Computing the Singular Value Decomposition on a Distributed System of Vector Processors†

Christian Bischof

87-869
(Revision of TR 86-798)

September 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†This work was supported by the U.S. Army Research Office through the Mathematical Sciences Institute of Cornell University, by the Office of Naval Research under contract N00014-83-K-0640, by NSF contract CCR 86-02310 and by the IBM Corporation. Computations were performed at IBM Kingston and the National Supercomputer Facility at Cornell which is supported in part by the National Science Foundation and IBM. This report has been submitted to *Parallel Computing*.

Computing the Singular Value Decomposition on a Distributed System of Vector Processors [†]

Christian Bischof

Department of Computer Science
Upson Hall
Cornell University
Ithaca, New York 14853

Abstract. Jacobi methods for computing the singular value decomposition (SVD) of a matrix are ideally suited for multiprocessor environments due to their inherent parallelism. In this paper we show how a block version of the two-sided Jacobi method can be used to compute the SVD efficiently on a distributed architecture. We compare two variants of this method that differ mainly in the degree to which they diagonalize a given subproblem. The first method is a true block generalization of the scalar scheme in that each off-diagonal block is completely annihilated. The second method is a scalar Jacobi algorithm reorganized in such a manner that it conforms to the block decomposition of the problem. We have performed experiments on the Loosely Coupled Array Processor (LCAP) system at IBM Kingston which for the purposes of this article can be viewed as a ring of up to ten FPS-164/MAX array processors. These experiments show that the block Jacobi algorithm performs very well on a distributed system, especially when the processors have vector processing hardware. As an example, we were able to achieve a sustained performance of 159 MFlops on a 960×720 SVD problem using eight processors. A surprising outcome of these experiments is that the determining factor for the performance of the algorithm on high-performance architectures is the subproblem solver, not the communication overhead of the algorithm.

[†]This work was supported by the U.S. Army Research Office through the Mathematical Sciences Institute of Cornell University, by the Office of Naval Research under contract N00014-83-K-0640, by NSF contract CCR 86-02310 and by the IBM Corporation. Computations were performed at IBM Kingston and the National Supercomputer Facility at Cornell which is supported in part by the National Science Foundation and IBM.

1. Introduction. The singular value decomposition (SVD) of an $m \times n$ matrix A $(m \geq n)$ is a very versatile and stable decomposition [17]. In the SVD we seek orthonormal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ such that

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

The σ_i are called the singular values of A; the columns of U and V are called left and right singular vectors, respectively. It is usually assumed that $\sigma_1 \geq \ldots \geq \sigma_n \geq 0$, but we do not insist on this normalization.

In the scalar Jacobi algorithm for a square matrix A we repeatedly compute sine-cosine pairs c_1, s_1 and c_2, s_2 such that

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}. \tag{1.1}$$

The elements a_{ij} are usually chosen in a cyclic-by-row ordering, but other orderings are possible if one incorporates a threshold criterion into the algorithm, i.e. no rotations are generated if a_{ij} is already sufficiently small. For details see [16] and [26]. Parallel SVD algorithms using scalar Jacobi rotations have recently received much attention for real-time signal processing on systolic arrays [7,20, 24].

For coarse-grained high-performance environments, it is well known that block algorithms are advantageous [5,12]. Given a $r \times r$ matrix block, the cost of data movement is $O(r^2)$, whereas typically $O(r^3)$ operations are performed using these data. Dongarra and Sorensen [14] call this "surface-to-volume effect" and is important in high performance environments where the data transfer rate is low compared to the arithmetic speed due to a memory hierarchy or an explicitly parallel architecture. Furthermore block algorithms typically lead to programs that are rich in matrix-vector and matrix-matrix operations. The regularity of these operations ensures excellent performance on machines with special vector hardware - almost a standard feature in current high-performance architectures. For an overview see [13].

To set the stage for the block Jacobi algorithm, consider A as a $k \times k$ block matrix (A_{ij}), $i,j = 1,\ldots,k$. The ith block column of A is denoted by A_i. It is not required that the blocks be square or all of the same size. The idea behind the two-sided block Jacobi approach is to make A block diagonal by solving a series of subproblems. To measure how close A is to the desired block diagonal form, we define $OFF(A)$ as the Frobenius norm of the off-diagonal blocks,

$$OFF(A) \equiv \mathrm{sqrt}\Big(\sum_{i \neq j} \|A_{ij}\|_F^2\Big).$$

We consider A to be block-diagonal if

$$OFF(A) = O(\varepsilon \|A\|_F).$$

Once A is block diagonal, it is straightforward to compute the SVD of A by computing the SVD of each diagonal block.

To decrease $OFF(A)$ we repeatedly consider two-by-two block subproblems S_{ij} of the form

$$S_{ij} \equiv \begin{bmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{bmatrix} . \tag{1.2}$$

We can choose orthogonal matrices W_{ij} and Z_{ij} such that for

$$\begin{bmatrix} B_{ii} & B_{ij} \\ B_{ji} & B_{jj} \end{bmatrix} \equiv W_{ij}^T S_{ij} Z_{ij}$$

we have

$$\|B_{ij}\|_F^2 + \|B_{ji}\|_F^2 \le \theta^2 (\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2) \tag{1.3}$$

for some fixed $0 \le \theta < 1$. If B is the matrix that we obtain by updating block rows and block columns i and j of A with W_{ij}^T and Z_{ij} respectively, then it is not hard to see that

$$OFF(B)^2 \le OFF(A)^2 - (1-\theta^2)(\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2).$$

So by solving a judiciously chosen sequence of two-by-two block subproblems we will be able to reduce A to block diagonal form. The most common sequence is the cyclic-by-row ordering [17, p. 299], but as will be seen later it is not suitable for parallel algorithms.

