Parallel Computation of the Singular Value Decomposition on Tree Architectures*

Zhou B. B. and Brent R. P.†
Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

Report TR-CS-93-05
January 1993
(revised May 1993)

Abstract

We describe three new Jacobi orderings for parallel computation of SVD problems on tree architectures. The first ordering uses the high bandwidth of a perfect binary fat-tree to minimise global interprocessor communication costs. The second is a new ring ordering which may be implemented efficiently on an ordinary binary tree. By combining these two orderings, an efficient new ordering, well suited for implementation on the Connection Machine CM5, is obtained.

1 Introduction

Let A be a real $m \times n$ matrix. Without loss of generality we assume that $m \geq n$. The singular value decomposition (SVD) of A is its factorization into a product of three matrices

$$A = U\Sigma V^T,$$

where U is an $m \times n$ matrix with orthonormal columns, V is an $n \times n$ orthogonal matrix, and Σ is an $n \times n$ non-negative diagonal matrix, say $\Sigma = diag(\sigma_1, \cdots, \sigma_n)$.

There are various ways to compute the SVD [6]. To achieve efficient parallel SVD computation the best approach may be to adopt the Hestenes one-sided transformation method [7] as advocated in [2].

The Hestenes method generates an orthogonal matrix V such that

$$AV = H,$$

where the columns of H are orthogonal. The nonzero columns $\tilde{H}$ of H are then normalised so that

$$\tilde{H} = U_r\Sigma_r$$

with $U_r^T U_r = I_r$, $\Sigma_r = diag(\sigma_1, \cdots, \sigma_r)$ and $r \leq n$ is the rank of A.

The matrix V can be generated as a product of plane rotations. Consider the transformation by a plane rotation:

$$\begin{pmatrix} a_i & a_j \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} a_i' & a_j' \end{pmatrix} \tag{1}$$

*Copyright © 1993, the authors. Accepted for presentation as a concise paper at the 22nd International Conference on Parallel Processing, St. Charles, Illinois, August 1993.

†E-mail addresses: {bing,rpb}@cslab.anu.edu.au

rpb138tr typeset using LaTeX

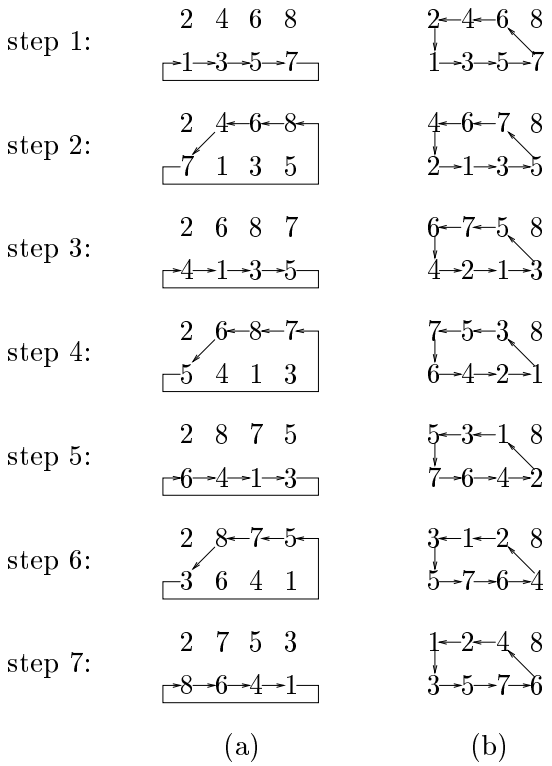

Figure 1: (a) the ring ordering and (b) the round robin ordering.

where $c = \cos\theta$, $s = \sin\theta$, and a_i and a_j are the i-th and j-th columns of the matrix A. We choose θ to make $a_i{}'$ and $a_j{}'$ orthogonal. As in the traditional Jacobi algorithm, the rotations are performed in a fixed sequence called a *sweep*, each sweep consisting of $n(n-1)/2$ rotations, and every column in the matrix is orthogonalised with every other column exactly once per sweep. The iterative procedure terminates if one complete sweep occurs in which all columns are orthogonal and no columns are interchanged. If the rotations in a sweep are chosen in a reasonable, systematic order, the convergence rate is ultimately quadratic [6]. Exceptional cases in which cycling occurs are easily avoided by the use of a threshold strategy [16].

It is possible to choose the rotations so that the, on convergence, the singular values emerge sorted in decreasing (or increasing) order of size. This is convenient for applications where sufficiently small singular values are regarded as zero.

It can be seen from equation (1) that one Jacobi plane rotation operation only involves two columns. Therefore, there are disjoint operations which can be executed simultaneously. In a parallel implementation, we want to perform as many non-interacting operations as possible at each parallel time step. Many parallel orderings have been introduced in the literature [1, 2, 3, 4, 5, 8, 10, 11, 12]. Two examples, the ring ordering [3] and the round robin ordering [2], are depicted in Fig. 1. In the Figures and throughout the paper, a transformation by a plane rotation is denoted by an index pair (i, j).

