Parallel Implementation of Eigenvalue Algorithms on Distributed Memory Machines

Zhou B. B. and Brent R. P.
Computer Sciences Laboratory
Research School of Physical Science and Engineering
The Australian National University
GPO Box 4, ACT 2601
Canberra, Australia

November 10, 1992

Abstract

This paper considers the parallel solution of large eigenvalue problems on a mesh-connected processor array with distributed memories. New parallel Jacobi algorithms are introduced for solving the problem. The algorithms require a small number of data communications between processing elements on our computing model. They have been implemented on the Fujitsu AP 1000. The paper also reports our analytical and experimental results.

1 Introduction

The eigenvalue decomposition of a symmetric $N \times N$ matrix A has the form

$$A = U \Lambda U^T \tag{1}$$

where U is orthogonal and Λ is diagonal. It has many important applications in modern signal processing. Because this problem is very computationally intensive, parallel computation has been considered in recent years.

The Jacobi method is an iterative algorithm for diagonalizing a symmetric matrix A. At each iteration a plane rotation is chosen to annihilate a symmetric pair of off-diagonal elements a_{ij} and a_{ji}:

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} = \begin{pmatrix} a'_{ii} & 0 \\ 0 & a'_{jj} \end{pmatrix} \tag{2}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. If the off-diagonal elements are annihilated in a reasonable, systematic order, the convergence rate to a diagonal matrix is ultimately quadratic [3].

Since the Jacobi method can easily be implemented in parallel, a lot of attention has been paid to using this technique for parallel computation of eigenvalue problems. In this paper we describe new parallel Jacobi algorithms for solving symmetric eigenvalue problems and particularly focus on data partitioning schemes since an algorithm without partitioning is hardly usable in practice to solve very large problems on a general-purpose parallel computer.

In Section 2 we briefly describe our computing model. Section 3 discusses an algorithm for parallel Jacobi ordering without partitioning. This algorithm is well suited for our computing model because it has lower communication requirements. We introduce two partitioning schemes in Section 4. The first (algorithm 1) is apparently new and the second (algorithm 2) is derived from the Schreiber's partitioning method [6]. Both have been implemented on the Fujitsu AP 1000. Our analytical and experimental results are reported in Section 5.

2 Computing Model

The system on which our experiments are performed is the Fujitsu AP 1000. This highly parallel computer is a distributed memory MIMD machine with up to 1024 independent 25 MHz SPARC processors. Each processor has 16 MByte of dynamic RAM and 128 KByte cache. The topology of the AP 1000 is a torus, with hardware support for wormhole routing. There are three communication networks — the B-net, for communication with the host; the S-net, for synchronization; and the T-net, for communication between processors. The T-net is the most significant for us. In practice it provides a bandwidth of about 6 MByte/sec

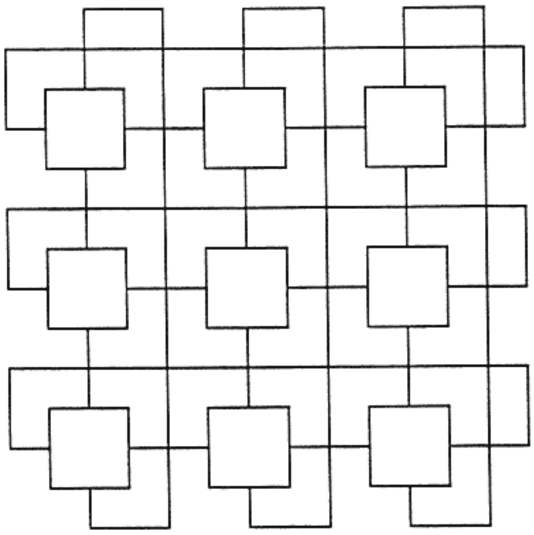


Fig. 1: The computing model.

between processors. (Theoretically it can achieve 25MByte/sec.) For details of the AP 1000 architecture and software environment, see [1][2].

It should be mentioned that the parallel algorithms we describe in this paper are not restricted to one particular machine. They can be implemented efficiently on any system in which the processing elements (PEs) are organised in a two-dimensional (2–D) mesh and connected by a 2–D torus network, as shown in Fig. 1. One–dimensional (1–D) arrays are considered as a special case of 2–D arrays with the number of rows (or columns) equal to one. Different algorithms might be peferable on MIMD machines with different topologies, *e.g.* hypercubes or trees [4].