Another SVD Jacobi method is the one-sided Jacobi method, originally described by Hestenes [18]. By implicitly applying the two-sided Jacobi method just described to $A^T A$, this algorithm computes an orthonormal $n \times n$ matrix V such that

$$Q \equiv AV \equiv [q_1, \ldots, q_n]$$

is an $m \times n$ matrix with orthogonal columns q_i. The short form of the SVD of A is then obtained by setting

$$\sigma_i \equiv \|q_i\|_2 , \; i = 1, \ldots, n \tag{1.4}$$

and scaling the columns of Q

$$U \equiv Q D^{-1}, \text{ where } D \equiv diag(\sigma_1, \ldots, \sigma_n) . \tag{1.5}$$

The one-sided Jacobi method requires somewhat less work than the two-sided one. Luk (1980) implemented it on the ILLIAC IV. More recently, Berry and Sameh [3] performed numerical experiments with a scalar Hestenes scheme on an eight processor Alliant FX/8 shared-memory multiprocessor machine. They found it to be faster than the LINPACK routines yet somewhat less accurate. We chose the two-sided approach as it is perfectly stable, handles rank deficiency, computes left **and** right singular vectors, and can easily be modified to solve the symmetric eigenvalue problem.

The paper is organized as follows. Section 2 introduces the parallel block Jacobi algorithm. In the next section we discuss the subtleties and difficulties associated with solving subproblem (1.2). Section 4 describes the two different approaches that we implemented. In section 5 we give a brief overview of the LCAP system and present our experimental results. Conclusions are offered in the final section.

2. The Block Jacobi Method on a Multiprocessor. The block Jacobi method has been studied in detail by several researchers [23,25]. The method proceeds in *sweeps*. In a sweep, each subproblem S_{ij} is solved exactly once. To guard against pathological cases (see section 3) and ensure convergence of the algorithm, it is necessary to incorporate a threshold criterion into the algorithm. Following Van Loan [25], we skip solving the S_{ij} subproblem if

$$\mu_{ij} \equiv \mathrm{sqrt}(\|A_{ij}\|_F^2 + \|A_{ji}\|_F^2) \leq \tau \tag{2.1}$$

for a positive subproblem threshold τ.

For a given ordering

$$(i_1, j_1), \ldots, (i_r, j_r), \; r = \tfrac{1}{2}k(k-1)$$

of the off-diagonal index pairs, the block Jacobi algorithm on one processor is described by Algorithm 1. For simplicity, we have assumed that the order in which subproblems are solved is the same in each sweep. The threshold criterion (2.1) ensures that in each sweep $OFF(A)$ is diminished by at least $\tau * \mathrm{sqrt}(1\text{-}\theta^2)$ and by choosing $\tau \leq \varepsilon \|A\|_F / k$, the termination criterion $OFF(A) < \varepsilon \|A\|_F$ is eventually satisfied. In practice this algorithm converges quadratically as might be expected from the quadratic convergence of the scalar algorithm. Note that this algorithm does not sort the singular values but if that is required, it can be done in time that is negligible to the overall cost of the algorithm [1,2].

If we have $p = \tfrac{1}{2}k$ processors available, we distribute A, U and V such that each processor holds two block columns (assume k even from now on). Each processor now has enough information to solve the subproblem that is defined by the two block columns of A it holds. The updates (2.2) can also be performed concurrently as each processor holds the required columns of A, U and V. Only for the update (2.3) each processor has to receive the W_{ij}'s of each other processor since rows of A are

Algorithm 1. Given a subproblem threshold $\tau \leq \varepsilon \parallel A \parallel_F / k$, the following algorithm computes orthogonal U and V such that $\mathrm{OFF}(U^T A V) \leq \varepsilon \parallel A \parallel_F$. The original A is overwritten with $U^T A V$.

$U \leftarrow I_m$; $V \leftarrow I_n$;
<u>while</u> $\mathrm{OFF}(A) > \varepsilon \parallel A \parallel_F$ <u>do</u>
  <u>for</u> $l = 1$ <u>to</u> $k(k\text{-}1)/2$ <u>do</u>
   $(i,j) \leftarrow (i_l, j_l)$
   <u>if</u> $\mu_{ij} \geq \tau$ <u>then</u>
    Solve subproblem S_{ij} yielding Z_{ij} and W_{ij}.

$$\left.\begin{array}{l} [A_i, A_j] \leftarrow [A_i, A_j] Z_{ij} \\ [U_i, U_j] \leftarrow [U_i, U_j] W_{ij} \\ [V_i, V_j] \leftarrow [V_i, V_j] Z_{ij} \end{array}\right\} \quad (2.2)$$

    <u>foreach</u> $q \in \{1,\ldots,k\}$ <u>do</u>

$$\begin{bmatrix} A_{iq} \\ A_{jq} \end{bmatrix} \leftarrow W_{ij}{}^T \begin{bmatrix} A_{iq} \\ A_{jq} \end{bmatrix} \quad (2.3)$$

    <u>end foreach</u>
   <u>end if</u>
  <u>end for</u>
<u>end while</u>

distributed over all the processors. Then we have to redistribute block columns among processors in order to solve a different set of subproblems in the next step.

To complete a sweep, we have to solve each of the $p(k-1)$ subproblems once. Each partition of the set $\{1,..,k\}$ in pairs of two defines a *rotation set*, i.e. a set of $p = \frac{1}{2}k$ subproblems that can be solved concurrently. By convention the ith processor P_i solves the subproblem defined by the ith tuple. The computations associated with one rotation set are referred to as a *stage*. A *parallel ordering* for a given distributed architecture is an ordering of subproblems such that a sweep is completed in $(k-1)$ stages. A parallel ordering is *optimal* for a given architecture if the redistributing of columns between stages requires only nearest-neighbour communication and each processor exchanges only one set of block columns between stages. Each processor has to exchange at least one set of block columns as otherwise it would solve the same subproblem again.

