PROCEEDINGS
SPIE—The International Society for Optical Engineering

Advanced Signal Processing: Algorithms, Architectures, and Implementations V

Franklin T. Luk
Chair/Editor

24–27 July 1994
San Diego, California

Sponsored and Published by
SPIE—The International Society for Optical Engineering

Volume 2296

SPIE (The Society of Photo-Optical Instrumentation Engineers) is a nonprofit society dedicated to the advancement of optical and optoelectronic applied science and technology.

PIPELINING MULTIPLE SVDs ON A SINGLE PROCESSOR ARRAY

Kishore Kota and Joseph R. Cavallaro
Department of Electrical and Computer Engineering
Rice University, Houston, TX 77251-1892
(713)-527-4719
Fax: (713) 524-5237
kota@rice.edu, cavallar@rice.edu

ABSTRACT

We present a new family of architectures for processor arrays to implement Jacobi SVD which allow systolic loading and unloading of input and result matrices. Unlike most of the previous SVD arrays in the literature, our architectures do not require special handling of external I/O and hence are closer to the traditional concept of systolic architectures. The boundary processors communicate with the host the same way any of the interior processors communicate with their neighbors. The arrays are surprisingly uniform and simple. The various architectures in the family represent different throughput-hardware tradeoffs corresponding to the degree to which the multiple sweeps have been unrolled and determine the number of independent SVDs which may be pipelined on the array.

We achieve systolic loading by using the flexibility provided by the cyclic Jacobi method on the order in which pivot pairs may be chosen. The array operates on the matrix data even as it is being loaded. Once the pipeline is full, the ordering is very similar to odd-even ordering. Our ordering is equivalent to cyclic-by-rows ordering and hence the algorithm is guaranteed to converge. Our systolic loading scheme is very important in an I/O limited system, since it allows more communication to occur in parallel, where the communication includes the loading and unloading operations. The array with the highest throughput in our family of architectures, which implement one-sided Jacobi (either Hestenes' method or Eberlein and Park's method), is a linear array of processors with unidirectional links between neighbors. The architectures with lower throughput require fewer processors connected in a ring, allowing data to recirculate among the processors. The input matrix is loaded one column at a time from the left and the results stream out one column at a time from the right.

Keywords: Jacobi method, SVD, pipelining, systolic arrays, systolic loading

1 INTRODUCTION

In this paper we propose a new ordering for the choice of pivots in the cyclic Jacobi method for the parallel computation of the singular value decomposition (SVD) or equivalently the Hermitian eigenvalue decomposition problem. This new ordering possesses some interesting properties which none of the orderings proposed previously in the literature possess. These properties make this ordering preferable over the others in some implementation

scenarios.

Numerous arrays have been proposed in the past decade for parallel computation of the SVD. The basis of most of these arrays is the cyclic Jacobi method coupled with the systolic paradigm. This results in a powerful algorithm where the entire computation can be performed in a uniform manner and hence is particularly amenable to parallelization either in custom arrays or general purpose parallel machines. In this paper we assume the reader to be familiar with the sequential cyclic Jacobi algorithm⁹ and any of the parallel variants.⁴ Since the literature on this problem is extensive we will refer only to those papers which are relevant to the problem at hand and refer the reader to the many papers on this problem for a tutorial introduction.

Some of the earliest algorithms developed for parallel SVD perform the computation *in-core*, for instance the Brent and Luk (BL) array.³ By "in-core" we mean that these algorithms assume the input matrix is already available distributed among the processors and leave the resulting matrices distributed. These algorithms are also the ones typically implemented on general purpose parallel machines, either the one-sided variant² or the two-sided variants.¹⁶ The cyclic Jacobi method allows a lot of flexibility in the order in which pivot pairs, *viz.* the next pair of off-diagonal elements to be annihilated, may be chosen. All the variants of the Jacobi method we find in the literature are a direct result of this flexibility. Some of the reasons to look for alternative orderings are to reduce communication, increase the number of computations that can be performed in parallel, and provide uniformity and ease of implementation of the orderings. Given the *in-core* assumptions, Gao and Thomas¹⁰ showed the lower bounds that can be achieved on the first two parameters, namely the number of parallel cycles of computation and the number of messages required for communication in *any* parallel version of the cyclic Jacobi method. They also proposed an ordering which achieves these lower bounds on a hypercube interconnection. Other orderings have been proposed to achieve these minimum bounds for the ring^{2,7} and torus⁵ topologies using bi-directional communication links between adjacent processors. When these algorithms are implemented as special-purpose VLSI processors, special cases in the orderings tend to add many additional states to the controllers, resulting in larger and slower controllers.¹³ From that point of view it appears that these new orderings would require more complex controllers than the BL ordering. A variant of the ordering we propose in this paper can achieve the lower bounds *and* result in simpler controllers. However, in this paper we are interested in a slightly different problem, namely the implications of relaxing the *in-core* assumption on the cyclic Jacobi algorithm. In particular, when a large number of matrices need to be continuously decomposed, the time to load and unload the matrices may become significant.