In this paper we present three new orderings which may be implemented efficiently on the tree architectures described in Section 2. The first ordering may be called a "fat-tree ordering" because it uses the high bandwidth of a fat-tree to minimise global interprocessor communication costs. The second ordering is a new ring ordering, which has some advantages over the ring ordering depicted in Fig. 1(a). Combining the two orderings, we obtain an efficient ordering suitable for "skinny" fat-tree architectures of the CM5 type.

The paper is organised as follows: Section 2 describes various tree architectures. Our fat-tree ordering is described in Section 3 and compared with the (different) fat-tree ordering of [8]. A new ring ordering, and a hybrid ordering suitable for the CM5 supercomputer are discussed in Sections 4 and 5. Our conclusions are given in Section 6.

2 Fat-Tree Architectures

A fat-tree, based on a complete binary tree, is a routing network for parallel communication [9]. In a fat-tree a set of processors is located at the leaves of the tree and there are two channels corresponding to each edge, that is, one from parent to child and the other from child to parent. The number of wires in a channel is called the *capacity* of the channel. If the levels from bottom (the leaves) up are numbered $1, 2, \ldots$ and the capacity of the channels at level 1 is γ, the capacity of the channels at level k is given by $2^{k-1}\gamma$ for a (perfect) binary fat-tree. In other words, the capacity of the channels in the tree is increased by a factor of two for each increase in level. Thus, the overall communication bandwidth at each level is constant. The fat-tree routing network can support very large multiprocessor systems.

Following [8], a tree is called a *skinny* fat-tree if the capacity of some channels does not grow as fast as it should in the case of a perfect fat-tree. We are interested in two types of skinny fat-tree. The first is an ordinary complete binary tree which is "skinny all over", i.e. its channels have the same capacity at all levels. The second is a tree in which the channels are skinny (relative to a perfect fat-tree) only above a certain level. ("Above" means "closer to the root" – as usual in Computer Science, our trees have their leaves at the bottom.)

The Connection Machine CM5 [15] almost fits into the second category. Although the CM5 network is a 4-way tree, it can be regarded as equivalent to a skinny binary fat-tree. The bottom level of the 4-way tree is equivalent to the bottom two levels of a perfect binary fat-tree. Only the channels above that level in the machine are skinny. At each level of the 4-way tree the channel capacity increases by a factor of 2 (not 4), so the 4-way tree is equivalent to a binary tree in which the channel capacities increase by factors of $2, 1, 2, 1, \ldots$ (or, approximately, by a factor of $\sqrt{2}$ at each level). The decision to skimp on channel capacities at high levels of the tree seems to invalidate the theoretical reasons [9] for choosing a fat-tree for the CM5 architecture.

A problem which is compute-bound on a serial computer may be communication-bound on a parallel computer. Thus a key issue in designing a parallel algorithm for a given problem is how to minimise the communication cost so that the computational capability of a parallel machine can be exploited to the full. Experimental results on the CM5 [13] suggest that, in order to achieve high performance on a (skinny) fat-tree architecture, communication should be kept local (especially for large messages) and contention should be avoided as far as possible.

3 Fat-Tree Orderings

When considering tree orderings we assume for convenience that n is a power of 2. We say that a communication is a *level-r* communication if the number of levels that a message from one leaf to another has to move up through the fat-tree (before coming down to its destination) is r. Thus, nearest neighbour communication between siblings in a tree architecture is level-one communication. Although the orderings illustrated in Fig. 1 may be implemented on a fat-tree architecture, they have the disadvantage that global communication is required at each step. Since locality is important for reducing the communication cost in a fat-tree architecture, new orderings are needed.

A fat-tree ordering was recently introduced in [8]. In the ordering of [8], most communications are local, and global communications are minimised. After one sweep the indices are permuted,

which is inconvenient for the SVD computation because the singular vectors (columns of V) end up in the "wrong" processors. (This only matters if, as above, we want the singular values to be ordered.) To overcome this problem, [8] recommends alternating *forward* and *backward* sweeps, where the backward sweep is just the forward sweep performed in reverse order. After each pair of sweeps the indices (and columns of V) are in the correct processors. The first rotation in each backward sweep does nothing, and may be omitted, because it operates on the same pair as the last rotation in the preceding forward sweep. The disadvantages of the scheme recommended in [8] are –

1. Convergence may be slower than usual, because the number of rotations between any fixed pair (i, j) is variable rather than constant.

2. On average an extra half-sweep has to be performed if the number of sweeps to termination has to be an *even* integer.