Various parallel orderings have been suggested for different topologies. Luk and Park [21] give a framework for several previously known orderings on a linear array of processors using a "caterpillar track" scheme. However, the linear array does not seem to allow for an optimal ordering. For a hypercube, Bischof [6] suggested an optimal parallel ordering. Since we were to perform our experiments on a ring architecture, we chose an optimal ordering for a ring suggested by Eberlein [15]. We demonstrate this ordering by going through one sweep for a 10-by-10 block matrix distributed over 5 processors.

Assuming that initially processor P_i contains block columns $2i-1$ and $2i$, $i = 1,\ldots,5$, we have the set-up shown in Fig. 1.a.

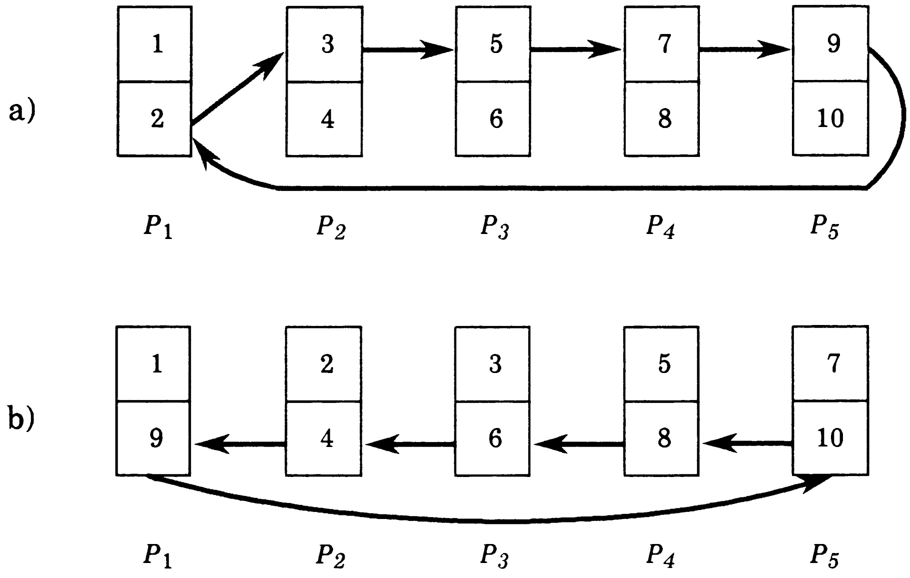

Fig. 1. *The Eberlein Ring Ordering*

So the first rotation set is

(1,2), (3,4), (5,6), (7,8), (9,10).

To prepare for the next stage, we shuffle block columns as indicated by the arrows in Fig. 1.a obtaining rotation set

(1,9), (2,4), (3,6), (5,8), (7,10).

Now we shuffle block columns as indicated by the arrows in Fig.1.b. This leads to rotation set

(1,4), (2,6), (3,8), (5,10), (7,9).

To generate the remaining rotation sets we alternate between these two ways for shuffling columns. The remaining subproblems are then solved in the following order where next to each stage it is denoted whether it resulted from a shuffle according to Fig..1.a or 1.b.

(1,7) (4,6), (2,8), (3,10), (5,9)	a
(1,6), (4,8), (2,10), 3,9), (5,7)	b
(1,5), (6,8), (4,10), (2,9), (3,7)	a
(1,8), (6,10), (4,9), (2,7), (3,5)	b

(1,3), (8,10), (6,9), (4,7), (2,5) a

(1,10), (8,9), (6,7), (4,5), (2,3) b

At this point we are finished with one sweep and to generate the first rotation set for the next sweep we shuffle according to Fig. 1.a to generate subproblems

(1,2), (10,9), (7,8), (6,5), (4,3)

and from there we continue as described. This example shows that this method for generating the subproblems in a sweep conforms to our definition of an optimal parallel ordering: We generate $k-1$ rotation pairs and every processor exchanges only one set of block columns between each rotation set. Due to the ring topology we only communicate between neighbours at any point. The orderings generated by this method differ from sweep to sweep, but every $2k$ rotations all block columns are back in their original position.

3. Importance of the Subproblem Solver. How we solve the subproblem (1.2) can have a dramatic effect on the convergence speed of the algorithm. A brief example illustrates this. Suppose A is a square symmetric matrix partitioned into four equal sized square blocks and distributed over two processors. The structure of A is as shown in Fig. 2. Here "ε" stands for a block with small norm, "★" is a block of non-negligible norm and blocks denoted by "●" are large in terms of norm with regard to the ε-blocks. The current subproblems are emphasized by a black box.

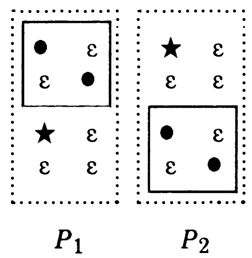

Fig. 2. *Initial Configuration*

Let P be the permutation that exchanges the first and second block row of a 2×2 block matrix. The subproblems in P_1 and P_2 are almost block diagonal. So orthogonal transformations $W=Z$ close to the identity I or close to P will solve either of the subproblems. Now if we let $W_1=Z_1$ close to I solve the subproblem in P_1 and $W_2=Z_2$ close to P solve the subproblem in P_2, then after the update A has the structure shown in Fig. 3.

If we exchange block columns according to Fig. 1.a and (for purposes of exposition) also exchange block rows accordingly, the structure of Fig. 3 remains preserved. Again a $U_1 = V_1$ close to the identity can solve the subproblem in P_1, a $U_2=V_2$ close to the block permutation P can solve the subproblem in P_2. Applying these transformations to A again leads to the structure of Fig. 2. It remains preserved

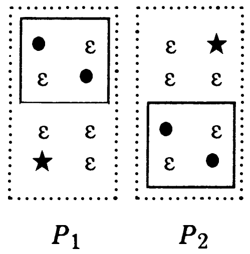

Fig. 3. *After the First Stage*

when we shuffle block columns and block rows according to Fig.1.b. So throughout a sweep, we have essentially not diminished $OFF(\mathrm{A})$ as the block ★ has not been reduced in a subproblem. This situation can continue for quite a number of sweeps until we eventually skip a subproblem due to the subproblem threshold criterion (2.1) and break this vicious cycle. But of course the overall convergence of the algorithm will be immeasurably slow.