There do exist previous SVD algorithms which do not make the *in-core* assumption for the input matrices but leave the results *in-core*, e.g. the triangular array by Luk,¹⁴ the SVD updating array of Moonen, van Dooren and Vandewalle.¹⁷ These arrays relax the *in-core* assumption by performing an initial QR followed by the Jacobi SVD. There are other arrays such as the ones by Finn⁸ and Schimmel¹⁹ which do not make the *in-core* assumption and let the results flow out of the array systolically. Finn showed that his array exhibits empirically observed linear convergence. Schimmel's array implements the one-sided Jacobi method but does not exploit all the parallelism available. None of the aforementioned methods utilize the flexibility afforded by the cyclic Jacobi method in the choice of the ordering to relax the *in-core* assumption. In this paper we present a new ordering which allows the implementation of the cyclic Jacobi method without the *in-core* assumption. Hence matrices are loaded as computation proceeds and the results flow out of the array in a seamless manner resulting in an array closer to the traditional concept of a systolic array. We develop our result for a one-sided Jacobi method and show that this results in a family of architectures with varying throughput-hardware tradeoffs. In particular the various members of the family correspond to varying degrees of unrolling the sweeps. The resulting array is also surprisingly uniform and simple, requiring only unidirectional communication links. We also describe a few applications of our method.

2 BACKGROUND

Given an $n \times n$ matrix A , the SVD of A is a decomposition of the form

$$A = U\Sigma V^T, \quad (1)$$

where U and V are $n \times n$ orthogonal matrices and Σ is an $n \times n$ diagonal matrix of singular values. The columns of U and V are called the left and right singular vectors, respectively. Most of the algorithms to compute the SVD consist of computing orthogonal matrices U_i and V_i such that the sequence of matrices A_i converges to a diagonal matrix, where the matrices A_i are defined as

$$\begin{aligned} A_0 &= A \\ A_{i+1} &= U_i^T A_i V_i. \end{aligned} \quad (2)$$

The *Jacobi* method consists of choosing a pivot (p_i, q_i) at an iteration i such that $1 \leq p_i < q_i \leq n$ and computing *Givens rotations* U_i and V_i to annihilate the (p_i, q_i) and the (q_i, p_i) elements of A_i . The *cyclic Jacobi* method consists of performing several *sweeps* of $n(n-1)/2$ iterations, where the pivots within a sweep are chosen a priori based on an *ordering* to successively annihilate each off-diagonal pair of elements. The *parallel cyclic Jacobi* method involves choosing a cyclic ordering which allows the Givens rotations of several iterations to be computed in parallel. The algorithm described above will be referred to as the *two-sided Jacobi* method. The *one-sided Hestenes* method¹² consists of computing a single Givens rotation V_i at each iteration to force the sequence of matrices A_i to converge to a matrix with orthogonal columns, where A_i are defined as

$$\begin{aligned} A_0 &= A \\ A_{i+1} &= A_i V_i. \end{aligned} \quad (3)$$

The one-sided method may be regarded as an implicit two-sided method applied to the symmetric matrix $A^T A$. A different one-sided method proposed by Eberlein and Park⁷ computes two rotations as in the two-sided method. However, only local column updates are performed, postponing the update of other matrix elements to when needed at a later iteration. The computational complexity of this method is comparable to the Hestenes method. For our purposes it suffices to group this method along with the Hestenes method under the unified title of *one-sided Jacobi* methods.