In this section we introduce a new fat-tree ordering. The communication cost is about the same as for the ordering of [8]. Only one procedure is required for every sweep, and the original order of the indices is maintained after the completion of each sweep. Therefore, our ordering avoids the problems noted above for the ordering of [8]. Also, as shown in Section 5, our fat-tree ordering may be combined with a ring ordering to make a very efficient hybrid ordering for solving SVD problems on skinny fat-tree architectures like the CM5.

Our fat-tree ordering is made up from two basic orderings, the *two-block* ordering and the *four-block* ordering, which are defined in Sections 3.1–3.2.

3.1 The two-block ordering

Suppose that there are two blocks, each containing 2^k indices. The objective of the two-block ordering is to let each index in one block meet each index in the other block once, so 2^{2k} different index pairs are generated. In the discussion below an ordering is called an ordering of size 2^k (or size 2^k ordering) if each block holds 2^k indices.

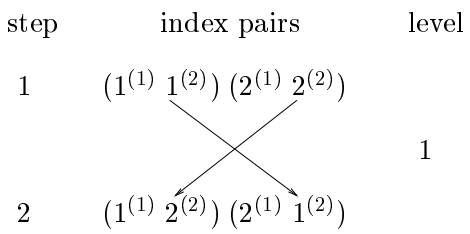

Figure 2: Basic module for two-block ordering.

3.1.1 Basic module for two-block ordering

The basic module for our two-block ordering is depicted in Fig. 2. In the figure each block contains only two indices. The superscript (i) on each index in the figure indicates to which block that particular index belongs. Since there are only two indices in each block, the procedure (or a sweep of the ordering) takes only two steps to complete. At the first step, the indices from the two blocks are interleaved, forming two index pairs. The two indices in block 2 (or block 1) are then interchanged so that another two index pairs are generated at the second step.

We first suppose that each leaf on the tree holds only two indices. Communication is required if indices from different leaves are to be interchanged. It can easily be seen that our basic module requires only level-one communication if the block size is two, which results in minimal communication cost on a tree architecture. Therefore, in the derivation of our fat-tree ordering we always divide a large problem into a number of problems of size 2 in order to minimise the total communication cost. Also, the two indices in block 2 are exchanged after a sweep. If the same procedure is repeated once, the order of indices will be restored. This is also true when the block size is greater than two.

3.1.2 Divide and conquer technique

We now consider the case where one block holds more than two indices. We apply the divide and conquer technique, that is, a large problem is first divided into smaller sub-problems, the sub-problems are solved, and the sub-results are combined to obtain a result for the original problem. In the following a block is called a *rotating* block if the two indices (or two sub-blocks of indices) in the block exchange their positions during a two-block ordering. For example, see block 2 in Fig. 2.

We first consider the ordering of size $4 = 2^2$. In this ordering each block is first divided into two sub-blocks, each containing two indices. If each sub-block is considered as a super-index, the basic module may be applied. Since one super-index contains two indices, each super-index pair forms a sub-problem of size 2. Therefore, we have actually divided the original problem into four half-size sub-problems. These sub-problems are solved in two super-steps, two at a time. The ordering is illustrated in Fig. 3.

Since there are interchanges of indices between sub-blocks (or super-indices), a level-two communication is required between the two super-steps. It can be seen from Fig. 3 that the two sub-blocks (1, 2) and (3, 4) in the second block have exchanged their positions after one sweep. However, the original order of the indices within each sub-block is maintained. This is because we always let the sub-blocks from the original second block be the rotating blocks when the ordering of size 2 is applied, and these sub-blocks are rotated twice during the computation. If the same procedure is executed once again the level-two communication is performed twice. Thus the order of the indices in block 2 will be restored. The indices in block 1 do not change

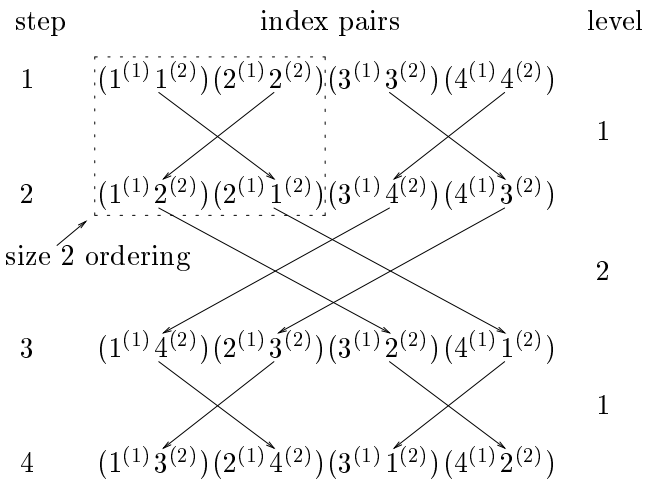

Figure 3: The two-block ordering of size 4

their positions during the computation.