To avoid this difficulty, our subproblem solver tries to compute orthogonal transformations that are close to the identity. In the scalar setting where we compute Givens rotations

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \text{ with } c \equiv \cos\psi,\ s \equiv \sin\psi$$

choosing a close-to-identity transformation means choosing a small rotation angle ψ. It is well known [16,22] that this choice is crucial for convergence of the scalar algorithm.

4. **Two Different Block Jacobi Methods.** Our first method is a true block method. It regards A as a square block matrix (possibly with rectangular blocks) and then uses a variant of the Chan SVD algorithm to diagonalize a given subproblem at each step. Thus the off-diagonal blocks are fully annihilated. The alternative method first reduces A to square form by applying a pipelined block QR algorithm based on the WY representation [5]. Then a regrouped scalar Jacobi algorithm is applied to the resulting square matrix. We apply one full scalar cyclic-by-row Jacobi sweep to the subproblems in the very first rotation set in every sweep. For all other subproblems, we generate Jacobi rotations only for the elements in the off-diagonal blocks.

To illustrate the first method, consider a subproblem S with 4-by-2 blocks. Using a variant of the SVD algorithm proposed by Chan [8] , we compute orthogonal matrices W and Z such that

$$W^T S Z = \Sigma$$

is diagonal. For the flop counts we assume that S is an $r \times c$ matrix. The included flop counts are estimates for the distribution of work in the algorithm. As high-performance computers usually have seperate adder and multiplier units, each multiplication and addition is counted as a separate flop.

Step 1: Use Householder transformations to compute an orthogonal matrix Q such that

$$Q^T S = \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \equiv R$$

The very first subproblem in each processor requires

$$4r^2c - 2rc^2 + \frac{2}{3}c^3$$

flops. Thereafter the diagonal blocks of S are diagonal and exploiting this fact we need only

$$3r^2c - 2rc^2 + \frac{5}{6}c^3$$

flops.

Step 2: Move the nonzero rows of R together and compute the SVD

$$U^T \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix} Z \equiv \Sigma$$

using Householder bidiagonalisation and QR iteration. The bidiagonalisation requires $16/3\ c^3$ flops, the QR iteration typically requires $7c^3$ flops.

Step 3: Permute off-diagonal entries of U and Z close to 1 in modulus to the diagonal to make U and Z close to identity transformations. This step typically requires just $O(c^2)$ flops, a small overhead which we can ignore in our flop count.

Step 4: Letting $U = (U_{ij})$, $i,j = 1,2$ with square blocks partitioned conforming to the block column partitioning of S, we set

$$W \equiv Q \begin{bmatrix} U_{11} & 0 & U_{12} & 0 \\ 0 & I & 0 & 0 \\ U_{21} & 0 & U_{22} & 0 \\ 0 & 0 & 0 & I \end{bmatrix}$$

This matrix-matrix multiplication requires $2rc^2$ flops.

Computing a "split" R in step 1 together with step 3 guarantees that the W computed in step 4 will be close to the identity. Note that off-diagonal blocks A_{ij} that have been zeroed in one subproblem will be filled in later when another subproblem updates block rows and columns i or j of A. Of course we suggested only one of many ways how to solve a rectangular subproblem. We chose this approach since in the QR factorisation we can exploit the fact that diagonal blocks are diagonal and because we can use the matrix-matrix multiplication hardware to execute step 4 very efficiently. Bischof and Van Loan [4] consider some other methods to solve a rectangular subproblem. Since this algorithm treats every off-diagonal block like a macro-element of a $k \times k$ matrix subjected to a scalar Jacobi method in that it fully annihilates it, ample experience with the scalar algorithm [7, 17, 25] suggests that it will take $O(\log k) = O(\log p)$ sweeps to converge.

For the second method, we first reduce A to square form (unless $m = n$) by a pipelined block QR algorithm based on the WY representation [5]. A detailed description of this algorithm will be reported elsewhere. The WY representation allows us to write the product

$$Q \equiv P_1 \ldots P_l$$

of l $n \times n$ Householder matrices in the form

$$Q = I + WY^T$$

where both W and Y are $n \times l$. Application of Q is then just two matrix-matrix multiplications, the operation of choice for processors with vector capabilities. In the pipelined block QR algorithm, each processor computes the QR factorization of a block column it contains. At the same time, it generates the WY factors determined by the Householder vectors reducing this subproblem. Once the subproblem is solved, it sends the WY factors to the right neighbour. It then receives and forwards $p-1$ WY factors generated by other processors and applies these to the block columns it houses until it is its turn again to solve the next subproblem and generate the next WY pair. While accumulating the WY factors, it computes the decomposition

$$A = Q \begin{bmatrix} H \\ 0 \end{bmatrix} \tag{4.1}$$

where H is an $n \times n$ square matrix. This reduction requires approximately

$$\frac{2}{p}\left(2m^2n - \frac{1}{3}n^3 \right)$$

flops on the average per processor and the transmission of

$$4mn - 2n^2$$

words per processor.

We then compute the SVD of H using a blocked scalar Jacobi method. In the very first rotation set in each sweep, every processor applies one scalar cyclic-by-row Jacobi sweep to its subproblem. In every other rotation set we apply Jacobi rotations only to the off-diagonal block, i.e. for a 4×4 subproblem, we generate point rotations

$$(1,3), (1,4), (2,3), (2,4).$$

We call this an off-diagonal Jacobi sweep. So our subproblems are of the form (1.1) and a variant of algorithm USVD in [7] is used in computing the Givens rotations. Since algorithm USVD performs point threshholding in solving subproblems of the form (1.1), the block thresholding criterion (2.1) can be omitted. If the subproblem is a $c \times c$ matrix, then applying a full Jacobi sweep and accumulating the rotations costs $12c^2$ flops, an off-diagonal sweep half as much.

