we choose the cosine-sine pair (c,s) such that

$$b_{pq} = b_{qp} = a_{pq}(c^2 - s^2) + (a_{pp} - a_{qq})cs = 0, \tag{2.1}$$

then B becomes "more diagonal" than A in the sense that

$$off(B) = off(A) - 2a_{pq}^2,$$

where

$$off(C) \equiv \sum_{i\neq j} c_{ij}^2 \quad \text{for } C = (c_{ij}) \in \mathbf{R}^{m\times n}.$$

Assuming that $a_{pq} \neq 0$, we have from equation (2.1) that

$$\rho \equiv \frac{a_{qq}-a_{pp}}{2a_{pq}} = \frac{c^2-s^2}{2cs} = \mathrm{ctn}(2\theta), \tag{2.2}$$

and that $t \equiv \tan(\theta)$ satisfies

$$t^2 + 2\rho t - 1 = 0.$$

By solving this quadratic equation and using a little trigonometry we find two possible solutions to (2.1):

$$t = -\mathrm{sign}(\rho)\left[\ |\rho| + \sqrt{1+\rho^2}\ \right] \ , \ c = \frac{1}{\sqrt{1+t^2}} \ , \ s = ct, \tag{2.3}$$

and

$$t = \frac{\mathrm{sign}(\rho)}{|\rho| + \sqrt{1+\rho^2}} \ , \ c = \frac{1}{\sqrt{1+t^2}} \ , \ s = ct. \tag{2.4}$$

The angle θ associated with (2.4) is the smaller of the two possible rotation angles and satisfies $0 \leq |\theta| \leq \pi/4$.

By systematically zeroing off-diagonal entries, A can be effectively diagonalized. To be specific, consider the iteration:

Algorithm Jacobi.

```
do until off(A)<ς
   for p = 1, · · · ,n-1
      for q = p+1, · · · ,n
         begin
         Determine c and s via (2.2) and (2.4);
         A := J(p,q,θ)^T A J(p,q,θ)
         end.
```

The parameter ς is some small machine-dependent number. Each pass through the "until" loop is called a "sweep". In this scheme a sweep consists of zeroing the off-diagonal elements according to the "row ordering" [6]. This ordering is amply illustrated by the $n = 4$ case:

$$(p,q) = (1,2)\,,\ (1,3)\,,\ (1,4)\,,\ (2,3)\,,\ (2,4)\,,\ (3,4)\ .$$

It is well-known that Algorithm Jacobi always converges [6] and that the asymptotic convergence rate is quadratic [23]. The rigorous proof of these results requires that (2.4) rather than (2.3) be used. Brent and Luk [2] conjecture that $O(\log n)$ sweeps are required by Algorithm Jacobi for typical values of ς, e.g., $\varsigma = 10^{-12} off(A_0)$, where A_0 denotes the original matrix. For most problems this usually means about six to ten sweeps.

The practical computation of c and s requires that we guard against overflow. If ϵ denotes the machine precision, then this can usually be accomplished as follows:

Algorithm CS.

$$
\begin{aligned}
&\mu_1 := a_{qq} - a_{pp};\\
&\mu_2 := 2a_{pq};\\
&\textbf{if } |\mu_2| \leq \epsilon\ |\mu_1| \textbf{ then}\\
&\quad \textbf{begin}\\
&\quad c := 1;\\
&\quad s := 0\\
&\quad \textbf{end}\\
&\textbf{else}\\
&\quad \textbf{begin}\\
&\quad \rho := \frac{\mu_1}{\mu_2};\\
&\quad t := \frac{\mathrm{sign}(\rho)}{|\rho| + \sqrt{1+\rho^2}};\\
&\quad c := \frac{1}{\sqrt{1+t^2}};\\
&\quad s := ct\\
&\quad \textbf{end.}
\end{aligned}
$$

Jacobi methods for the symmetric eigenproblem are of interest because they lend themselves to parallel computation [2, 19]. Of particular interest to us is the "parallel ordering" described in [1] and illustrated in the $n = 8$ case by

$$
\begin{aligned}
(p,q) \;=\; &(1,2)\,,\,(3,4)\,,\,(5,6)\,,\,(7,8)\,,\\
&(1,4)\,,\,(2,6)\,,\,(3,8)\,,\,(5,7)\,,\\
&(1,6)\,,\,(4,8)\,,\,(2,7)\,,\,(3,5)\,,\\
&(1,8)\,,\,(6,7)\,,\,(4,5)\,,\,(2,3)\,,\\
&(1,7)\,,\,(8,5)\,,\,(6,3)\,,\,(4,2)\,,\\
&(1,5)\,,\,(7,3)\,,\,(8,2)\,,\,(6,4)\,,\\
&(1,3)\,,\,(5,2)\,,\,(7,4)\,,\,(8,6)\,.
\end{aligned}
$$

Note that the rotation pairs associated with each "row" of the above can be calculated concurrently. Brent and Luk [2] have developed a systolic array which exploits this concurrency to such an extent that a sweep with the parallel ordering can be executed in $O(n)$ time. Our aim is to extend their work to the SVD problem.

At first glance this seems unnecessary since software (or hardware) for the symmetric eigenvalue problem can in principle be used to solve the SVD problem. For example, we may compute the eigenvalue decomposition

$$V^T(A^TA)V = diag(\sigma_1^2, \cdots, \sigma_n^2),$$

where $V = [v_1, \cdots, v_n]$ is orthogonal and the σ_i satisfy

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

with $r = \text{rank}(A)$. We next calculate the vectors

$$u_i = \frac{Av_i}{\sigma_i} \qquad (i=1,...,r), \tag{2.5}$$

and then determine the vectors $u_{r+1}, \cdots, u_m$ so that the matrix $U = [u_1, \cdots, u_m]$ is orthogonal. The factorization $U^TAV = \text{diag}(\sigma_1, \cdots, \sigma_n)$ gives the SVD of A. Thus, one can theoretically compute the SVD of A via the eigenvalue decomposition of A^TA. Unfortunately, well-known numerical difficulties are associated with the explicit formation of A^TA.

A way round this difficulty is to apply the Jacobi method implicitly. This is the gist of the "one-sided" Hestenes approach in which the matrix V is determined so that the columns of AV are mutually orthogonal [14]. Implementations are discussed in Luk [17] and in Brent and Luk [1]. In the latter reference a systolic array is developed that is tailored to the method. However, inner products of m-vectors are required for each (c,s) computation. Because of this, the speed of their parallel algorithm is $O(mn\log n)$ for a linear array of processors, and $O(n\log m\log n)$ for a two-dimensional array of $O(mn)$ processors with some special interconnection patterns for inner-product computations. Another drawback of the one-sided Jacobi method is that it also does not directly generate the vectors $u_{r+1}, \cdots, u_m$. This is an inconvenience in the systolic array setting since one would need a special architecture to carry out the matrix-vector multiplications in (2.5). It should be pointed out, however, that these vectors are not necessary for solving the least squares problem $min \| Ax - b \|_2$.