The idea can easily be extended to the general case. Suppose that the problem size is 2^k, that is, each block holds 2^k indices, $k > 1$. To solve this problem we first divide it into four half-size sub-problems. These four sub-problems may be solved in two super-steps by applying the basic module, and a level k communication is required between the super-steps. Each sub-problem may be further divided into four sub-subproblems. The procedure continues until we obtain a number of size-two problems. The computation then starts from bottom up. If we always use the sub-blocks from the original second (or first) block as rotating blocks, after a sweep of the ordering the two sub-blocks in the original second (or first) block will exchange their positions, but the original order of indices inside each sub-block is maintained.

3.2 The four-block ordering

Suppose that we have four blocks, each containing 2^k indices. Our aim is to let each of the 2^{k+2} indices meet each other exactly once in a sweep of the ordering, to generate a total number of $2^{k+1}(2^{k+2} - 1)$ different index pairs.

3.2.1 Basic module for four-block ordering

We first consider the simplest case, where there are only four indices involved in the ordering. To generate six different index pairs one sweep of the ordering requires three steps. There are many ways to do this; two of them are depicted in Fig. 4.

If we enumerate the indices from the left, starting with 1, the original order of the indices will be (1, 2, 3, 4). This order is maintained after a sweep with the first ordering depicted in Fig. 4(a). However, with the second ordering depicted in Fig. 4(b) the positions of indices 3 and 4 are reversed after the first sweep, and the order is only restored after two consecutive sweeps of the ordering.

The first algorithm has another advantage. It can be seen from Fig. 4(a) that the left index in any index pair is always smaller than the right index. If we store the column with larger norm on the left after each step of the SVD computation, then the singular values are obtained in nonincreasing order.

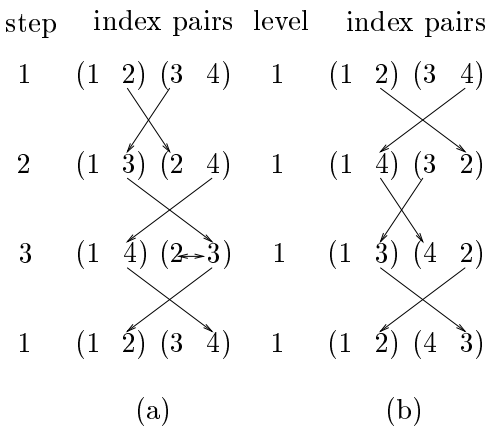

Figure 4: Basic modules for four-block ordering.

Note that in Fig. 4(a) there is a left-right arrow in an index pair in step 3. This indicates that the two indices in that pair have to be swapped before the communication between index pairs takes place for the next step. This implies that the two associated columns have to be exchanged in the SVD computation, which may degrade performance. To overcome this problem we use the following procedure. For the two given columns a_i and a_j, a transformation as in equation (1) should be applied. Since the two computed columns have to be swapped, we have

$$\begin{pmatrix} a_i'' & a_j'' \end{pmatrix} = \begin{pmatrix} a_i' & a_j' \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2}$$

Combining equations (1) and (2) yields

$$\begin{pmatrix} a_i'' & a_j'' \end{pmatrix} = \begin{pmatrix} a_i & a_j \end{pmatrix} \begin{pmatrix} -s & c \\ c & s \end{pmatrix} \tag{3}$$

Therefore, using transformation (3) instead of (1) avoids the need to explicitly interchange columns.

3.2.2 Block ordering

We now describe the scheme for the block ordering. The more general scheme is presented in Section 3.3.

Assume that the four blocks of indices are formed into two groups and each group holds two blocks. If each block is considered as a super-index, applying the basic operating procedure will allow these super-indices to meet each other once in three super-steps. Since each block contains 2^k indices, more computations are expected in each of the super-index pairs. In our block ordering, the computation for the super-index pairs in super-step 1 will be different from that in the other two super-steps. In super-step 1 we let the indices in the same group meet each other once, in other words, we allow not only the indices in each block to meet each other once, but also each index in one block to meet each index in the other block of the same group. Since the indices in the same block have met each other once already, in the next two super-steps each index in one block has only to meet each index in the other block when the two blocks from different groups form a super-index pair.

In Section 3.3 we prove that the original order of the indices is maintained after a sweep of our fat-tree ordering when the ordering procedure depicted in Fig. 4(a) is applied.

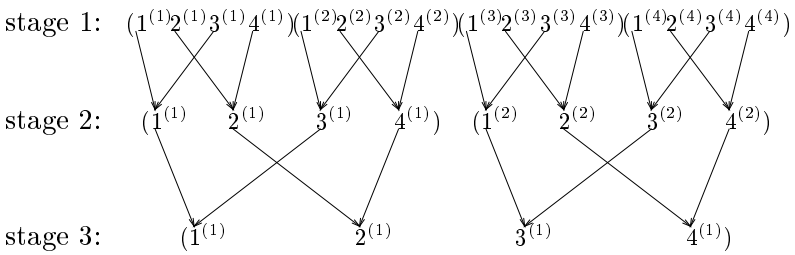

Figure 5: The scheme of our merge procedure.