It is easy to see that using this scheme every scalar subproblem (1.1) in H is solved exactly once during each sweep. This shows that we are applying one full scalar Jacobi sweep to H during each sweep. Except for the different ordering, this is much like the "AB supersweep" regime suggested by Schreiber [24]. Hence we expect this algorithm to take O(log$n$) sweeps until convergence.

After we have computed the SVD

$$H = U \Sigma V^T$$

of H, we permute the block columns back into their original positions by exploiting the $4p$-cyclicity of the Eberlein ordering and then update

$$Q \leftarrow Q \begin{bmatrix} U & 0 \\ 0 & I \end{bmatrix} \tag{4.2}$$

to complete the SVD of A. The update of Q requires $2n^2\ m/p$ flops and the transmission of

$$2(p-1)\frac{n^2}{p}$$

words per processor.

The main difference between these two methods is the degree of diagonalization we achieve in each subproblem. While the first approach always fully diagonalizes a subproblem. the second just applies one scalar Jacobi sweep to the whole matrix during each sweep. So in fact we are not even guaranteed that condition (1.2) will be satisfied in every subproblem. The threshold criterion (2.1) or alternatively point threshholding within the subproblem solver guarantees convergence, though. It is also clear that the second method will be more efficient whenever m is sufficiently larger than n although we expect it to take more sweeps to converge.

5. Numerical Results on the LCAP-1 System at IBM Kingston. The LCAP-1 system (LCAP is an acronym for Loosely Coupled Array Processors) consists of ten Floating Point Systems FPS-164/MAX processors connected by bulk memories manufactured by Scientific Computing Associates (SCA) and by two FPS buses. A simplified diagram of LCAP-1 is given in Fig. 4; the host machine is an IBM 3090 mainframe and is not shown here. For a detailed description see [10].

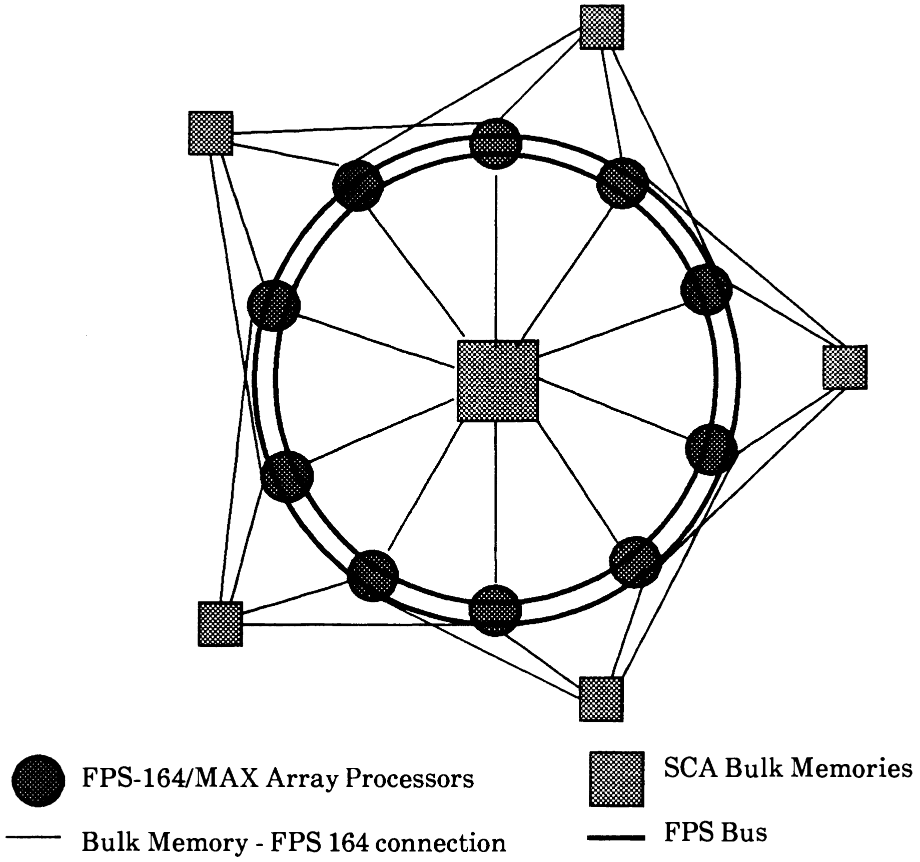

Fig. 4. *The LCAP-1 System at IBM Kingston*

The bulk memories can be used as a mailbox for message passing, as shared memory between processors and as fast out-of-core memory local to a processor. Together with the rich topology of the system, this provides for a very high degree of flexibility. The user can easily make use of the bulk memories through precompiler directives in his Fortran code. Different users can share the system through the use of a scheduler.

The FPS-164 is a general-purpose 64-bit attached scientific processor. It contains a pipelined floating-point unit allowing a theoretical peak performance of 11 MFlops. The FPS-164 can be augmented with so-called MAX boards. These boards are special purpose hardware performing only a limited number of operations, yet those at great speed [9]. The MAX boards are especially well suited for matrix-matrix multiplication and this is the only feature we are using; here each MAX board adds another 22 MFlops. Each FPS-164 on LCAP-1 is equipped with two MAX boards so the theoretical peak performance of LCAP-1 is 550 MFlops.

For the implementation of our SVD algorithm we regarded LCAP-1 as a ring of processors where processors do not share memory and communicate through message passing using the SCA memories. There are two types of communication required by the parallel version of Algorithm 1. Associated with each rotation set is a broadcast. Each processor must receive the orthogonal transformation of every other processor. This is accomplished in a merry-go-round fashion. The orthogonal matrices are passed around the ring and are applied to the block columns of A local to a processor at each "stop". We also use this broadcast to transmit convergence and threshhold information by piggybacking it onto the orthogonal matrix. After the update, the block columns have to be redistributed around the ring and here we exploit the nearest neighbour topology of the ring by using the Eberlein shuffle.