Yet another approach to the SVD problem is to compute the eigenvalue decomposition of the $(m+n)\times(m+n)$ symmetric matrix

$$C = \begin{bmatrix} O & A \\ A^T & 0 \end{bmatrix}.$$

Note that if

$$\begin{bmatrix} O & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \sigma \begin{bmatrix} u \\ v \end{bmatrix},$$

then $A^T A v = \sigma^2 v$ and $A A^T u = \sigma^2 u$. Thus, the eigenvectors of C are "made up" of the singular vectors of A . It can also be shown that the spectrum of C is given by

$$\lambda(C) = \{ \pm \sigma_1 , \cdots , \pm \sigma_n , 0 , \cdots , 0 \}.$$

The disadvantages of this approach to the SVD are that C has expanded dimension and that recovering the singular vectors may be a difficult numerical task (cf. [9]). In addition, the case $\mathrm{rank}(A) < n$ requires extra work to generate $v_{r+1} ,\dots, v_n$.

To summarize, it is preferable from several different points of view *not* to approach the SVD problem as a symmetric eigenvalue problem.

3. The Two-by-Two SVD Problem

In this section we discuss how to compute cosine-sine pairs (c_1,s_1) and (c_2,s_2) such that

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} w & x \\ y & z \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}, \tag{3.1}$$

where

$$A = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

is given. It is always possible to choose the above rotations such that $|d_1| \geq |d_2|$. However, since $\det(A) = d_1 d_2$, it may not be possible to achieve $d_1 \geq d_2 \geq 0$. Consequently, we refer to (3.1) as an "unnormalized" SVD. To generate a genuine, "normalized" SVD, it may be necessary to use reflections as well as rotations. We will return to this detail at the end of this section.

Forsythe and Henrici [6, p.21] suggest the following approach for computing (3.1). Let θ_1 and θ_2 be the angles generating the cosine-sine pairs (c_1,s_1) and (c_2,s_2), respectively. Then θ_1 and θ_2 are solutions to the equations:

$$\tan(\theta_1 + \theta_2) = \frac{y+x}{z-w} \quad , \quad \tan(-\theta_1 + \theta_2) = \frac{y-x}{z+w} \ .$$

We may use the following procedure to determine the cosine-sine pairs.

Algorithm FHSVD.

$\mu_1 := z - w;$
$\mu_2 := y + x;$

{ Find $\chi_1 = \cos(\frac{\theta_1 + \theta_2}{2})$ and $\sigma_1 = \sin(\frac{\theta_1 + \theta_2}{2})$ }

if $|\mu_2| \le \epsilon\ |\mu_1|$ **then**
  begin
  $\chi_1 := 1;$
  $\sigma_1 := 0$
  end
else
  begin
  $\rho_1 := \frac{\mu_1}{\mu_2};$
  $\tau_1 := \frac{\mathrm{sign}(\rho_1)}{|\rho_1| + \sqrt{1+\rho_1^2}}\ ;$
  $\chi_1 := \frac{1}{\sqrt{1+\tau_1^2}};$
  $\sigma_1 := \chi_1\tau_1$
  end;

$\mu_1 := z + w;$
$\mu_2 := y - x;$

{ Find $\chi_2 = \cos(\frac{-\theta_1 + \theta_2}{2})$ and $\sigma_2 = \sin(\frac{-\theta_1 + \theta_2}{2})$ }

if $|\mu_2| \le \epsilon\ |\mu_1|$ **then**
  begin
  $\chi_2 := 1;$
  $\sigma_2 := 0$
  end
else
  begin
  $\rho_2 := \frac{\mu_1}{\mu_2};$
  $\tau_2 := \frac{\mathrm{sign}(\rho_2)}{|\rho_2| + \sqrt{1+\rho_2^2}}\ ;$
  $\chi_2 := \frac{1}{\sqrt{1+\tau_2^2}};$
  $\sigma_2 := \chi_2\tau_2$
  end;

{ Find (c_1,s_1) and (c_2,s_2) }

$c_1 := \chi_1\chi_2 + \sigma_1\sigma_2;$
$s_1 := \sigma_1\chi_2 - \chi_1\sigma_2;$
$c_2 := \chi_1\chi_2 - \sigma_1\sigma_2;$
$s_2 := \sigma_1\chi_2 + \chi_1\sigma_2.$

An alternative method for computing (3.1) is to first symmetrize A:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} p & q \\ q & r \end{bmatrix}, \tag{3.2}$$

and then diagonalize the result:

$$\begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}^T \begin{bmatrix} p & q \\ q & r \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}.$$

Equation (3.1) holds by setting

$$\begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix}^T \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T = \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T, \tag{3.3}$$

that is,

$$c_1 = c_2 c - s_2 s,$$
$$s_1 = s_2 c + c_2 s.$$

The formulae for c and s are easily derived. From (3.2) we have

$$cx - sz = sw + cy .$$

Thus if $x \neq y$, then we have $\rho \equiv \mathrm{ctn}(\theta) = \dfrac{w+z}{x-y}$, $s = \dfrac{\mathrm{sign}(\rho)}{\sqrt{1+\rho^2}}$, $c = s\rho$.

For reasons that will be explained in Section 6, it is necessary that our two-by-two SVD algorithm handle the special cases $x=z=0$ and $y=z=0$ as follows:

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} w & 0 \\ y & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & 0 \end{bmatrix}, \tag{3.4}$$

and

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T \begin{bmatrix} w & x \\ 0 & 0 \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & 0 \end{bmatrix}. \tag{3.5}$$

In other words, θ_2 should be zero if $x=z=0$, and θ_1 should be zero if $y=z=0$. Our scheme as described already produces (3.4). However, (3.5) is only achieved if $|w|>|x|$. To rectify this we apply our algorithm to the transpose problem and obtain

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} w & 0 \\ x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & 0 \end{bmatrix}.$$

We then set $c_2=c_1$, $s_2=s_1$, $c_1=1$, and $s_1=0$. Let us present the overall scheme:

Algorithm USVD.

flag := 0;
if $y = 0$ and $z = 0$ then
  begin
  $y := x$; $x := 0$; flag := 1
  end;

$\mu_1 := w + z$;
$\mu_2 := x - y$;

if $|\mu_2| \le \epsilon\ |\mu_1|$ then
  begin
  $c := 1$; $s := 0$
  end
else
  begin
  $\rho := \dfrac{\mu_1}{\mu_2}$;

  $s := \dfrac{\mathrm{sign}(\rho)}{\sqrt{1+\rho^2}}$;
  $c := s\rho$
  end;

$\mu_1 := s(x+y) + c(z-w)$; $\{= r - p\}$
$\mu_2 := 2(cx - sz)$; $\{= 2q\}$

if $|\mu_2| \le \epsilon\ |\mu_1|$ then
  begin
  $c_2 := 1$; $s_2 := 0$
  end
else
  begin
  $\rho_2 := \dfrac{\mu_1}{\mu_2}$;
  $t_2 := \dfrac{\mathrm{sign}(\rho_2)}{|\rho_2| + \sqrt{1+\rho_2^2}}$;

  $c_2 := \dfrac{1}{\sqrt{1+t_2^2}}$;
  $s_2 := c_2 t_2$
  end;

$c_1 := c_2 c - s_2 s$;
$s_1 := s_2 c + c_2 s$;
$d_1 := c_1(wc_2 - xs_2) - s_1(yc_2 - zs_2)$;
$d_2 := s_1(ws_2 + xc_2) + c_1(ys_2 + zc_2)$;

if flag = 1 then
  begin
  $c_2 := c_1$; $s_2 := s_1$;
  $c_1 := 1$; $s_1 := 0$
  end.