3 Parallel Algorithm without Partitioning

One Jacobi iteration described in Equation 2 may be denoted by using an index (or a Jacobi) pair (i, j). For example, the Jacobi pair (1, 2) represents the Jacobi plane rotation operations for annihilating the off–diagonal elements $a_{1,2}$ and $a_{2,1}$. For an $N \times N$ problem, there are $N(N-1)/2$ distinct pairs. In serial computing these pairs are ordered and then the Jacobi plane rotation operations associated with them are executed sequentially. We refer to the sequence of these Jacobi pairs as a "sweep". One such sequence is the cyclic–by–row ordering which is depicted below for $N = 4$:

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

stage 1:	(1, 2)(3, 4)(5, 6)	(1, 2)(3, 4) 5
stage 2:	2 (1, 4)(3, 6) 5	2 (1, 4)(3, 5)
stage 3:	(2, 4)(1, 6)(3, 5)	(2, 4)(1, 5) 3
stage 4:	4 (2, 6)(1, 5) 3	4 (2, 5)(1, 3)
stage 5:	(4, 6)(2, 5)(1, 3)	(4, 5)(2, 3) 1
stage 6:	6 (4, 5)(2, 3) 1	**(b) N odd**
	(a) N even	

Fig. 2: The odd-even ordering.

The Jacobi plane rotation operations associated with one Jacobi pair only involve two rows and two columns of data items in the matrix. Therefore, there are disjoint pairs, *e.g.* $(1, 4)$ and $(2, 3)$ in the above ordering, which can be executed simultanously. In a parallel implementation, we want to perform as many non–interacting operations as possible at each time "stage". One example is the odd–even ordering shown in Fig. 2.

In the following, we describe an algorithm for Jacobi ordering without partitioning. We show that the communication structure of this algorithm exactly matches the computing model described in the previous section. It also has lower communication requirements than other well–known algorithms for the same problem.

The algorithm is described through examples. See Fig. 3(a) for N even and Fig. 3(b) for N odd. In the figures, the horizontal arrows denote the index movement, while the up-and-down arrows indicate that two indices within the column have to be swapped before one index is moved to the other column. If we ignore all the columns with an up-and-down arrow inside, after N stages every index will meet all other $N-1$ indices exactly once. This will make a total number of $(N-1)N/2$ different Jacobi pairs, which completes one sweep of Jacobi ordering for N indices.

The index movement in Fig. 3 uses wraparound. This ring structure just matches our computing model. The mapping of the algorithm onto the computing model is briefly described as follows. For an $N \times N$ problem, we use $\frac{N}{2} \times \frac{N}{2}$ processing elements. Before the computation starts, the original matrix is stored in the system in a natural order and each processing element holds a 2×2 submatrix. In each computational stage, each diagonal processing element first generates a plane rotation to annihilate the off–diagonal elements of its submatrix and then the rotation is sent to all other processing elements in the same row and the same column. The off–diagonal processing elements, on receiving rotations, then perform the submatrix updating. After the rotations have been applied by the off–diagonal processing elements, which completes one stage of computation, columns and rows are interchanged for the next stage.

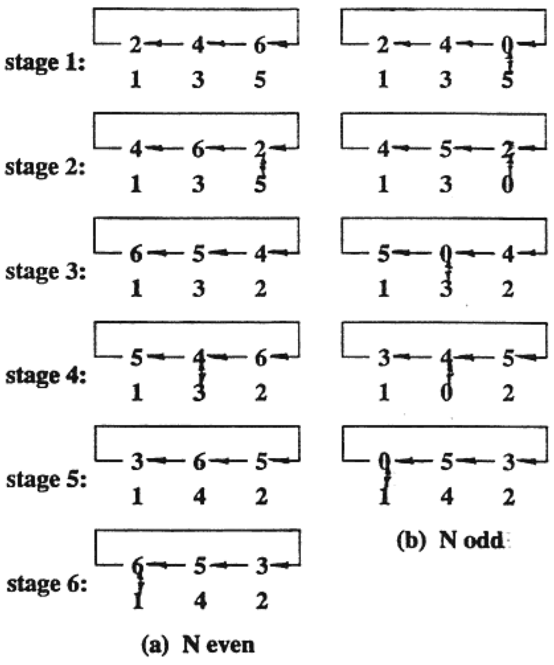


Fig. 3: The Jacobi ordering with low communication requirements.