Let us now summarize the computational cost of our algorithms per processor. Assume we are solving an $m \times n$ problem on $p \geq 2$ processors and that the first method requires sub_1 subproblems, our second method sub_2 subproblems until convergence. Further define

$$sweep_i \equiv \frac{sub_i}{2p-1}, \; r \equiv \frac{m}{p} \text{ and } c \equiv \frac{n}{p}.$$

Note that $sweep_i$ is the number of sweeps required for convergence in the given method. Subproblems are of size $r \times c$ in the first method and of size $c \times c$ in the second method.

Using the first subproblem algorithm described in the previous section, we spend

$$f^1_{sub} \equiv 4r^2c + 16c^3 + (sub_1 - 1)(3r^2c + 11\frac{1}{6}c^3)$$

flops to solve the subproblem. The block column updates (2.2) and (2.3) require

$$f^1_{block} \equiv 2sub_1((m-r)c^2 + mr^2 + nc^2 + (m-r)rc)$$

flops. Communication requires the transmission of

$$w^1_{broad} \equiv 2\, sub_1\,(p-1)r^2$$

words for the merry-go-round broadcast and

$$w^1_{shuffle} \equiv sub_1\,(mc + mr + nc)$$

words for the Eberlein shuffle.

For the second method, steps (4.1) and (4.2) require approximately

$$f^2_{qr} \equiv \frac{2}{p}\,(2m^2n + mn^2 - \frac{1}{3}n^3)$$

flops as well as the transmission of

$$w^2_{qr} \equiv 2(2mn - \frac{1}{p}n^2)$$

words. We spend

$$f^2_{sub} \equiv (12\, sweep_2 + 6(sub_2 - sweep_2))\, c^3$$

flops to solve the subproblem. The block column updates require

$$f^2_{block} \equiv 4sub_2((n-c)c^2 + nc^2)$$

flops. Communication requires the transmission of

$$w^2_{broad} \equiv 2\, sub_2\,(p-1)c^2$$

words for the merry-go-round broadcast and

$$w^2_{shuffle} \equiv 3\, sub_2\, nc$$

words for the Eberlein shuffle. In situations where block columns are not in the processors they started out in, we use the fact that the Eberlein ordering is $4p$-cyclic to permute columns back into their original positions by doing a few extra Eberlein shuffles at the end.

For both methods, we have

$$f_{sub} \ll f_{block}$$

in most cases. Thus the algorithm is ideally suited for a parallel architecture where each node has special vector hardware. The reason is that block column updates (2.2) and (2.3) are straightforward matrix-matrix multiplications - the operation guaranteed to exploit fully the vector processing capability. Furthermore

$$f^i_{sub} = O(\frac{1}{p^2}) \tag{5.1}$$

$$f^i_{block} = O(\frac{1}{p}) \tag{5.2}$$

$$w^i_{broad} + w^i_{shuffle} = O(1) \tag{5.3}$$

for fixed m and n and $i = 1,2$. So for a problem of fixed size an increase in the number of processors shifts the computational load from solving the subproblem to updating the block columns. This is desirable as we expect the block column updates to execute much faster in the presence of a vector unit. The communication time should stay about the same with perhaps a slight penalty for issuing more sends and receives.

Our experiments on the LCAP-1 system led to some surprises. We mention that our code was written entirely in Fortran-77, using the assembler-coded BLAS and matrix-matrix multiplication routines available in the FPS mathematics library. The code was compiled with the apftn64 compiler under release G, using optimization level 3. We performed a series of tests with large matrices on up to ten processors. The

p	2	3	4	6	8	10
960×720	–	–	–	2010	1350	1270
960×480	–	–	1190	975	813	716
960×240	–	–	474	481	405	360
528×528	1660	1370	1020	772	613	–
528×264	381	325	270	256	–	–
528×120	123	130	119	102	–	–

a) Method 1

p	2	3	4	6	8
960×720	–	–	–	632	449
960×480	–	–	315	212	171
960×240	–	–	57	43	40
528×528	570	372	242	166	126
528×264	137	86	63	56	–
528×120	17	13	11	12	–

b) Method 2

Table 1: *Total Execution Time in Seconds*

test data were generated randomly with a uniform distribution on the interval [0,1]. In some sense this is the worst case for the Jacobi method as the diagonal blocks of the matrix are small compared to the rest of the matrix.

To assess computational performance, let M_{tot} be the sustained MFlop rate per processor including communication overhead. M_{sub} and M_{block} are the MFlop rates of the subproblem solver and the block column updates, respectively and T_{com} is the average transfer rate in MBytes/sec. Table 1 shows the total execution time required by the two methods. Due to technical difficulties we were not able to perform the ten processor runs for method 2. It is apparent that the second method is by far superior in in that it performs up to ten times faster than the first method. Note the slow-down for the 528×128 problem as we move from 4 to 6 processors.This is a common observation when too small a problem is solved on too many processors. Table 2 shows how many sweeps were required for convergence in method 1. It seems that not only the number

p	2	3	4	6	8	10
960×720	–	–	–	11.0	10.7	11.4
960×480	–	–	9.1	9.7	10.5	11.2
960×240	–	–	8.4	9.2	9.7	10.2
528×528	7.0	9.2	9.4	10.4	10.8	–
528×264	6.7	8.0	8.4	9.9	–	–
528×120	6.3	7.4	8.1	8.6	–	–