Noting the structure of Givens rotations, several parallel orderings have been proposed which allow the rotations of $k = O(n)$ successive iterations to be computed independent of the intervening iterations. The one-sided Jacobi methods, in addition, allow these rotations to be applied independently to columns of A_i , to update A_i directly to A_{i+k} . Consequently parallel implementations of the one-sided Jacobi methods exploit $O(n)$ parallelism by performing approximately $n/2$ iterations in parallel, resulting in *linear* arrays to compute the SVD. In two-sided Jacobi methods, on the other hand, updates to 2×2 submatrices of A_i are independent of each other, allowing update of these submatrices directly to the corresponding 2×2 submatrices in A_{i+k} . The application of these plane rotations can be pipelined to result in effectively a constant time update, thus achieving another $O(n)$ parallelism. This is the essence of the square arrays for two-sided Jacobi methods. Hence, the computation now consists of several sweeps where each sweep can be executed in $O(n)$ time. It is conjectured that a predetermined number of sweeps $O(\log n)$ suffice in practice, resulting in an algorithm that computes the SVD in $O(n \log n)$ time on $O(n^2)$ processors or $O(n^2 \log n)$ time on $O(n)$ processors. Most of the previous algorithms thus result in a throughput of either $O(1/n \log n)$ or $O(1/n^2 \log n)$. Note that if higher throughput is required, it is always possible to achieve it by using multiple arrays in parallel and multiplexing the inputs to and outputs from these arrays. Though such a scheme increases the throughput by an arbitrary amount (at least in theory), the latency is still limited to $O(n \log n)$ for a square array and $O(n^2 \log n)$ for a linear array. Hence a scheme that just improves the throughput without any other benefits would not be interesting in itself.

Although a large number of parallel orderings have been proposed previously, there is unfortunately no common notation to represent all these parallel orderings equally well. This is because some notations tend to be more

Figure 1: A completely unrolled linear array has $nS/2$ processors, where n is the number of columns in the matrix and S is the number of sweeps, both of which are restricted to be even. The resulting array achieves the highest throughput for this family of architectures.

descriptive for some orderings than for others. Hence, we describe our ordering as a set of simple rules to be followed by each processor. Several examples using one of the other notations serve to illustrate some of the interesting features of our ordering.

3 PIPELINING ON LINEAR ARRAYS

In order to focus on the ordering aspects of linear arrays, we abstract the arrays previously described in the literature as follows. These arrays consist of $n/2$ processors, typically interconnected in a ring with bi-directional links. Each processor holds the data of two columns of the matrix. Computation proceeds in timesteps, which we call subsweeps. Each subsweep begins with processors exchanging data using some distributed algorithm that implements the desired ordering. Following this the processors perform some computation that updates the columns. We omit the details of this update since they have been described numerous times previously.^{3,7} We will assume the Hestenes method is being implemented, since replacing it with the other one-sided method by Eberlein and Park does not affect any of our conclusions. There are a number of possible scenarios for host to array I/O. Since all of them result in the same conclusions we assume the host to array I/O is through the leftmost processor and the array to host I/O is through the rightmost processor. The processors pass the data

Figure 2: Snapshots of a completely unrolled linear array in operation for $n = 6$. We show only two of the sweeps being performed since the figure repeats for the rest of the sweeps. We define a subsweep as the time to compute an angle, update the columns stored in the array and communicate with the neighboring processors. A new problem is loaded and a new solution is obtained every 6 subsweeps. A new column is loaded and unloaded every subsweep, once the pipeline is full. Exceptions to normal operation, due to the presence of an *on* token associated with the first column in the processor, is indicated in this figure by oval boxes for the processors. Presence of an *on* token is indicated in boldface. The numbers in the boxes indicate the numbers associated with the data in the columns when they were first introduced into the array. The figure shows that all the pairings required to complete a sweep indeed occur.

not intended for them to the processor on the right. The array operates synchronously in either the SVD mode, or the host I/O mode.

Our ordering is a variant of the odd-even ordering first introduced by Stewart.²² In fact, our ordering may be considered as the odd-even ordering coupled with additional data movement to allow a simple implementation on a distributed memory machine. Typically the odd-even ordering in previous work is described using the notion of a combined distributed-shared memory configuration.^{14,17,22} We prefer a fully distributed memory implementation.

We define our processor array interconnected as shown in Fig. 1. The array consists of $nS/2$ processors, where S is the number of sweeps which need to be performed for convergence and is constrained to be an even number. The processors are connected left to right using uni-directional links. The host to array I/O takes place through the left communication link of the leftmost processor and the array to host I/O takes place from the right communication link of the rightmost processor. On such an array it is possible to pipeline S independent

problems at any time.