When one index is transferred from one Jacobi pair to another between each two successive stages, not only is one row of data items to be transferred vertically, but also one column transferred horizontally from the associated processing elements. Thus, the more the transfer of indices, the higher the cost for communications. We can see from Fig. 3 that only $N/2$ indices change their positions at each stage. In contrast to other well-known Jacobi ordering methods, for example, those described in [5], this smaller number will result in a less number of data items to be interchanged between processing elements during parallel computation.

4 Partitioning Schemes

An algorithm without partitioning is hardly useful for general-purpose parallel computations in practice because the system configration is fixed, but the size of user's problem may vary. This section will describe two partitioning schemes based on the algorithm described in the previous section.

For simplicity we assume in the following discussion that an $N \times N$ problem is to be solved using a $p \times p$ square array for $N = 2qp$, where q is an integer. For the data loading, the original matrix is still stored in the system in a natural order, but each processing element now holds a $2q \times 2q$ submatrix.

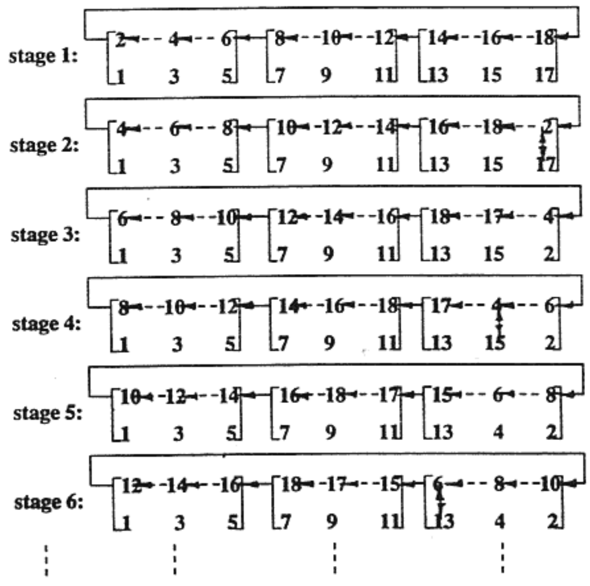


Fig. 4: The first attempt at partitioning.

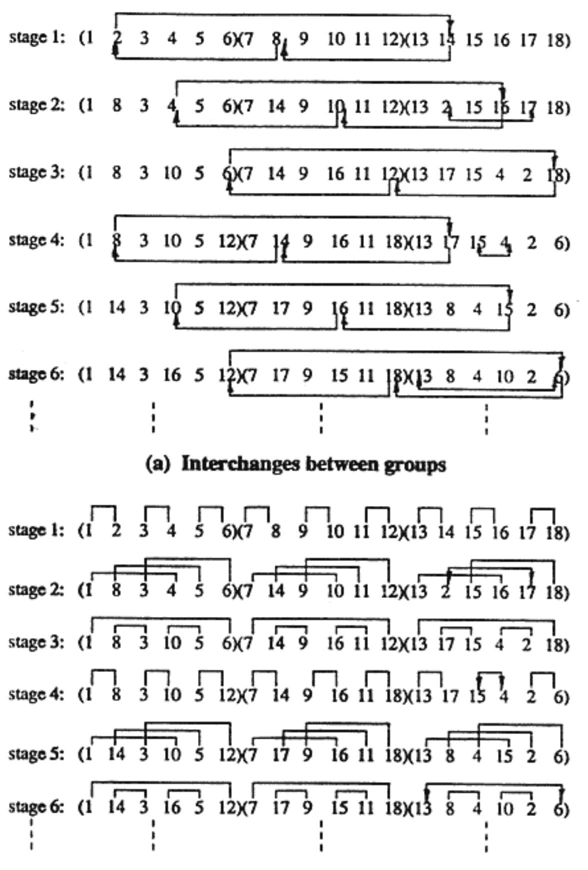
4.1 Algorithm 1

Our first partitioning algorithm is derived directly from Fig. 3. Divide N indices into p groups with each group holding $2q$ indices. An example of $N = 18$ and $p = 3$ is depicted in Fig. 4. In the figure solid arrow lines indicate the indices interchange between groups, while the dashed arrow lines denote the local permutation of indices within a group. Although only one out of $2q$ indices needs to be interchanged from each group at one stage, there is a large number of indices taking part in the local permutation. This will result in large number of data items to be permuted within the local memory of each processing element, which can seriously degrade the overall performance. To avoid this problem we apply a new strategy.