3.3 The merge procedure

Our fat-tree ordering algorithm is derived by using the following merge procedure. Suppose that there is a total number of 2^n indices. To begin the procedure these indices are first organised into 2^{n-2} groups, each holding only four indices. The four-block ordering is then applied so that the indices in each group will meet each other once. Next each pair of two consecutive groups is combined to form a super-group. Each group in a super-group is also divided into two blocks, so there are four blocks in each super-group. If each block is considered as a super-index, the four-block ordering may be applied. Each two consecutive super-groups may further form a super-supergroup and the four-block ordering is once again applied. The operation terminates if the 2^n indices are just in a big group and the four-block ordering applied to this big group is completed. The scheme of our merge procedure is depicted in Fig. 5. It is easy to realise from the figure that it will take $n-1$ stages to complete a sweep of the ordering for a total number of 2^n indices. It should be noted that our objective is to let the 2^n indices meet each other exactly once in a sweep. Thus at stage k the two indices are not allowed to meet if they have met at a previous stage of the same sweep.

In the merge procedure we continuously merge each two consecutive groups into one bigger group and allow the indices in the bigger group to meet each other once if they have not met previously in the same sweep. Thus the key is how to derive a *good* method for this kind of merge. By a *good* method we mean that the communication should be regular, the communication cost should be as low as possible, the original order of the indices should be maintained after one sweep, and the method should be extensible to larger problems.

Example

Consider the case $n = 3$, $2^n = 8$. We first divide the indices into two groups. Each group holds four consecutive indices. After a four-block ordering procedure applied to each group, the indices in the same group meet each other once. The two groups are then merged to form a super-group. The four blocks of indices in the super-group are organised in such a way that the indices in blocks 1 and 2 from the left group are interleaved and the indices in block 3 and 4 from the right group are organised in the same manner. To be specific, blocks 1, 2, 3 and 4 contain indices $(1^{(1)}, 3^{(1)})$, $(4^{(1)}, 2^{(1)})$, $(1^{(2)}, 3^{(2)})$ and $(4^{(2)}, 2^{(2)})$, respectively (see step 4 in Fig. 6). Note that the indices in each original group have already been combined with each other in the previous stage, which is exactly the computation required in super-step 1 of the four-block ordering of Section 3.2.2. Thus, only super-steps 2 and 3 remain to be performed. Since the blocks are interleaved in each super-index pair, the two-block ordering procedure may be applied to let the indices from different blocks in each super-index pair meet each other once, which completes the merge procedure. The details are illustrated in Fig. 6.

In the above example the indices in block 2 and block 4 are in a reverse order after super-

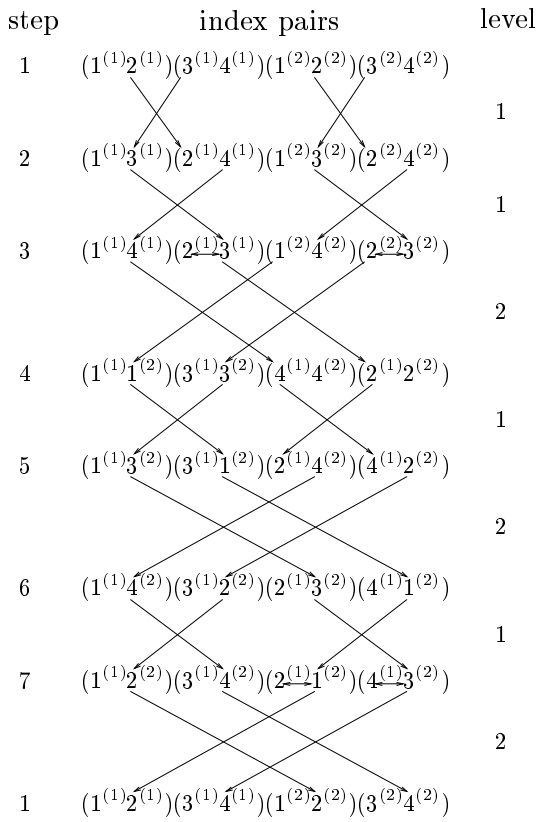

Figure 6: The four-block ordering for eight indices.

step 1 (or the first three steps). According to our four-block ordering block 2 and block 3 are then interchanged so that the left super-index pair contains block 1 and 3 while the right holds block 2 and 4 before the computation of super-step 2 starts. If block 2 and block 3 are considered as the rotating blocks in the two different super- indices, after a sweep of the two-block ordering (or the second super-step) the indices in these blocks will be exchanged. Thus the order of block 2 is normal, but the order of block 3 is reversed. Before the third super-step block 3 and block 4, both with a reversed index order inside, are interchanged. If we consider these two blocks as the rotating blocks, obviously the order of indices in both blocks will become normal after super-step 3. Therefore, the order of all the eight indices will return to the original after one sweep of the four-block ordering or after the merge procedure.