If a "normalized" SVD is required then further computations must be performed. The diagonal elements must be sorted on modulus and then (if necessary) premultiplied by -1, as shown in the following algorithm.

Algorithm NSVD.

```
Apply Algorithm USVD;

if |d2| > |d1| then
   begin
   τ := c1 ; c1 := -s1 ; s1 := τ ;
   τ := c2 ; c2 := -s2 ; s2 := τ ;
   τ := d1 ; d1 := d2 ; d2 := τ
   end;
κ := 1;
if d1 < 0 then
   begin
   d1 := -d1;
   c1 := -c1;
   s1 := -s1;
   κ := -κ
   end;
if d2 < 0 then
   begin
   d2 := -d2;
   κ := -κ
   end.
```

The "normalized" SVD is given by

$$\begin{bmatrix} c_1 & s_1\kappa \\ -s_1 & c_1\kappa \end{bmatrix}^T \begin{bmatrix} w & x \\ y & z \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}. \tag{3.6}$$

Note that if κ=−1 then the left transformation is a reflection, i.e., a two-by-two orthogonal matrix of the form $\begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) \\ -\sin(\theta_1) & -\cos(\theta_1) \end{bmatrix}$. We denote the corresponding n-by-n orthogonal transformation by $\hat{J}(p,q,\theta_1,\kappa) \equiv \hat{J}$, where $p<q$. The matrix $\hat{J}$ is same as the identity except for the four elements:

$$\hat{J}_{pp} = \cos(\theta_1) , \hat{J}_{pq} = \kappa\sin(\theta_1) ,$$
$$\hat{J}_{qp} = -\sin(\theta_1) , \hat{J}_{qq} = \kappa\cos(\theta_1) ,$$

and it is a reflection whenever $\kappa=-1$.

4. Some Two-Sided Jacobi SVD Procedures

By solving an appropriate sequence of two-by-two SVD problems, we may compute the SVD of a general $n \times n$ matrix A. Analogous to Algorithm Jacobi for the symmetric eigenvalue problem, we propose the following routine:

Algorithm JSVD.

do until $off(A) < \varsigma$
  for each (p,q) in accordance with the chosen ordering
    begin
    Set $w = a_{pp}$, $x = a_{pq}$,
      $y = a_{qp}$, $z = a_{qq}$;

    Compute (c_1,s_1), (c_2,s_2) and $\kappa \epsilon \{-1,1\}$ so that
      the (p,q) and (q,p) entries
      of $\hat{J}(p,q,\theta_1,\kappa)^T A \hat{J}(p,q,\theta_2,\kappa)$ are zero;

    $A := \hat{J}(p,q,\theta_1,\kappa)^T A \hat{J}(p,q,\theta_2,\kappa)$
    end.

We have experimentally compared the three angle formulas FHSVD, USVD and NSVD. The convergence of Algorithm JSVD in conjunction with any one of these formulas and with any ordering has not been rigorously proved. However, there are some important theoretical observations that can be made.

Let θ_1 and θ_2 be the angles generating the cosine-sine pairs (c_1,s_1) and (c_2,s_2), respectively, and suppose that $-b \leq \theta_1$, $\theta_2 \leq b$. Although Forsythe and Henrici [6] prove that JSVD with the "row ordering" always converges if $b < \frac{\pi}{2}$, they also demonstrate when this condition may fail to hold. As a remedy, they suggest an under- or over-rotation variant of FHSVD and prove its convergence. However, we do not favor this variant, believing that it will eliminate the empirically observed quadratic convergence of Algorithm JSVD. One can show that the bound b equals (i) $\frac{\pi}{2}$ for FHSVD, (ii) $\frac{3\pi}{4}$ for USVD, and (iii) $\frac{5\pi}{4}$ for NSVD even if no reflection is involved. We conjecture that the smaller the rotation angles are the faster the procedure JSVD will converge. Our experimental results agree with the conjecture.

We also point out that the computed diagonal matrix in JSVD may not have sorted diagonal entries even if the "normalized" SVD of each two-by-two submatrix is calculated. To obtain the "normalized" SVD of A we must sort the diagonal entries and permute the columns of U and V accordingly.

It is quite obvious that the "row ordering" is not amenable to parallel processing. A new "parallel ordering" has been introduced in [1] and [2] for doing $\left\lfloor \frac{n}{2} \right\rfloor$ rotations simultaneously. For the symmetric eigenvalue problem, the latter ordering has been found to be superior to the former both empirically [2] and theoretically (see [2] and [12]) even on a single-processor machine. Our consideration of systolic array implementation necessitates the use of the "parallel ordering" in this paper.

We have applied each of the three angle formulas with the "parallel ordering" to random $n \times n$ matrices A, whose elements were uniformly and independently distributed in [-1,1]. The stopping criterion was that $off(A)$ was reduced to 10^{-12} times its original value. Table 1 gives the average and, in parentheses, the maximum number of sweeps required for convergence.

n	trials	FHSVD	USVD	NSVD
4	1000	2.97(3.67)	2.97(4.00)	2.97(4.33)
6	1000	3.73(4.80)	3.76(4.87)	4.17(5.40)
8	1000	4.19(5.07)	4.21(5.14)	4.76(5.96)
10	1000	4.51(5.38)	4.55(5.44)	5.18(6.40)
20	100	5.50(6.13)	5.54(6.01)	6.44(7.44)
30	100	6.03(6.52)	6.09(6.80)	7.30(7.98)
40	100	6.36(6.98)	6.40(6.98)	7.94(8.50)
50	100	6.66(7.11)	6.72(7.34)	8.51(9.00)

Table 1 : Simulation results for the "parallel ordering"

We conclude that our "unnormalized" SVD method USVD performs just as well as the more complicated Forsythe-Henrici scheme FHSVD and marginally faster than the "normalized" SVD method NSVD.

We have also compared the "row" and "parallel" orderings for the method USVD. The test data and convergence criterion are as described in the last paragraph. The average and the maximum number of sweeps are presented in Table 2. The convergence rates for the two orderings appear to be about the same.

n	trials	row ordering	parallel ordering
4	1000	3.22(4.17)	2.97(4.00)
6	1000	4.00(5.13)	3.76(4.87)
8	1000	4.52(5.50)	4.21(5.14)
10	1000	4.83(6.02)	4.55(5.44)
20	100	5.81(6.59)	5.54(6.01)
30	100	6.26(6.79)	6.09(6.80)
40	100	6.60(7.00)	6.40(6.98)
50	100	6.78(7.62)	6.72(7.34)
80	30	7.31(7.81)	7.30(7.79)
100	10	7.46(7.79)	7.56(8.00)
120	5	7.69(7.90)	7.73(7.98)
150	3	7.76(7.80)	7.73(8.03)
170	2	7.86(7.92)	8.02(8.02)
200	1	7.90(7.90)	8.10(8.10)
230	1	8.35(8.35)	8.43(8.43)

Table 2 : Simulation results for the USVD method

In the remainder of the paper we will focus our attention exclusively on the USVD method for solving two-by-two SVD problems.

5. A Systolic Array

We now describe a systolic array for implementing the Jacobi SVD method JSVD for an $n \times n$ real matrix A. Our array is very similar to the one detailed in [2]. For pedagogic purpose, we first idealize the array so that it has the ability to broadcast the rotation parameters in constant time.