In order to simplify the discussion, we enumerate indices in each group from 0 to $2q-1$ and the stages, which are equal to the total number of indices, from 1 to N. Our new algorithm for indices interchange is described as follows:

```
count= 1;
for ($k = 1$; $k <= N$; $k$++){
    send a index in position equal to
     count to its left group;
    receive one index from the right group and
     place it in position equal to count;
    count+= 2;
    count%= 2 * q;
    }
```

An example of $N = 18$ and $q = p = 3$ is depicted in Fig. 5(a). It should be mentioned that the index exchanges denoted by the up-and-down arrows in



(a) Interchanges between groups

(b) Jacobi pairs within each groups

Fig. 5: Partitioning algorithm 1.

Fig. 4 need to be maintained in order to make the algorithm work correctly. Therefore, we are required to use an extra pointer to indicate these exchanges. The pointer first points to the position 1, and then moves two positions up to the right each time after one exchange until it reaches the right boundary. Each time the index in the position pointed to by the pointer will be exchanged. The counterpart of it is the one in the position $(2q-1)$ minus the value of the pointer. For example, if the value of the pointer is 1 and $q = 3$, then $2q - 1 - 1 = 4$. Thus the indices in positions 1 and 4 will be exchanged. The equivalence can be seen by comparing the up-and-down arrows in Fig. 4 and the left-and-right arrows in Fig. 5(a).

If we are only concerned that every group will still obtain the same indices at each stage after the modification described above, but do not care where the indices are actually placed in the group, we can easily prove that the above algorithm is equivalent to the one depicted in Fig. 4.

At stage 1 the indices in position 1 of each group are interchanged. This is equivalent to stage 1 in Fig. 4. Since there are no index exchanges within each individual group, the indices in position 3 of each group will be interchanged at stage 2 in order to obtain the equivalence. For the same reason the indices in position $2i + 1$ will be interchanged at stage i until $count = 2q + 1$. When $count = 2q + 1$, one group needs to obtain an index which is originally placed in position 1 of its second right group. However, this index has been moved to position 1 of its first right group at stage 1. Since the indices movement forms a ring, we have proven the equivalence.

In Fig. 4 the indices in each column form a Jacobi pair. This order is scrambled by the new interchange scheme. A question is whether these Jacobi pairs can be reorganized in a simple way. The answer to this is yes. The following describes how easily the Jacobi pairs may be obtained. In order to form correct Jacobi pairs, each index in the odd position, say i, will only be required to combine with its counterpart in position $i+count$ where the value of $count$ is defined previously. Since the value of $count$ is odd, $i+count$ must be even. The proof of the correctness of this process is similar to the one for our new indices interchange scheme and is omitted. An example is depicted in Fig. 5(b).

In the above algorithm it requires N stages to complete one sweep of Jacobi ordering of order N. There is only one index to be interchanged at each stage. The total number of indices transferred in one sweep is then N for each group.

4.2 Algorithm 2

Our second algorithm is based on Schreiber's partitioning method [6], which is briefly discussed below.

Divide N indices into $2p$ blocks, each holding q indices. Considering each block as a super–index and using the ordering described in Section 3, each super–index will meet every other super–indices in some columns exactly once after $2p$ super–stages, as shown in Fig. 6. Once two blocks are in the same column, some operations are performed to obtain Jacobi pairs by using indices in the blocks. If an index in one block is combined with every index in the other block once only, each index in one block will meet all the indices in other $2p - 1$ blocks exactly once after $2p$ super–stages. However, one sweep of Jacobi ordering of order N has not been completed unless the indices within the same block meet each other once. Since this extra process is performed locally within each individual block, it is combined into the first super–stage in [6].