We find it easier to describe the operation of the array with an example shown in Fig. 2. The example is for a matrix with 6 columns. The figure illustrates the operation of the array for only 2 sweeps, but certainly more sweeps would be required for convergence. Each of the boxes in the figure represents a processor, with the numbers in each box $[c_1 \ c_2]$ representing the two columns being processed in the processor. The number for any column represents the original column number when it was introduced in the array. The vertical axis represents time and shows the snapshots of the array after each timestep starting with timestep 0. At each timestep a new column is introduced into the array from the left end of the array. The columns are introduced in the order 6, 5, 4, 3, 2, 1 in this example. However, the columns could have been introduced in any order and this does not affect the rest of the discussion. Along with the data, a single bit of information, which we call a *token*, is associated with each column. The token could be either *on* or *off*, to differentiate between columns of different matrices. When we say a column has moved to some processor we mean that the data and the token associated with that column have moved. We initially associate an *on* token with column 6 and an *off* token with all other columns. The token is used as a control mechanism.

The functionality of the processors is very simple. Each timestep consists of an *external data exchange*, computation and an *internal data exchange* in that order. The external data exchange essentially consists of shifting the columns (both the data and the token) one position to the right. Hence external data exchange for a processor involves passing the second column c_2 to the processor on the right; shifting the first column c_1 to become the second column; and receiving the column from the processor on the left to form the new first column. The next two steps, i.e. the computation and the internal data exchange are conditional on the token associated with the first column in the processor after the external data exchange. If the token associated with the first column is *off*, a Givens rotation is computed to orthogonalize the two columns. Note that there are two possible rotations that orthogonalize the columns, namely the *inner* rotation and the *outer* rotation.⁹ It is well-known that the outer rotation may be regarded as an inner rotation followed by a permutation.²² Hence if the processor uses inner rotations to transform the two columns, this needs to be followed by an explicit exchange of the data in the two columns to complete the timestep. This internal data exchange exchanges only data between columns, not the tokens. If the outer rotation is used, then no explicit internal data exchange is required. On the other hand, if the token associated with the first column is *on*, then no computation or internal data exchange is required for that timestep. As can be verified from Fig. 2, this rather simple sequence of operations performed by every processor results in an ordering in which every pivot pair occurs and hence if the number of processors is large enough, a sufficient number of sweeps are performed to result in a matrix with orthogonal columns. In addition if the number of processors is some even multiple, S , of $n/2$, then exactly S sweeps would be performed on the input matrix A , and the columns streaming out of the rightmost processor would be columns of the resultant matrix $A_{nS/2}$ as defined in Eq. 3. At timestep 6 in this example, a new problem could be introduced into the array, thus resulting in pipelining multiple SVDs on the array. If for some reason a new problem is not introduced, it is still necessary to introduce an *on* token in timestep 6 to indicate the end of the current matrix. Thus if S sweeps are required for convergence, then the use of $nS/2$ processors results in an array in which S problems may be pipelined, resulting in a throughput corresponding to a new problem every n timesteps.

It is easy to verify that the resulting ordering is equivalent to the cyclic-by-rows ordering, where *equivalence* is the same notion used by Hansen¹¹ and later by Luk and Park,¹⁵ Shroff and Schreiber.²¹ For example, from Fig. 2 the sequence of rotations could be represented as follows:

$$\begin{aligned}
 V &= V_{56} V_{46} V_{36} V_{45} V_{26} V_{35} V_{16} V_{25} V_{34} V_{15} V_{24} V_{65} V_{14} V_{23} V_{64} V_{13} V_{54} V_{63} V_{12} \cdots \\
 &\equiv V_{56} V_{46} V_{36} V_{26} V_{45} V_{35} V_{16} V_{25} V_{34} V_{15} V_{24} V_{65} V_{14} V_{23} V_{64} V_{13} V_{54} V_{63} V_{12} \cdots \\
 &\equiv V_{56} V_{46} V_{36} V_{26} V_{16} V_{45} V_{35} V_{25} V_{34} V_{15} V_{24} V_{65} V_{14} V_{23} V_{64} V_{13} V_{54} V_{63} V_{12} \cdots \\
 &\equiv V_{56} V_{46} V_{36} V_{26} V_{16} V_{45} V_{35} V_{25} V_{15} V_{34} V_{24} V_{65} V_{14} V_{23} V_{64} V_{13} V_{54} V_{63} V_{12} \cdots \\
 &\equiv V_{56} V_{46} V_{36} V_{26} V_{16} V_{45} V_{35} V_{25} V_{15} V_{34} V_{24} V_{14} V_{65} V_{23} V_{64} V_{13} V_{54} V_{63} V_{12} \cdots \\
 &\equiv (V_{56} V_{46} V_{36} V_{26} V_{16} V_{45} V_{35} V_{25} V_{15} V_{34} V_{24} V_{14} V_{23} V_{13} V_{12})(V_{65} V_{64} V_{54} V_{63} \cdots
 \end{aligned}$$

The Jacobi method using cyclic-by-rows and cyclic-by-columns orderings has been proven to converge in a seminal

paper by Forsythe and Henrici⁹ and hence by equivalence, our algorithm is guaranteed to converge. We have included a sketch of a formal proof of the above statements in the appendix.

3.1 Throughput-Hardware tradeoffs

Figure 3: A linear array where all the sweeps have not been unrolled. The idea is to reuse portions of the unrolled array at the expense of the ability to load new problems. Depending on the size of the array it may be possible to compute several problems in a pipelined fashion.

The array described in the previous section is the completely unrolled array, where all the sweeps have been unrolled. This allows new problems to be introduced into the array, one every n subsweeps. Hence an S -fold improvement in throughput is obtained over an array with $n/2$ processors. However if the throughput requirements are lower, it is possible to lower the hardware cost. This is possible by reusing processors with a feedback connection that allows the matrix to recirculate in the array. In particular, smaller arrays with $nx/2$ processors are possible, where x is a design parameter, which can be chosen to be any factor of S , the number of sweeps required for convergence. In a linear array with $nx/2$ processors, it is possible to pipeline x problems at a time. New sets of x problems get introduced into the array once every nS subsweeps. Similarly, solutions to x problems are obtained every nS subsweeps. However, the price paid for this is additional control complexity. Figure 3 shows an array, where switches are necessary at the boundaries to allow either the feedback connection or the connection to the host. The input switch allows connection to the external host for the first nx subsweeps followed by the feedback connection for the remaining subsweeps. The output switch feeds back the data and the

token for the first $n(S - x)$ subsweeps and enables the connection to the external host for the last nx subsweeps. The special case of an array with $x = 1$ is always possible. We now have a family of processor arrays starting with $n/2$ processors all the way to $nS/2$ processors for higher throughputs. In the other direction it is possible to obtain lower throughput arrays than what is possible with $n/2$ processors by using Schreiber's ideas,²⁰ where each processor simulates several processors in the unrolled array.

3.2 Comparison with in-core methods

So what did we gain from this new ordering? At first glance it appears to take more subsweeps (i.e. parallel timesteps) for each SVD and hence more time to compute each SVD than other methods previously proposed. However, there is an implicit assumption made in such a comparison: it ignores the time taken to load the array from and unload the array to the host processor. Thus if a large number of SVDs are to be performed, this time needs to be accounted.

In order to compare our array with previous arrays, let us assume that the time required to transfer one column from one processor to the next is $n\alpha$, and the time to compute an outer angle and apply it to two columns is $n\beta$. In a linear array there is no need to scale the communication ability of the host, quantified by α , with the problem size, unless the computational speed β is also scaled. This is because the time to load the array would be $O(n^2)$, while computation of the SVD takes approximately $O(n^2 \log n)$. The same is not true of the square arrays. In square arrays the communication capacity of the host has to scale with problem size, even when the processors' computation speed is kept a constant.