Assume that n is even and that we have a square array of $\frac{n}{2}$ by $\frac{n}{2}$ processors, each containing a 2×2 submatrix of A. Initially processor P_{ij} contains $\begin{bmatrix} a_{2i-1,2j-1} & a_{2i-1,2j} \\ a_{2i,2j-1} & a_{2i,2j} \end{bmatrix}$ (i, j = 1, $\cdots$, $\frac{n}{2}$). Processor P_{ij} is connected to its "diagonally" nearest neighbors $P_{i\pm 1, j\pm 1}$ (1 < i,j < $\frac{n}{2}$). For the boundary processors and for a complete picture, see Figures 1 and 2. In general P_{ij} contains four real numbers $\begin{bmatrix} \alpha_{ij} & \beta_{ij} \\ \gamma_{ij} & \delta_{ij} \end{bmatrix}$.

The diagonal processors P_{ii} (i = 1, $\cdots$, $\frac{n}{2}$) act differently from the off-diagonal processors P_{ij} ($i \neq j$, $1 \leq i,j \leq \frac{n}{2}$). At each time step the diagonal processors P_{ii} compute rotation pairs (c_i^L, s_i^L) and (c_i^R, s_i^R) to annihilate their off-diagonal elements β_{ii} and γ_{ii} (cf. Algorithm USVD) :

$$\begin{bmatrix} c_i^L & s_i^L \\ -s_i^L & c_i^L \end{bmatrix}^T \begin{bmatrix} \alpha_{ii} & \beta_{ii} \\ \gamma_{ii} & \delta_{ii} \end{bmatrix} \begin{bmatrix} c_i^R & s_i^R \\ -s_i^R & c_i^R \end{bmatrix} = \begin{bmatrix} \alpha_{ii}' & 0 \\ 0 & \delta_{ii}' \end{bmatrix} .$$

To complete the rotations which annihilate β_{ii} and γ_{ii} (i = 1, $\cdots$, $\frac{n}{2}$) the off-diagonal processors P_{ij} ($i \neq j$) must perform the transformations

$$\begin{bmatrix} \alpha_{ij} & \beta_{ij} \\ \gamma_{ij} & \delta_{ij} \end{bmatrix} \leftarrow \begin{bmatrix} \alpha_{ij}' & \beta_{ij}' \\ \gamma_{ij}' & \delta_{ij}' \end{bmatrix} ,$$

where

$$\begin{bmatrix} \alpha_{ij}' & \beta_{ij}' \\ \gamma_{ij}' & \delta_{ij}' \end{bmatrix} = \begin{bmatrix} c_i^L & s_i^L \\ -s_i^L & c_i^L \end{bmatrix}^T \begin{bmatrix} \alpha_{ij} & \beta_{ij} \\ \gamma_{ij} & \delta_{ij} \end{bmatrix} \begin{bmatrix} c_j^R & s_j^R \\ -s_j^R & c_j^R \end{bmatrix} .$$

We assume that the diagonal processor P_{ii} broadcasts the rotation parameters (c_i^L, s_i^L) to other processors on the i-th row, and (c_i^R, s_i^R) to other processors on the i-th column in constant time, so that the off-diagonal processor P_{kj} has access to the parameters (c_k^L, s_k^L) and (c_j^R, s_j^R) when required.

To complete a step, columns and corresponding rows are interchanged between adjacent processors so that a new set of n off-diagonal elements is ready to be annihilated by the diagonal processors during the next time step. We have assumed diagonal connections for convenience. They can easily be simulated if only horizontal and vertical connections are available (cf. [2]). The diagonal outputs and inputs are illustrated in Figures 1, where the subscripts (i,j) are omitted. They are connected in the obvious way, as shown in Figure 2. For example,

$$out\,\beta_{ij} \text{ is connected to } \begin{cases} in\,\gamma_{i-1,j+1} & \text{if } i>1,\ j<n/2 \\ in\,\alpha_{i,j+1} & \text{if } i=1,\ j<n/2 \\ in\,\delta_{i-1,j} & \text{if } i>1,\ j=n/2 \\ in\,\beta_{i,j} & \text{if } i=1,\ j=n/2 \end{cases}$$

Formally, we can specify the diagonal interchanges performed by processor P_{ij} as follows.

Algorithm Interchange.

if i=1 and j=1 then $\begin{bmatrix} out\,\alpha \leftarrow \alpha;\ out\,\beta \leftarrow \beta; \\ out\,\gamma \leftarrow \gamma\ ;\ out\,\delta \leftarrow \delta; \end{bmatrix}$

else if i=1 then $\begin{bmatrix} out\,\alpha \leftarrow \beta;\ out\,\beta \leftarrow \alpha; \\ out\,\gamma \leftarrow \delta\ ;\ out\,\delta \leftarrow \gamma; \end{bmatrix}$

else if j=1 then $\begin{bmatrix} out\,\alpha \leftarrow \gamma:\ out\,\beta \leftarrow \delta; \\ out\,\gamma \leftarrow \alpha;\ out\,\delta \leftarrow \beta; \end{bmatrix}$

else $\begin{bmatrix} out\,\alpha \leftarrow \delta;\ out\,\beta \leftarrow \delta; \\ out\,\gamma \leftarrow \beta;\ out\,\delta \leftarrow \alpha; \end{bmatrix}$

{ wait for outputs to propagate to inputs of adjacent processors }

$$\begin{bmatrix} in\,\alpha \leftarrow \alpha\ ;\ in\,\beta \leftarrow \beta\ ; \\ in\,\gamma \leftarrow \gamma\ \ ;\ in\,\delta \leftarrow \delta\ . \end{bmatrix}$$