To summarize, N indices are first divided into $2p$ blocks with equal number of indices. The ordering described in Section 3 is then applied. In each super–stage, indices in one block will meet every index in the other block within the same column (ex-

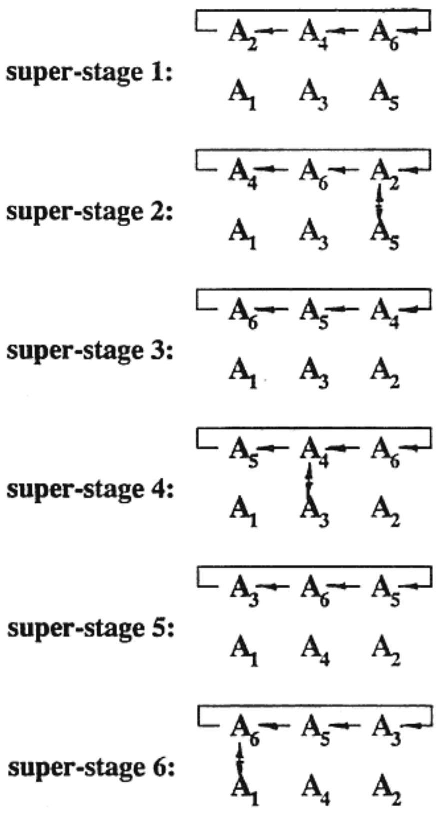


Fig. 6: Partitioning algorithm 2.

cept the column containing up-and-down arrows) to form Jacobi pairs. There are additional operations performed in the first super–stage, that is, each index has to meet once every other index within the same block. The algorithm takes $2p$ super–stages to complete a sweep of Jacobi ordering of order N. Since there are q indices in one column to be interchanged at each super–stage, the total number of indices transferred in one sweep from each column or group is also N.

5 Analytical and Experimental Results

There are two key issues which affect the efficiency of a parallel algorithm, that is, the cost of communication and the active/idle ratio of processing elements in the system. It is desirable if the algorithm designed for a given problem can not only minimize the communication cost, but also keep every processing elements active all the time during the computation.

Before we estimate the performance, we emphasize the following factors: In both algorithms one index corresponds to one row and one column. Therefore, there are two rows and two columns of data items in the matrix involved in one Jacobi plane rotation operation (called one Jacobi pair) and the amount of floating–point arithmetic operations executed in each associated processing element is $O(q) = O(N/p)$ since each processing element holds a $2q \times 2q$ submatrix. When a index is transferred from one group to another, not only is one row of data items transferred vertically, but also one column is transferred horizontally from the associated processing elements. To simplify the analysis, we ignore any overlap of communication and computation.

Communication costs:

We first estimate the communication costs. We shall assume that the time required to send or receive a message of w words is $c_0 + c_1 w$, where c_0 is a "startup" time and $1/c_1$ is the transfer rate [1].

In Algorithm 1 there is only one index transferred from one group to another at each stage. There are $2q$ data items in one row or one column of the submatrix in each processing element. The time required for one interchange is then $2(c_0 + 2c_1 q) = 2(c_0 + c_1 N/p)$. However, there are N interchanges in one sweep. Therefore, the time spent for communication in one sweep is $2N(c_0 + c_1 N/p) = 2Nc_0 + 2c_1 N^2/p$.

Since there are q indices to be transferred from each group in Fig. 6, the communication time in one super–stage of Algorithm 2 is $2(c_0 + c_1(2q^2))$. There are only $2p$ super–stages in one sweep. Thus the communication time spent in one sweep of Algorithm 2 is $2p(2(c_0 + c_1(2q^2))) = 4pc_0 + 2c_1 N^2/p$.

By comparing the above two results, it can be seen that the number of words or data items transferred during one sweep of operation is the same, that is, $2N^2/p$ for both algorithms. Since Algorithm 1 requires more times for data interchange, however, there is a difference in total "startup" time which is $2c_0(N - 2p)$.

Active/idle ratio:

We now compare the time for floating–point arithmetic operations in the two algorithms.

In order to simplify the comparison of the two algorithms, we introduce the term "equivalent stage". The time consumed by one equivalent stage is the time for executing the Jacobi plane rotations denoted by q Jacobi pairs.

We first consider Algorithm 1. It is easy to see from Fig. 5(b) that each group will generate exactly q Jacobi pairs (excluding those with an up–and–down arrow). Therefore, for Algorithm 1 the total number of equivalent stages in one sweep is N.

Now consider Algorithm 2. Since an index in one block only combines once with all the indices in the other block of the same column and one block contains q indices, in each super–stage but the first one there are q^2 Jacobi pairs formed in each column (excluding those with an up–and–down arrow), which is q equivalent stages. For the extra operations in the first super–stage there are an additional $q(q-1)$ Jacobi pairs. Since there are $2p$ super–stages in one sweep, the total number of equivalent stages is $2pq + q - 1 = N + q - 1$.