The method is easily extensible to problems of larger sizes. Suppose that after the $k-1$ stages the original order is maintained for the problem of a total number of 2^k indices. We prove that the result holds when the number of indices is 2^{k+1}. After $k-1$ stages there are two groups, each holding 2^k indices, to be merged. Each group is divided into two blocks in the same way as discussed in the previous example. We further cut each block in half and consider the left half and the right half as two sub-blocks. The indices in each sub-block must be in the original order and the order of the two sub-blocks in blocks 2 and 4 must be reversed. Otherwise the order will not return to the original for a problem with a total of 2^k indices. Note that the indices in each group have already met each other once in the previous $k-1$ stages. This completes super-step 1 of a sweep of the four-block ordering. In the next two super-steps only the two-block ordering procedure is required to allow each index in one block to meet every index in the other block of the same super-index pair. ¿From Section 3.1, the order of the two sub-blocks in the rotating

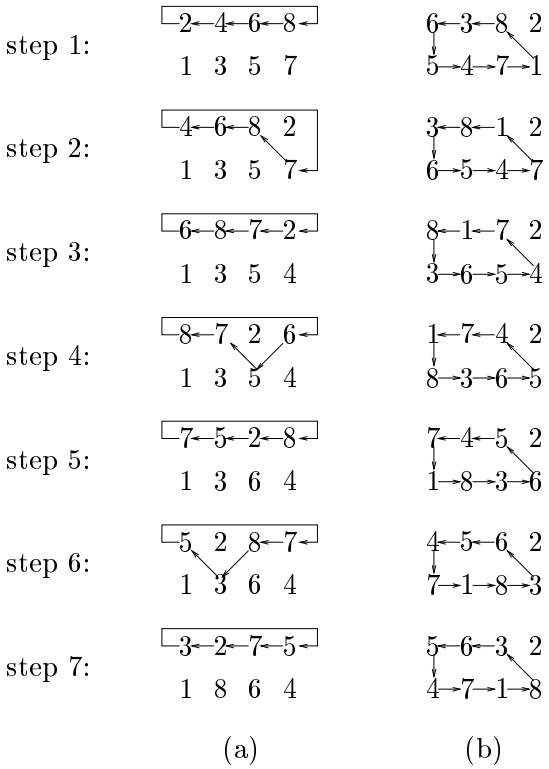

Figure 7: (a) a new ring ordering and (b) its equivalent round robin ordering.

block is reversed, but the original order of the indices in each sub-block is maintained after one sweep of the two-block ordering. Since the sub-blocks in block 2 and 4 are rotated once and the sub-blocks in block 3 are rotated twice, the order of all the indices will certainly be restored after the k-th stage of the merge procedure.

4 Ring Orderings

The ring ordering algorithm depicted in Fig. 1(a) is well suited for implementation on an ordinary tree architecture, because it needs only one-way communication and the messages can be evenly distributed on the tree without contention. This section describes a new ring ordering. One important feature of the ordering is that the messages travel between processors in only one direction throughout the computation. Thus it has an advantage over the ordering of Fig. 1(a) in certain situations, especially when custom VLSI implementation is considered.

Our new ring ordering algorithm is described by an example, as shown in Fig. 7(a). In the figure the two indices in the same column form an index pair, and after each step some indices are shifted to new places as shown by the arrows. Suppose that there are n indices. A sweep of the ordering takes $n-1$ steps for n even (or n steps for n odd). After a sweep the positions of indices 1 and 2 are unchanged, while the order of the indices numbered from 3 to n is reversed. However, it is easy to verify that all the indices will return to their original positions after another sweep with the same procedure. Therefore, the original order of the indices is restored after two consecutive sweeps with our ring ordering.

Definition Let O_i be a parallel Jacobi ordering, and let one sweep of the ordering for n indices be represented as $S_i^{(n)}$. We say that two orderings O_1 and O_2 are equivalent if $S_1^{(n)}$ can