Let us first compute the time required for an SVD using a typical algorithm that makes *in-core* assumptions. Assuming an optimal ordering as defined by Gao and Thomas,¹⁰ the number of subsweeps required for each sweep would be $n(n-1)/2 \times 1/\lfloor n/2 \rfloor$. Each subsweep involves $n\beta$ time for computation and $n\alpha$ for communication. In addition we need $n^2\alpha$ to load the array and $n^2\alpha$ to unload the array. However, if we assume the existence of a forward link and a reverse link from the host to the array, then the unloading of one problem can be overlapped with the loading of another problem. Hence the amortized time for each SVD is:

$$\begin{aligned} T_{in-core} &= n^2\alpha + \frac{n(n-1)}{2} \times \frac{1}{\lfloor n/2 \rfloor} n(\alpha + \beta) \times S \\ &= \begin{cases} n^2\alpha + (n^2\alpha + n^2\beta) \times S & \text{if } n \text{ odd} \\ n^2\alpha + (n-1)(n\alpha + n\beta) \times S & \text{if } n \text{ even.} \end{cases} \end{aligned} \quad (4)$$

Using the same numbers let us now compute the time required for the SVD using the array we propose in this paper. To perform S sweeps it takes $n(S+1)$ subsweeps. However, the last n subsweeps overlap with those for the next SVD. Since each subsweep takes $n\alpha + n\beta$ time, the amortized time for an SVD is:

$$T_{pipe-svd} = (n^2\alpha + n^2\beta) \times S. \quad (5)$$

In a compute limited system, if n is even, the *in-core* assumption could result in a faster implementation. However, the chief benefit of the new array is when the system is I/O limited. In this case, since S is $O(\log n)$ and in practice always less than n , the proposed array actually computes the SVD faster. In addition, it does not require the links between processors to be bi-directional. The processor descriptions are all the same and hence the proposed array should result in cheaper hardware. Quantitatively, the condition under which the proposed array would be faster than one that makes *in-core* assumptions is

$$\frac{\alpha}{\beta} > \begin{cases} \frac{S}{n-S} & \text{if } n \text{ even} \\ 0 & \text{if } n \text{ odd.} \end{cases} \quad (6)$$

Since $S/(n-S)$ tends to zero as $n \rightarrow \infty$, for any given set of technology constraints parameterized as α/β , there exists n_0 such that for any $n > n_0$, the pipelined array will be faster.

4 APPLICATION

4.1 Subspace Tracking

In subspace tracking problems it is necessary to track the subspace spanned by the incoming observation vectors. When the parameters to be estimated are time-varying it is necessary to apply a window on the data. The most common windows are the rectangular block window, sliding exponential window and a rectangular sliding window in increasing order of computational complexity. The linear array we have proposed is useful in applications with low to moderate requirements. The application we are interested in is Code Division Multiple Access (CDMA), a multiple access strategy to be used in mobile digital cellular communications. The acquisition problem has recently been formulated in the same framework as direction finding problems.¹ The parameters to be estimated in the CDMA acquisition problem are the relative time delays of all the users. These time-delays are later used in the detectors to decode the transmitted data. The parameters are likely to change with time since the users are mobile. Our initial estimates show that the problem requires the SVD of at least a 64×256 matrix, with a throughput of one new 64-element vector approximately every $50\mu s$ with a latency of $10ms$. In this application a linear array is a feasible solution in that it is sufficient to support the required latency and throughput requirements and is cost effective compared to a square array. Though other implementations for the linear array are possible, it appears that the pipelined array matches the problem very well in that the I/O requirements of the array match the data acquisition in real time. In addition, the simplicity of the array allows a custom VLSI implementation using CORDIC techniques as one of the implementation alternatives.

5 CONCLUSIONS

The linear pipelined array we have proposed in this paper is developed at a sufficiently high level that other ideas for low level pipelined implementations can be easily incorporated.^{6,18} The pipelining idea results in a very simple array which should considerably ease practical implementations. Theoretically, we found that for large problems the pipelined array never performs worse than arrays that make *in-core* assumptions. The host I/O requirements are sufficiently simple that they match the data acquisition in subspace tracking problems. The array should result in a cost-effective solution in subspace tracking applications which do not have very stringent latency requirements.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grant DMI-9202639, and by DOE Sandia National Laboratory Contracts #18-4379A and DE-AC04-94AL850.