From the above discussion we see that Algorithm 2 requires $q - 1$ more equivalent stages in a sweep than Algorithm 1. This results in some additional time for performing extra $O(N^3/p^3)$ floating–point arithmetic operations to complete a sweep. The reason is explained as follows.

It is easy to see from Fig. 3 that an index pair with an up-and-down arrow will indicate only the exchange of the two indices, but not generate any Jacobi pair. This causes the associated processing elements to be idle. In Algorithm 1 the total number of these index pairs is $N/2$ which is independent of p or q. Although there are only p super–index pairs with an up-and-down arrow in Fig. 6, one such pair is equivalent to q^2 index pairs. This is because each column in Fig. 6 produces q^2 Jacobi pairs. Thus the total is equivalent to $pq^2 = qN/2$ which is q times the number in Algorithm 1. The active/idle ratio of the system when using Algorithm 2 is lower than the ratio for Algorithm 1 by a factor of q.

It can be seen from the above analysis that Algorithm 2 requires $2c_0(N - 2p)$ less time for communication. However, Algorithm 1 takes less time for floating–point operations than Algorithm 2. Usually the startup time for communication is much greater than one floating–point arithmetic operation in a system. If N is not large, Algorithm 2 may give better performance. However, if $N >> p$, Algorithm 1 will be more efficient than Algorithm 2.

Both the above described algorithms have been implemented on the Fujitsu AP 1000. While implementing algorithm 2, we made a minor modification in order to enhance the active/idle ratio, that is, we allow columns with an up-and-down arrow in Fig. 6 to generate Jacobi pairs like the other columns. Thus some Jacobi pairs will be produced more than once in one sweep. By doing this we expect that the total number of sweeps may be reduced for solving a given problem. One set of our experimental results is shown in Table 1. In this experiment a 5×5 array is used. We can see that the results agree with our analysis. When N is small, Algorithm 2 is more efficient. This is because the startup time in the AP 1000 is about 100 μs for each communication call. However, Algorithm 1 is preferable as the problem size becomes larger.

Problem size N		50	70	90	110	130	150	170	190
Algorithm 1	time(sec.)	1.04	2.26	4.50	7.10	11.2	17.4	22.4	34.0
	sweeps	11	12	13	13	13	14	14	14
Algorithm 2	time(sec.)	0.731	2.03	3.81	7.16	11.1	17.0	23.5	34.1
	sweeps	10	12	12	13	12	13	12	13
Problem size N		210	230	250	270	290	310	330	350
Algorithm 1	time(sec.)	45.0	60.6	76.5	98.4	119	140	180	219
	sweeps	14	15	15	16	16	15	16	17
Algorithm 2	time(sec.)	45.4	62.3	85.2	99.2	126	160	183	225
	sweeps	13	14	15	14	14	15	14	14

Table 1: Experimental results

Even though the number of sweeps is reduced by using the modified version of Algorithm 2, it still takes longer time than Algorithm 1 when N is large.

6 Conclusions

This paper has described two practical parallel algorithms (with partitioning) for solving eigenvalue problems on memory distributed machines. The algorithms require a minimum number of data communications between processing elements when implemented on a 2–D mesh connected system. We have shown both theoretically and experimentally that Algorithm 2 gives better performance when the problem size is small, while Algorithm 1 is more efficient as the problem size becomes larger. We hope that in a near future startup time for communication in parallel machines will be dramatically reduced with more advanced technology so that Algorithm 1 will become more attractive even in the case of small problem size.

References

[1] R. P. Brent (editor), *Proceedings of the CAP Workshop '91*, Australian National University, Canberra, Australia, November 1991.

[2] R. P. Brent and M. Ishii (editors), *Proceedings of the First CAP Workshop*, Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

[3] G. H. Golub and C. F. van Loan, "Matrix Computations", The Johns Hopkins University Press, Baltimore, MD, second ed., 1989.

[4] T. J. Lee, F. T. Luk and D. L. Boley, "Computing the SVD on a fat-tree architecture", preprint, 1992.

[5] F. T. Luk and H. Park, "On parallel Jacobi orderings", SIAM J, Sci. and statist. comput., 10, 1989, pp. 18–26.

[6] R. Schreiber, "Solving eigenvalue and singular value problems on an undersized systolic array", *SIAM. J. Sci. Stat. Comput.*, 7, 1986, pp. 441–451.