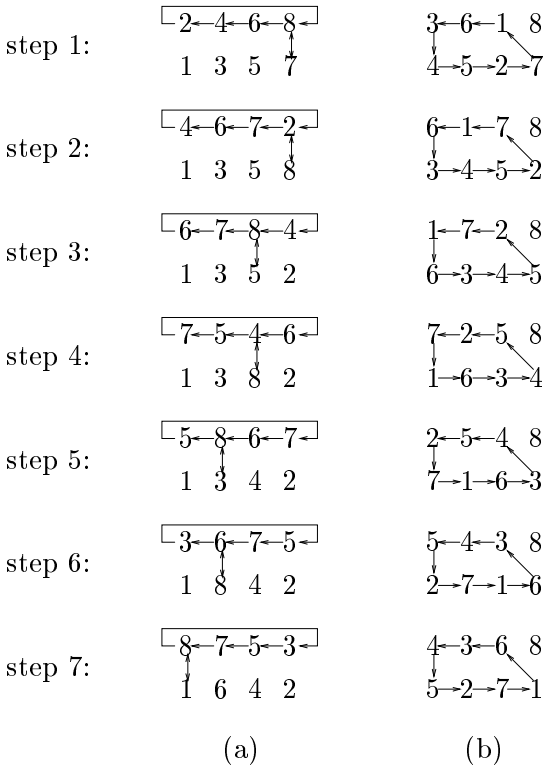

Figure 8: (a) a modified version of the ring ordering and (b) its equivalent round robin ordering.

be obtained from $S_2^{(n)}$ by a relabelling of indices [12].

If two orderings are proved to be equivalent, they will have the same convergence properties. We now show that our ring ordering is equivalent to the round-robin ordering in Fig. 1(b).

To prove the equivalence we first permute the original positions of the indices in Fig. 1(b). Divide the original index pairs into two equal parts. (If the number is odd, the left part will contain one more index pair than the right part.) Next swap the positions of the two indices in each index pair of the left part, except the leftmost one. Then fold the two parts together from the middle so that the index pairs in the two parts are interleaved. After the above permutation we apply the round-robin ordering to the "relabelled" indices. The same index pairs for each step in the ring ordering can then be generated (see Fig. 7), which shows the equivalence of the two orderings.

It can be seen from Fig. 7(a) that the number on the second row is smaller than the one on the first row of the same index pair, except for the pairs containing index 2. With a little control we may store the column with larger norm in the position associated with the index of a smaller number during the SVD computation. This ensures that the singular values are obtained in nonincreasing order after an even number of sweeps.

A modified version of the ring ordering is depicted in Fig. 8(a). Using this ordering we may obtain the singular values in nonincreasing order after an even number of sweeps, but in nondecreasing order after an odd number of sweeps.

5 A Hybrid Ordering

Since the channels above a certain level in a skinny fat-tree are skinny compared to those in a perfect fat-tree, contention will occur if our fat-tree ordering is implemented on such an architecture. This can cause serious performance degradation [13]. To avoid this problem we introduce a new ordering called a *hybrid* ordering. Our hybrid ordering is a combination of the fat-tree ordering and the ring ordering. We first consider the block ring ordering, which is based on Schreiber's partitioning method [14]. Suppose that there are 2^n indices. We divide the indices into 2^m groups, each holding 2^{n-m} consecutive indices. In each group the indices are further divided into two blocks. Indices in the two blocks are interleaved. Consider each block as a super-index. By applying the ring procedure described in the previous section, the blocks will be combined with each other once in $2^m - 1$ super-steps. Since each block holds more than one index, when two blocks meet in a group we let each index in one block combine once with each index in the other block of the same group. If we also allow the indices within each block to meet each other once in super-step 1, we can guarantee that a total number of $2^{n-1}(2^n - 1)$ different index pairs can be generated in a sweep of the ordering.

The computation described above for each group may be performed in parallel, which leads to our hybrid ordering. In super-step 1 the fat-tree ordering is applied within each group so that the indices in the same group will meet each other once. Since the two blocks in a group are interleaved, the two-block ordering may be applied to let each index in one block combine once with each index in the other block of the same group in the succeeding super-steps. The communications between the super-steps are the same as those in the block ring ordering. Since the messages are evenly distributed at each super-step when the block ring ordering is used, we may properly choose the block size so that the number of messages passing through the lowest skinny level do not cause contention. Therefore, it is guaranteed that no contention will occur anywhere in the tree.

The positions of block 2 and 4 are reversed in the last step of the fat-tree ordering (see Fig. 4 and Fig. 6), and the two sub-blocks in the rotating block are interchanged after a sweep of the two-block ordering. Special care has to be taken to ensure that the order of the indices in each block is maintained after a sweep of the hybrid ordering. If this can be done, the order of the indices will be restored after two consecutive sweeps of the ring ordering. The key to achieve this is to rotate the two sub-blocks in a block an *even* number of times if they have to be rotated at all during the ordering.