Table 2: *Sweeps Taken Until Convergence in Method 1*

of processors, but also the size of the problem play a role in determining the number of sweeps required. It also turned out that method 2 does not require significantly more sweeps until convergence. Letting

$$R \equiv \frac{sweep_2}{sweep_1}$$

be the ratio between the number of sweeps required in the two methods, we observe a mean of 1.04 and a standard deviation of 0.24 for R. The maximum value for R was 1.49. Furthermore, the ratio R decreases for fixed p. This conforms to our expectations, which predicted

$$R = O\left(\frac{\log n}{\log p} \right).$$

Since the subproblem solver for method 2 is cheaper method 2 requires fewer flops than method 1. The observed flop ratios are shown in Table 3.

p	2	3	4	6	8
960×720	–	–	–	1.55	1.57
960×480	–	–	2.8	3.0	3.2
960×240	–	–	11.3	12.6	13.2
528×528	1.57	1.53	1.68	1.75	1.76
528×264	2.2	2.7	2.8	3.2	–
528×120	10.1	12.4	14.2	15.2	–

Table 3: *Observed flop ratio* $\frac{f^1_{tot}}{f^2_{tot}}$

So the additional work we invested in diagonalizing every subproblem did not pay off. The measurements in Table 4 demonstrate that method 2 is very efficient. For

p	2	3	4	6	8
960×720	–	–	–	19.3	19.4
960×480	–	–	16.6	16.8	15.9
960×240	–	–	13.9	12.5	10.2
528×528	12.2	16.0	16.6	16.2	16.0
528×264	11.5	12.4	12.8	9.9	–
528×120	10.9	10.1	8.7	5.7	–

Table 4: *Sustained MFlop Rate per Processor in Method 2*

example we are able to achieve a sustained performance of 159 MFlops on eight processors. This is made possible by the reliance of our algorithm on matrix-matrix multiplication. Furthermore the execution rate of method 2 is higher than that of method 1 in most cases. Letting

$$MR \equiv \frac{M^2_{tot}}{M^1_{tot}}$$

be the ratio of the execution rates, we observe a mean of 1.49 and a standard deviation of 0.66 for MR. Maximum and minum values were 2.7 and 0.6, respectively. A closer look reveals that MR increases with p for a fixed-sized problem. Method 2 executes slower only in the 960×240 and 528×128 cases. This stems from a degradation in the performance of the block column updates due to shorter vector length. Table 1 shows that method 2 is nonetheless far superior.

We also realize that in most cases M_{tot} decreases as the number of processors increases, contrary to what the inspection of (5.1) - (5.3) had suggested. This is in spite of the fact that the transfer rate is always very high: it varies between 4 and 12 MBytes/sec per processor resulting in a very low communication overhead for the algorithm. A closer look at the timing data revealed that most of the communication time is spent in the broadcast and that here the effective transfer rate increases with the number of processors. Apparently the synchronisation overhead decreases; when there are more processors, they spend less time waiting for each other and so the transfer rate does not deteriorate.

To better understand why M_{tot} decreases, we take the 960×480 problem as representative example and look at the effect that the different parts of the algorithm have on the overall performance in Table 5. Here $pt_{qr}, pt_{sub}, pt_{block}, pt_{com}$ and pt_{rest} are the percentages of the total execution time spent on the pipelined QR factorization, the subproblem, the block column updates, communication and the rest of the code, respectively. Table 5 shows us the critical importance of the subproblem solver. First

p	M_{sub}	M_{block}	T_{com}	pt_{sub}	pt_{block}	pt_{com}
4	3.6	37.4	5.4	65.8	26.8	5.9
6	2.8	32.1	2.7	51.6	29.4	17.0
8	2.1	31.4	3.7	50.1	30.5	16.9
10	1.7	28.0	5.2	47.9	33.7	14.9

a) Method 1

p	M_{qr}	M_{sub}	M_{block}	T_{com}	pt_{qr}	pt_{sub}	pt_{block}	pt_{com}
4	23.2	5.8	33.7	10.8	7.4	50.5	35.6	4.8
6	20.1	5.1	31.2	11.1	8.5	38.1	42.2	8.3
8	19.2	4.6	27.5	10.1	8.3	29.6	46.9	11.7

b) Method 2

Table 5: *Execution Profile of the 960×480 Problem*

we notice that the Jacobi SVD subproblem solver executes much faster than our version of the Chan SVD. This stems from the fact that the Chan SVD algorithm is quite complicated. The pipelined architecture of the FPS-164 executes vector floating-point operations quickly, but control flow instructions and subroutine calls are very expensive in comparison. The simple Jacobi code, unlike the Chan SVD, does not suffer from these effects to the same extent. For the same reason, the performance of the Jacobi solver deteriorates much more gracefully as the subproblem size decreases. So the shift in the computational load from subproblem to block column updates is reflected in method 2, where the percentage of execution time spent on the subproblem decreases from 51% to 30%. On the other hand, the decline in execution rate mostly offsets the shift in computational load for method 1 and here pt_{sub} only drops from 66% to 48%, much less than what we had expected. This again shows that counting flops is an unreliable measure of performance in high performance environments. Also note that the block column updates and the QR algorithm executed very quickly due to their reliance on matrix-matrix multiplication.

Comparing our parallel algorithm with the one-processor case, we observed the performance shown in Table 6. It shows the total execution time for the LINPACK

problem size	*LINPACK SVD*	*Modified Chan SVD*
528×264	156	114
528×120	48	30

Table 6: *Execution Time in Seconds for the Uniprocessor Case*

SVD [11] using the assembler-coded BLAS from the APMATH64 library and for the subproblem solver of method 1 on a 528×264 and 528×120 problem. All other problems were too big to fit into the memory of one processor. A comparison with Table 1 shows that our reliance on matrix-matrix multiplication in the parallel code has paid off handsomely. Our second method is quite competitive with the one-processor case and certainly a good choice for problems that are too big to fit into the memory of one processor.