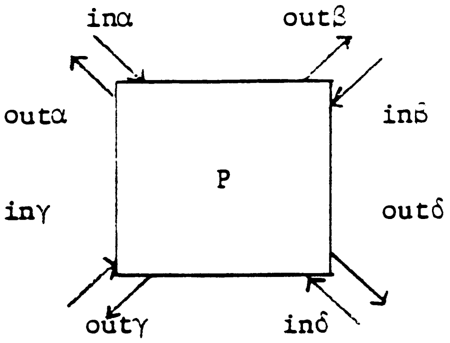

Figure 1: Diagonal input and output lines for processor P_{ij}

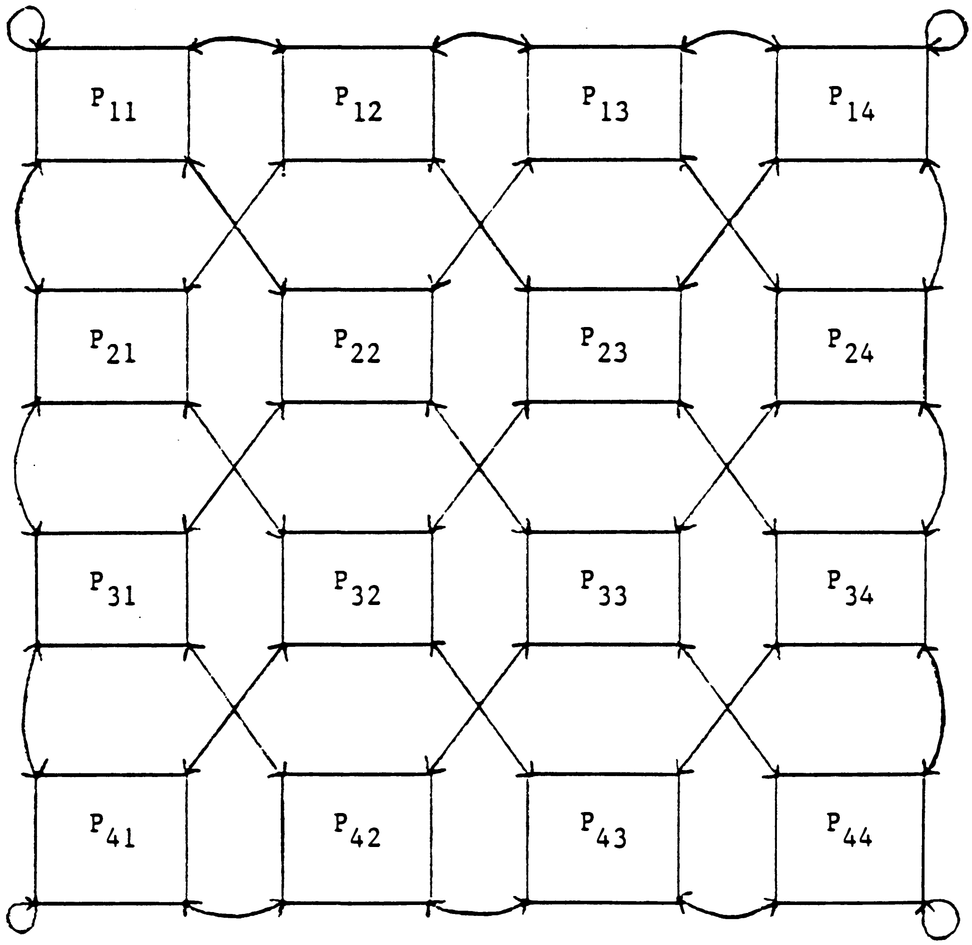

Figure 2: "Diagonal" connections, n = 8

(here $\longleftrightarrow$ stands for $\rightleftarrows$)

It is clear that a diagonal processor P_{ii} might omit rotations if its off-diagonal elements β_{ii} and γ_{ii} were sufficiently small. All that is required is to broadcast $(c_i^L, s_i^L) = (1,0)$ and $(c_i^R, s_i^R) = (1,0)$ along processor row and column i, respectively. As is well known (cf. [1] and [2]) a suitable threshold strategy guarantees convergence, although we do not know any example for which our algorithm fails to give convergence even without a threshold strategy.

If singular vectors are required, the matrices U and V of singular vectors can be accumulated at the same time as A is being diagonalized. To compute U each systolic processor P_{ij} ($1 \le i,j \le \frac{n}{2}$) needs four additional memory cells $\begin{bmatrix} \mu_{ij} & \nu_{ij} \\ \sigma_{ij} & \tau_{ij} \end{bmatrix}$ During each step it updates

$$\begin{bmatrix} \mu_{ij} & \nu_{ij} \\ \sigma_{ij} & \tau_{ij} \end{bmatrix} \leftarrow \begin{bmatrix} \mu_{ij} & \nu_{ij} \\ \sigma_{ij} & \tau_{ij} \end{bmatrix} \begin{bmatrix} c_j^L & s_j^L \\ -s_j^L & c_j^L \end{bmatrix} .$$

Each processor transmits its $\begin{bmatrix} \mu & \nu \\ \sigma & \tau \end{bmatrix}$ values to its adjacent processors in the same way as the $\begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$ values (see Algorithm Interchange). Initially we set $\mu_{ij} = \nu_{ij} = \sigma_{ij} = \tau_{ij} = 0$ ($i \neq j$), and $\mu_{ii} = \tau_{ii} = 1$, $\sigma_{ii} = \nu_{ii} = 0$. After a sufficiently large and integral number of sweeps, we have U defined by

$$\begin{bmatrix} u_{2i-1,2j-1} & u_{2i-1,2j} \\ u_{2i,2j-1} & u_{2i,2j} \end{bmatrix} = \begin{bmatrix} \mu_{ij} & \nu_{ij} \\ \sigma_{ij} & \tau_{ij} \end{bmatrix} .$$

The matrix V can be computed in an identical manner, but the rotation $\begin{bmatrix} c_j^L & s_j^L \\ -s_j^L & c_j^L \end{bmatrix}$ is replaced by $\begin{bmatrix} c_j^R & s_j^R \\ -s_j^R & c_j^R \end{bmatrix}$.

If the dimension n is odd we either modify the algorithm for processors P_{1i} and P_{i1} (i = 1, $\cdots$, $\left\lceil \frac{n}{2} \right\rceil$) in a manner analogous to that used in [1] and [2], or simply border A by a zero row and column. The details of handling problems where n is larger or smaller than the effective dimension of the array are presented in the next section.

Instead of broadcasting the parameter pairs (c_i^L, s_i^L) and (c_i^R, s_i^R) it may be preferable to broadcast only the corresponding tangent values t_i^L and t_i^R and let each off-diagonal processor P_{ij}

compute (c_i^L,s_i^L) and (c_j^R,s_j^R) from t_i^L and t_j^R (cf. (2.4)). Communication costs are thus reduced at the expense of requiring off-diagonal processors to compute two square roots per time step. This is not significant since the diagonal processors must compute three square roots per step in any case . In what follows a "rotation parameter" may mean either t_i or the pair (c_i,s_i) (superscripts omitted).

Let us show how we may avoid the idealization that the rotation parameters are broadcast along processor rows and columns in constant time. We can retain time $O(n)$ per sweep for our algorithm by merely transmitting rotation parameters at constant speed between adjacent processors.

Let $\Delta_{ij} = |$ i - j $|$ denote the distance of processor P_{ij} from the diagonal. The operation of processor P_{ij} will be delayed by Δ_{ij} time units relative to the operation of the diagonal processors, in order to allow time for the rotation parameters to be propagated at unit speed along each row and column of the processor array.

A processor cannot commence a rotation until data from earlier rotations are available on all its input lines. Thus, processor P_{ij} needs data from its four neighbors $P_{i\pm 1,j\pm 1}$ ($1 < i, j < \frac{n}{2}$). For the other cases see Figure 2 . Since

$$| \Delta_{ij} - \Delta_{i\pm 1,j\pm 1} | \leq 2$$

it is sufficient for processor P_{ij} to be idle for two time steps while waiting for the processors $P_{i\pm 1,j\pm 1}$ to complete their (possibly delayed) steps. Thus, the price paid to avoid broadcasting is that each processor is active for only one-third of the total computation. A similar inefficiency occurs with many other systolic algorithms, see, e.g., [2] and [15]. The fraction can be increased to almost unity if the rotation parameters are propagated at greater than unit speed.

A typical processor P_{ij} ($1 < j \leq i < \frac{n}{2}$) has input and output lines as shown in Figure 3 (with subscripts (i,j) or (i,i) omitted). Figure 3 differs from Figure 1 in that it shows the horizontal and vertical lines *inht*, *outht*, ,*invt*, *outvt* for transmitting the rotation parameters. Processors interconnect as shown in Figure 4.