6 REFERENCES

- [1] S. E. BENSLEY, *Channel Estimation for Code Division Multiple Access Communication Systems*, Master's thesis, Rice Univ., Houston, TX, May 1994.
- [2] C. H. BISCHOF, *Computing the Singular Value Decomposition on a Distributed System of Vector Processors*, *Parallel Computing*, 11 (1989), pp. 171-186.
- [3] R. P. BRENT AND F. T. LUK, *The Solution of Singular-Value and Symmetric Eigenvalue Problems on*

- Multiprocessor Arrays*, SIAM Journal of Scientific and Statistical Computing, 6 (January 1985), pp. 69–84.
- [4] R. P. BRENT, F. T. LUK, AND C. F. VAN LOAN, *Computation of the Singular Value Decomposition Using Mesh-Connected Processors*, Journal of VLSI and Computer Systems, 1 (1985), pp. 242–270.
 - [5] J.-M. DELOSME, *Parallel Implementations of the SVD using implicit CORDIC arithmetic*, in SVD and signal processing, II, Algorithms, Analysis and Applications, R. J. Vacarro, ed., Elsevier Science Publishers B. V., 1991, pp. 33–56. 655 Avenue of the Americas, New York, N. Y. 10010.
 - [6] E. F. DEPRETTERE, P. DEWILDE, AND R. UDO, *Pipelined CORDIC Architectures for Fast VLSI Filtering and Array Processing*, IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, 3 (March 1984), pp. 41A.6.1–41A.6.4.
 - [7] P. J. EBERLEIN AND H. PARK, *Efficient Implementation of Jacobi Algorithms and Jacobi Sets on Distributed Memory Architectures*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 358–366.
 - [8] A. M. FINN, *Systolic Array Computation of the Singular Value Decomposition*, PhD thesis, School of Electrical Engineering, Cornell Univ., Ithaca, NY, May 1983.
 - [9] G. E. FORSYTHE AND P. HENRICI, *The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix*, Transactions of the American Mathematical Society, 94 (January 1960), pp. 1–23.
 - [10] G. R. GAO AND S. J. THOMAS, *An Optimal Parallel Jacobi-Like Solution Method for the Singular Value Decomposition*, IEEE Int'l Conference on Parallel Processing, (August 1988), pp. 47–53.
 - [11] E. R. HANSEN, *On Cyclic Jacobi Methods*, J. Soc. Indust. Appl. Math., 11 (1963), pp. 448–459.
 - [12] M. R. HESTENES, *Inversion of matrices by biorthogonalization and related results*, J. Soc. Indust. Appl. Math., 6 (1958), pp. 51–90.
 - [13] K. KOTA AND J. R. CAVALLARO, *CMOS processor element for a fault-tolerant SVD array*, in Advanced Signal Processing Algorithms, Architectures, and Implementations IV, F. T. Luk, ed., vol. Proc. SPIE 2027, 1993, pp. 483–494.
 - [14] F. T. LUK, *A Triangular Processor Array for Computing Singular Values*, Journal of Linear Algebra and Its Applications, 77 (1986), pp. 259–273.
 - [15] F. T. LUK AND H. PARK, *A Proof of Convergence for Two Parallel Jacobi SVD Algorithms*, IEEE Trans. on Computers, 38 (June 1989), pp. 806–811.
 - [16] J. J. MODI AND J. D. PRYCE, *Efficient Implementation of Jacobi's Diagonalization Method on the DAP*, Numerische Mathematik, 46 (1985), pp. 443–454.
 - [17] M. MOONEN, P. VAN DOOREN, AND J. VANDEWALLE, *A Singular Value Decomposition Updating Algorithm for Subspace Tracking*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 1015–1038.
 - [18] J. H. MORENO AND T. LANG, *Multilevel Pipelined Processor for the Singular Value Decomposition*, in Real Time Signal Processing IX, W. J. Miceli, ed., vol. Proc. SPIE 698, 1987, pp. 100–112.
 - [19] D. E. SCHIMMEL AND F. T. LUK, *A New Systolic Array for the Singular Value Decomposition*, in Advanced Research in VLSI, C. E. Leiserson, ed., 1986, pp. 205–217.
 - [20] R. SCHREIBER, *Solving Eigenvalue and Singular Value Problems on an Undersized Systolic Array*, SIAM Journal Sci. Stat. Comput., 7 (April 1986), pp. 441–451.
 - [21] G. SHROFF AND R. SCHREIBER, *On the Convergence of the Cyclic Jacobi method for parallel block orderings*, SIAM J. matrix anal. appl., 10 (1989), p. 326.
 - [22] G. W. STEWART, *A Jacobi-Like Algorithm for Computing the Schur Decomposition of a Nonhermitian Matrix*, SIAM Journal of Scientific and Statistical Computing, 6 (October 1985), pp. 853–864.