6. Conclusions. Our experiments have shown that the two-sided Jacobi method is indeed well suited for distributed systems with vector nodes. We achieve very high execution speed and communication overhead stays low. What limits the computational performance is not the communication overhead but the subproblem solver. We compared two versions of the block Jacobi method that mainly differed in the degree to which they diagonalize a given subproblem. The first method is a true block generalization of the scalar scheme in that each off-diagonal block is completely annihilated. The second method is a scalar Jacobi algorithm reorganized so that it conforms to the block decomposition of the problem. The comparison of the two methods shows that we want to spend as little work as possible in the subproblem solver. Even highly optimized, it will be the slowest part of the code. Unfortunately, investing more flops to make the subproblem more diagonal is not rewarded by an adequate reduction in the number of sweeps required for convergence. This suggests that the performance of the blocked version of a scalar Jacobi SVD will be difficult to surpass.

ACKNOWLEDGEMENTS

I wish to thank my advisor Charles Van Loan for introducing me to Jacobi methods and for his many stimulating suggestions that found their way into the algorithm. I am also grateful to Gautam Shroff from RPI for sharing his knowledge about Jacobi methods with me. I am deeply indebted to Dr. Enrico Clementi for giving me the opportunity to gain numerical experience with the algorithm on the LCAP system. I wish to thank all the members of his group and especially Doug Logan for their kind assistance.

REFERENCES

[1] C. Baudet and D. Stevenson, Optimal Sorting Algorithms for Parallel Computers, *IEEE Transactions on Computers* **C-27** (1978) 84-87.

[2] G. Bilardi and A. Nicolau, Bitonic Sorting with O(NlogN) comparisons, *Proceedings of the 20th Annual Conf. on Information Sciences and Systems*, Princeton, NJ (1986).

[3] M. Berry and A. Sameh, Multiprocessor Jacobi Algorithms for Dense Symmetric Eigenvalue and Singular Value Decompositions, *Proc. Int. Conf. on Parallel Processing* (1986) 433-440.

[4] C. H. Bischof and C. F. Van Loan, Computing the Singular Value Decomposition on a Ring of Array Processors, in: J. Cullum and R. A. Willoughby, eds., *Large Scale Eigenvalue Problems*, Mathematics Studies Series **127** (North-Holland, Amsterdam, 1986) 51-66.

[5] C. H. Bischof and C. F. Van Loan, The WY Representation for Products of Householder Matrices, *SIAM J. Scientific and Statistical Computing* **8** (1987) s2-s13.

[6] C. H. Bischof, A Parallel Ordering for the Block Jacobi Method on a Hypercube Architecture, in: M. T. Heath, ed., *Hypercube Multiprocessors 1987* (SIAM, Philadelphia, 1987) 612-618.

[7] R. P. Brent, F. T. Luk and C. F. Van Loan, Computation of the Singular Value Decomposition using Mesh-Connected Processors, *J. VLSI Computer Systems* 1 (1985) 242-270.

[8] T. Chan, An Improved Algorithm for Computing the Singular Value Decomposition, *ACM Trans. Math. Software* 8 (1982) 72-83.

[9] A. E. Charlesworth and J. L. Gustafson (1986), Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer, *IEEE Computer* **19** (1986) 10-23.

[10] E. Cementi et al., Large-Scale Computations on a Scalar, Vector and Parallel "Supercomputer", *Parallel Computing* **5** (1987) 13-44.

[11] J. Dongarra, J. R. Bunch, C. Moler and G. W. Stewart, *LINPACK User's Guide* (SIAM, Philadelphia, 1979).

[12] J. J. Dongarra and D. C. Sorensen, Linear Algebra on High-Performance Computers, in: U. Schendel, Ed. *High-Performance Computers 85*, (North-Holland, Amsterdam, 1986) 3-32.

[13] J. J. Dongarra and I. S. Duff, Advanced Computer Architectures, Argonne National Laboratories Report ANL-MCS-TM 57 (Revision 1), 1987.

[14] J. J. Dongarra and D. C. Sorensen, Block Reduction to Tridiagonal Form for the Symmetric Eigenvalue Problem, Manuscript, 1987.

[15] P. Eberlein , On Using the Jacobi Method on the Hypercube, in: M. T. Heath, Ed., *Hypercube Multiprocessors 1987* (SIAM, Philadelphia, 1987) 605-611.

[16] 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.

[17] G. H. Golub and C. F. Van Loan, *Matrix Computations* (Johns Hopkins University Press, Baltimore, 1983).

[18] M. Hestenes, Inversion of Matrices by Biorthogonalization and Related Results , *SIAM J. Applied Mathematics* **6** (1958) 51-90.

[19] F. T. Luk, Computing the Singular Value Decomposition on the ILLIAC IV, *ACM Trans. Math. Software* **6** (1980) 524-539.

[20] F. T. Luk, A Triangular Processor Array for Computing the Singular Value Decomposition, *Linear Algebra and Applications* **77** (1986) 259-274.

[21] F. T. Luk and H. Park, On Parallel Jacobi Orderings, Cornell University, School of Electrical Engineering Report EE-CEG-86-5, 1986.

[22] F. T. Luk and H. Park, A Proof of Convergence for two Parallel Jacobi Methods, Cornell University, School of Electrical Engineering Report EE-CEG-86-12, 1986.

[23] D. Scott, M. T. Heath, R. C. Ward, Parallel Block Jacobi Eigenvalue Algorithms using Systolic Arrays, University of Texas at Austin, Dept. of Computer Science Report TR 85-01, 1985.

[24] R. Schreiber, Solving Eigenvalue and Singular Value Problems on an Undersized Systolic Array, *SIAM J. Scientific and Statistical Computing* **7** (1986) 441-451.

[25] C. F. Van Loan, The Block Jacobi Method for Computing the Singular Value Decomposition, Cornell University, Dept. of Computer Science Report TR 85-680, 1985.

[26] J. H. Wilkinson, *The Algebraic Eigenvalue Problem* (Clarendon Press, Oxford, 1965).