Suppose that the matrix A is available in the systolic array at time T = 0. Then the operation of processor P_{ij} proceeds as described in Algorithm Processor. We also assume that each time step has nonoverlapping read and write phases; the result of a write at step T should be available at the read phase of step T+ 1, T+ 2 and T+ 3 in a neighboring processor, but should not interfere with a read at step T in a neighboring processor. The first time steps at which data are available on various processors' input lines are indicated in Figure 4.

Algorithm Processor.

```
if ( T ≥ Δ ) and ( T - Δ ≡ 0 ( mod 3 ) ) then
   begin
   if T ≠ Δ then [α β; γ δ] ← [inα inβ; inγ inδ] ;
   if Δ = 0 then { diagonal processor }
      Use (3.5) to determine t^L, t^R and [α β; γ δ] ← [α' 0; 0 δ']
   else { off-diagonal processor }
      begin
      t^L ← inht ; t^R ← invt ;
      Use (2.4) to recover (c^L,s^L) and (c^R,s^R);
      [α β; γ δ] ← [c^L s^L; -s^L c^L]^T [α β; γ δ] [c^R s^R; -s^R c^R]
      end;
   outht ← t^L ; outvt ← t^R ;
   if i > j then set outβ as in Algorithm Interchange;
   if i < j then set outγ as in Algorithm Interchange
   end
else if ( T ≥ Δ ) and ( T - Δ ≡ 1 ( mod 3 ) ) then
   begin
   if ( i = 1 ) or ( j = 1 ) then set outα as in Algorithm Interchange;
   if ( i = n/2 ) or ( j = n/2 ) then set outδ as in Algorithm Interchange
   end
else if ( T ≥ Δ ) and ( T - Δ ≡ 2 ( mod 3 ) ) then
   begin
   if ( i > 1 ) and ( j > 1 ) then set outα as in Algorithm Interchange;
   if i ≤ j then set outβ as in Algorithm Interchange;
   if i ≥ j then set outγ as in Algorithm Interchange;
   if ( i < n/2 ) and ( j < n/2 ) then set outδ as in Algorithm Interchange
   end
else
   Do nothing this time step.
```

Where the matrix assignments are:

$$\begin{bmatrix}\alpha & \beta\\ \gamma & \delta\end{bmatrix} \leftarrow \begin{bmatrix} in\alpha & in\beta\\ in\gamma & in\delta\end{bmatrix}; \quad \begin{bmatrix}\alpha & \beta\\ \gamma & \delta\end{bmatrix} \leftarrow \begin{bmatrix}\alpha' & 0\\ 0 & \delta'\end{bmatrix}; \quad \begin{bmatrix}\alpha & \beta\\ \gamma & \delta\end{bmatrix} \leftarrow \begin{bmatrix}c^L & s^L\\ -s^L & c^L\end{bmatrix}^T \begin{bmatrix}\alpha & \beta\\ \gamma & \delta\end{bmatrix} \begin{bmatrix}c^R & s^R\\ -s^R & c^R\end{bmatrix}$$