APPENDIX

In this appendix, we sketch some of the important steps in formally proving the properties of the ordering we introduced in this paper. Before we start, here are a few definitions. The *position*, $\mathcal{P}(c)$, of a column c denotes its position in the array starting from the leftmost processor. The position is an integer such that $0 \leq \mathcal{P}(c) < nS$. Thus column 1 in the leftmost processor would be numbered 0, while column 2 would be numbered 1, column 1 in the processor to the right of the leftmost processor is 2 and so on. As described earlier, the subsweeps are numbered starting from 0 and increasing in time.

For a column c , we define the function $\Psi(c)$ to be the number of the subsweep at which it was introduced into the array. The function $\Omega(c)$ is the number of the latest subsweep, including $\Psi(c)$, at which an *on* token was introduced into the array. For any column c , we also define the notion of a *relative subsweep*. A subsweep t is defined to be the relative subsweep $t - \Omega(c)$ for the column c .

The following facts may be easily verified:

PROPOSITION 1. *A column associated with an off token when present in an even position in the array will always move two positions to the right in one subsweep. Hence that column will continue to remain in an even position as long as the condition required for this proposition is satisfied.*

PROPOSITION 2. *A column c in an odd position in the array at some subsweep will remain in the same position for the next subsweep if the token associated with the column in position $\mathcal{P}(c) - 1$ is off. Thus the column remains stuck in that position as long as the condition required for this proposition is satisfied.*

PROPOSITION 3. *An on token always moves right by one position in one subsweep under any conditions.*

The following theorem gives an exact relation for the position of a column c at any subsweep.

PROPOSITION 4. *The position of a column c at some relative subsweep k is given by*

$$\mathcal{P}_k(c) = \begin{cases} 2(k-p) - 2jn & \text{if } \max(2p + (2j-1)n, p-1) < k \leq 2p + 2jn \\ 2p + 1 + 2jn & \text{if } 2p + 2jn < k \leq 2p + (2j+1)n, \end{cases} \quad (7)$$

where $p \equiv \Psi(c) - \Omega(c)$ is the relative subsweep at which the column c was introduced into the array. The quantity j is an integer that satisfies either the constraints

$$\max(2p + (2j-1)n, p-1) < k \leq 2p + 2jn$$

or the constraints

$$2p + 2jn < k \leq 2p + (2j+1)n.$$

Proof. This proposition may be verified by induction using the Propositions 1,2,3. $\square$

From Proposition 4, the following corollaries can be derived.

COROLLARY 1. *Every pair of columns a and b which are introduced into the array such that $\Omega(a) = \Omega(b)$ will get paired at every subsweep t that satisfies*

$$t - [\Psi(a) + \Psi(b) - \Omega(a)] \equiv 0 \text{ mod } n.$$

To show equivalence of this ordering with the cyclic-by-rows ordering, we restate a result shown by Shroff and Schreiber²¹ about cyclic wavefront orderings. In the same paper they also show that cyclic wavefront orderings are equivalent to cyclic by rows ordering.

PROPOSITION 5. In a cyclic ordering O of the pairs $\{(g, h), 1 \leq g < h < n\}$, let $I(g, h)$ be the index at which pair (g, h) occurs. If

$$I(g, h-1) < I(g, h) < I(g+1, h)$$

for all $1 \leq g < h \leq n$ then O is called a cyclic wavefront ordering.

PROPOSITION 6. The ordering described by Eq. 7 is equivalent to a cyclic wavefront ordering, and is hence equivalent to the cyclic-by-rows ordering.

Proof. Choose two arbitrary columns a, b such that $\Omega(a) = \Omega(b)$ and $\Omega(a) \leq \Psi(a) < \Psi(b) \leq \Omega(a) + n$. Let b_{-1} be the column introduced into the array immediately before b and a_{+1} be the column introduced into the array immediately following a . Then Proposition 5, can be verified using corollary 1, i.e. it can be verified that the pairs (a, b_{-1}) , (a, b) and (a_{+1}, b) occur in that order in a given sweep. It can also be verified that (a, b_{-1}) from the next sweep will not occur until (a, b) from the current sweep is completed. Similar precedence constraints are satisfied by other pairs. These two conditions are sufficient to guarantee equivalence of the ordering to cyclic wavefront orderings. $\square$