Subdiagonal $(1 < j < i < n/2)$ Diagonal $(1 < i < n/2)$

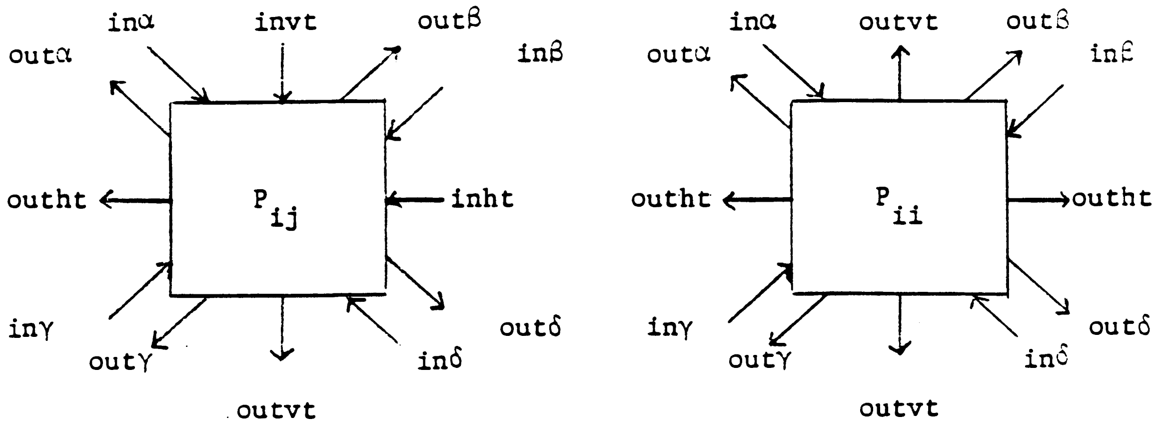

Figure 3: Input and output lines for typical subdiagonal and diagonal processors

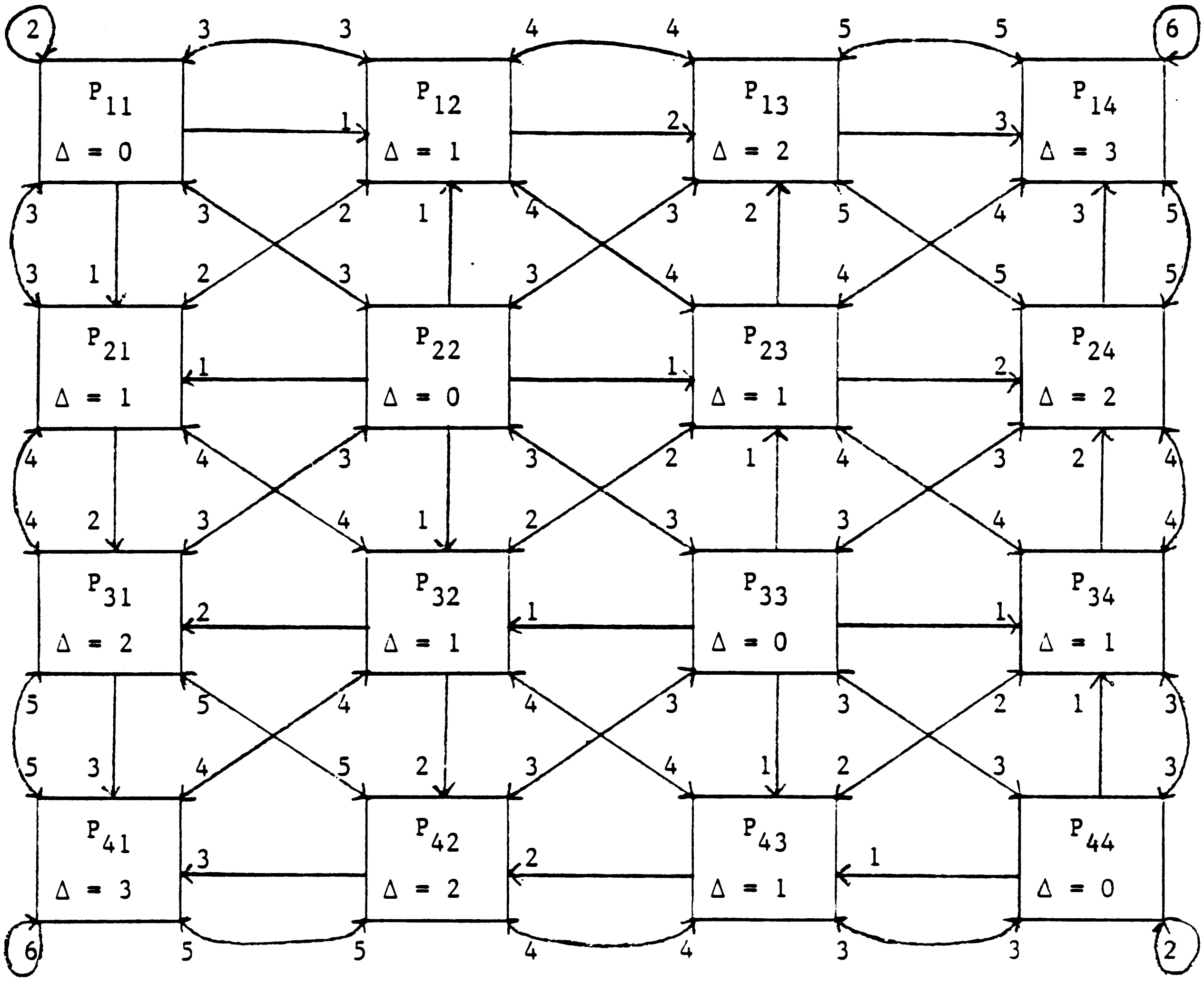

Figure 4: Interprocessor connections $(n = 8)$

(The first times at which inputs are available are indicated.)

Algorithm Processor does not compute singular vectors, but may easily be modified to do so. We have also omitted a termination criterion. The simplest is to perform a fixed number S (say conservatively 10) sweeps; then processor P_{ij} halts when $T = 3S(n-1) + \Delta_{ij} + 3$, since a sweep takes $3(n\text{-}1)$ time steps. A more sophisticated criterion is to stop when no nontrivial rotations were performed during the previous sweep. This requires communication along the diagonal, which can be done in $n/2$ time steps.

6. Handling Rectangular, Undersized, and Oversized Problems

In the previous two sections we assume that the matrix A is square. If A has more rows than columns then one of two strategies can be adopted.

One approach is to apply our "square" algorithm to the matrix

$$\bar{A} = [A, 0] = (\bar{a}_{ij}) \in \mathbf{R}^{m\mathrm{x}m}.$$

Note that in the SVD problem:

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} \bar{a}_{pp} & \bar{a}_{pq} \\ \bar{a}_{qp} & \bar{a}_{qq} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix},$$

we will have $\begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ whenever $q > n$, because the symmetrization step (3.2) will diagonalize the two-by-two matrix. (See equation (3.4).) Thus, the sparsity structure of $\bar{A}$ will be preserved during the iteration., and we will emerge with the factorization,

$$U^T \begin{bmatrix} A & 0 \end{bmatrix} \begin{bmatrix} V & 0 \\ 0 & I \end{bmatrix} = diag(\sigma_1, \cdots, \sigma_n, 0, \cdots, 0),$$

from which we get

$$U^T A V = diag(\sigma_1, \cdots, \sigma_n)$$

as the desired SVD. A drawback with this approach is the expanded dimension of $\bar{A}$, especially painful if $m >> n$. On the other hand, no additional hardware is required so long as the array of Section 5 can handle m-by-m problems.

In contrast, the second approach to handling rectangular SVD problems is more efficient but entails the interfacing of a SVD array with a QR array. The idea is to first compute the QR factorization of A:

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where $R \in \mathbf{R}^{n\mathrm{x}n}$ is upper triangular, and then compute the square SVD:

$$W^T R V = \Sigma = diag(\sigma_1, \cdots, \sigma_n).$$

Defining

$$U = Q \begin{bmatrix} W & 0 \\ 0 & I \end{bmatrix},$$

we get

$$U^T A V = diag(\sigma_1, \cdots, \sigma_n).$$

Actually,in the conventional one-processor setting, it is advisable to first compute the QR factorization anyway, especially if $m >> n$. See Chan[3].

We have not looked into the details of the interface between the QR and the SVD arrays. It would be preferable to have a single array with enough generality to carry out both phases of the computation. For example, perhaps the QR factorization could be obtained as A is "loaded" into the combination QR-SVD array.

We conclude with some remarks about the handling of SVD problems whose dimension differ from the effective dimension of the systolic array of Section 5. To fix the discussion, suppose that A is an n-by-n matrix whose SVD we want and that our array can handle SVD problems with maximum dimension N.

If $n < N$ then it is natural to have the array compute the SVD of

$$\bar{A} = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix},$$

i.e.,

$$\bar{U}^T \bar{A} \bar{V} = diag(\sigma_1, \cdots, \sigma_n, 0, \cdots, 0).$$

Our method of computing this SVD (based on JSVD) ensures that

$$\bar{U} = \begin{bmatrix} U & 0 \\ 0 & I \end{bmatrix} \quad \text{and} \quad \bar{V} = \begin{bmatrix} V & 0 \\ 0 & I \end{bmatrix},$$

whence $U^T A V = diag(\sigma_1, \cdots, \sigma_n)$. This follows because of (3.4) and (3.5). (Whenever a two-by-two SVD involves an index greater than n, the special form of the orthogonal update guarantees that $\bar{A}$'s block structure is preserved.) Let us point out that an SVD procedure needs not produce a $\bar{U}$ and a $\bar{V}$ matrix with the above block structure in the case

$rank(\bar{A}) = rank(A) < n$. For example, if N=3 and $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, then one of the infinitely many SVD's of $\bar{A}$ is

$$\begin{bmatrix} p & p^2 & p^2 \\ p & -p^2 & -p^2 \\ 0 & -p & p \end{bmatrix}^T \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p & p^2 & p^2 \\ p & -p^2 & -p^2 \\ 0 & -p & p \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

where $p = \frac{1}{\sqrt{2}}$. Thus, further computations are necessary before the SVD of A can be obtained.

Lastly, we discuss how oversized SVD problems might be handled. Partition the matrix $A \epsilon \mathbf{R}^{n \times n}$ so that

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1k} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ A_{k1} & \cdots & A_{kk} \end{bmatrix},$$

where each A_{ij} is $\frac{N}{2}$-by-$\frac{N}{2}$. (Assume that N, the dimension of the systolic array, is even so that $n = kN/2$.) One way to compute the SVD of A might be a block Jacobi scheme. In this scheme we repeatedly pick (p,q) satisfying $1 \le p < q \le k$ and use the array of Section 5 to compute the N-by-N SVD:

$$\begin{bmatrix} U_{pp} & U_{pq} \\ U_{qp} & U_{qq} \end{bmatrix}^T \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix} = \begin{bmatrix} D_p & 0 \\ 0 & D_q \end{bmatrix}. \tag{6.1}$$

Next, we construct the n-by-n orthogonal matrix U so that it is equal to the identity matrix except for the four strategic blocks in the (p,p), (p,q), (q,p) and(q,q) positions. Those blocks assume the values as given by (6.1). The n-by-n orthogonal matrix V is constructed in an identical manner. Then the matrix $B = U^T A V$ will have the property that

$$off(B) = off(A) - 2||A_{pq}||_F^2 - off(A_{pp}) - off(A_{qq}).$$

The indices (p,q) can be chosen according to the "row" or the "parallel" ordering. In the latter case we could exploit a block systolic array:

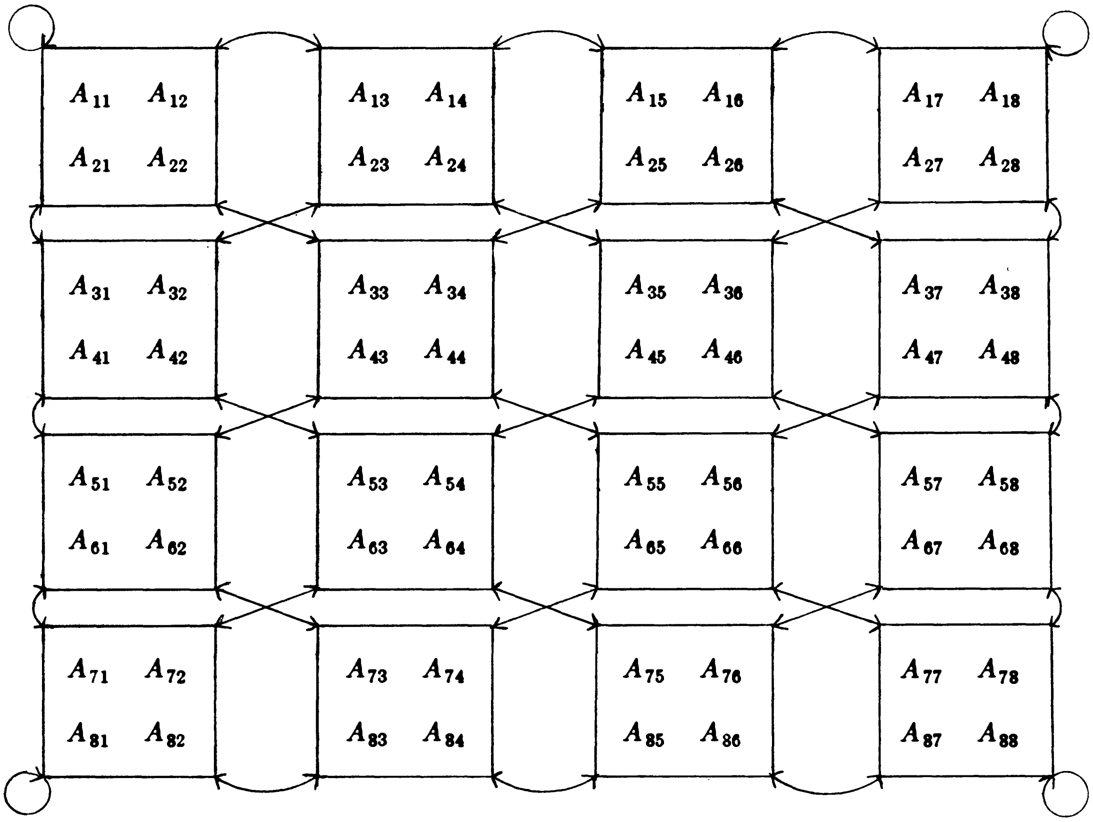

Figure 5: Interarray connections ($k = 8$)

(The initial state is illustrated)

The diagonal arrays are SVD arrays. The off-diagonal arrays are matrix-multiply arrays. In this scheme, the blocks move around this array of arrays in exactly the same fashion as the a_{ij} do in the array of Section 5. Obviously, the success of this technique will depend upon the nature of the interconnections, and a host of other unexamined issues.

7. References

[1] R.P. Brent and F.T. Luk, "A systolic architecture for the singular value decomposition", Technical Report TR-CS-82-522, Department of Computer Science, Cornell University, 1982.

[2] R.P. Brent and F.T. Luk, "A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem", Technical Report TR-CS-82-525, Department of Computer Science, Cornell University, 1982.

[3] T.F. Chan, "An improved algorithm for computing the singular value decomposition ", *ACM Trans. Math. Softw. 8* (1982), 72-83.

[4] J.J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[5] A.M. Finn, F.T. Luk, and C. Pottle, "Systolic array computation of the singular value decomposition", *Proc. SPIE Symp. East 1982, Vol. 341, Real-Time Signal Processing V*, 1982.

[6] G.E. Forsythe and P. Henrici, "The cyclic Jacobi method for computing the principal values of a complex matrix", *Trans. Amer. Math. Soc. 94* (1960), 1-23.

[7] B.S. Garbow, J.M. Boyle, J.J. Dongarra and C.B. Moler, *Matrix Eigensystem Routines - EISPACK Guide Extension*, Springer-Verlag, Berlin, 1977.

[8] W.M. Gentleman and H.T. Kung, "Matrix triangularization by systolic arrays", *Proc. SPIE Symp. 1981, Vol. 298, Real-Time Signal Processing IV*, 1981.

[9] G.H. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix", *J. SIAM Ser. B: Numer. Anal. 2* (1965), 205-224.

[10] G.H. Golub and F.T. Luk, "Singular value decomposition: applications and computations", *Trans. 22nd Conf. Army Mathematicians, ARO Report 77-1* (1977), 577-605.

[11] G.H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions", in [23], 134-151.

[12] E.R. Hansen, "On cyclic Jacobi methods", *J. Soc. Indus. Appl. Math 11* (1963), 448-459.

[13] D.E. Heller and I.C.F. Ipsen, "Systolic networks for orthogonal equivalence transformations and their applications", *Proc. 1982 Conf. on Advanced Research in VSLI*, MIT, Cambridge, Massachusetts (1982), 113-122.

[14] M.R. Hestenes, "Inversion of matrices by biorthogonalization and related results", *J. Soc. Indust. Appl. Math 6* (1958), 51-90.

[15] H.T. Kung, "Why systolic architectures?", *IEEE Computer 15* (1982),37-46.

[16] S.Y. Kung and R.J. Gal-Ezer, "Linear or square array for eigenvalue and singular value decompositions?", *Proc. USC Workshop on VLSI and Modern Signal Processing*, Los Angeles, California (Nov. 1982), 89-98.

[17] F.T. Luk, "Computing the singular-value decomposition on the ILLIAC IV", *ACM Trans. Math. Softw. 6* (1980), 524-539.

[18] H. Rutishauser, "The Jacobi method for real symmetric matrices", in [23], 202-211.

[19] A.H. Sameh, "On Jacobi and Jacobi-like algorithms for a parallel computer", *Math. Comput. 25* (1971), 579-590.

[20] R. Schreiber, "A systolic architecture for singular value decomposition", *Proc. 1st Internat. Coll. on Vector and Parallel Computing in Scientific Applications*, Paris, France (Mar. 1983), to appear.

[21] J.M. Speiser and H.J. Whitehouse, "Architectures for real-time matrix operations", *Proc. 1980 Government Microcircuits Applications Conf.,* Houston, Texas (Nov. 1980).

[22] H.J. Whitehouse, J.M. Speiser and K. Bromley, "Signal processing applications of systolic array technology", *Proc. USC Workshop on VLSI and Modern Signal Processing,* Los Angeles, California (Nov. 1982), 5-10.

[23] J.H. Wilkinson, "Note on the quadratic convergence of the cyclic Jacobi process", *Numer. Math 4* (1962), 296-300.

[24] J.H. Wilkinson and C. Reinsch, eds., *Handbook for Automatic Compuation, Vol. 2 (Linear Algebra),* Springer-Verlag, New York, 1971.