We can see from Fig. 7(a) that index 1 stays during a sweep of the ordering. Index 2 moves once every two steps and returns to its original position. Since we have assumed that the number of groups is 2^m in our hybrid ordering, index 2 will be shifted an even number of times after a sweep. For the rest of the indices, the order is reversed after a sweep. It is easy to see that indices 3 and 4 have to move twice to arrive to their destinations. Indices 5 and 6 are originally next to and to the right of indices 3 and 4. However, they move to the left of indices 3 and 4 after a sweep. Thus it takes two additional steps for indices 5 and 6 to reach their destinations. For the same reason indices 7 and 8 take two more steps than indices 5 and 6 to reach their destinations, and so on. Therefore, all the indices (except index 1, which does not move), are shifted an even number of times after a sweep. If we consider a block as a rotating block if it is to be shifted, or if it is on the ring formed by the arrows in each super-step, the two sub-blocks in that block will be rotated an even number of times, so the original order of the indices inside each block will be maintained after a sweep of the ordering. The similar result can also be obtained when the ring ordering algorithm depicted in Fig. 8 is used.

An example of our hybrid ordering is depicted in Fig. 9. In that example, sixteen indices are divided into four groups. The fat-tree ordering is applied within each group, while the communications between groups are carried out using the ring ordering depicted in Fig 7.

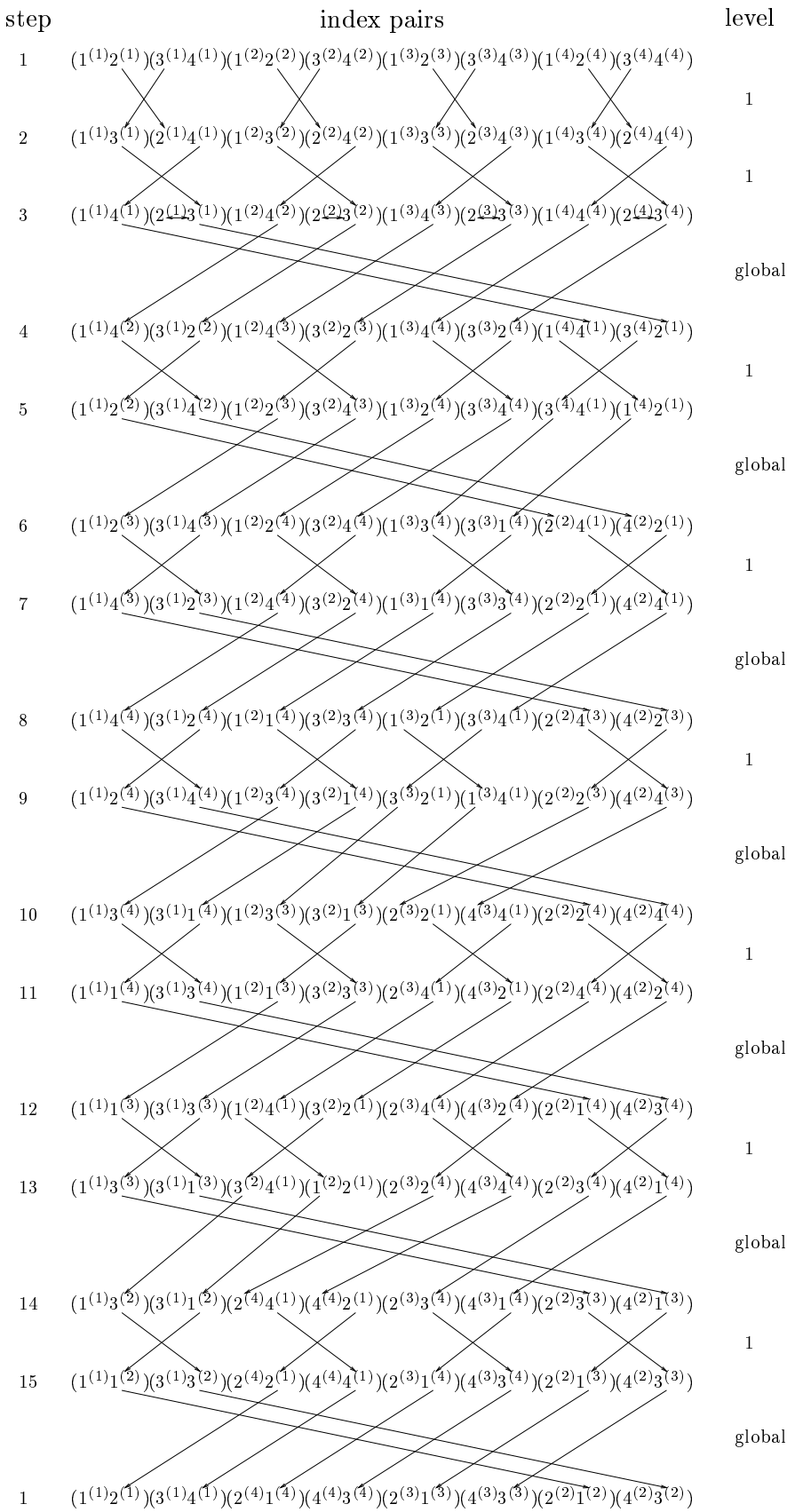

Figure 9: The hybrid ordering for sixteen indices

6 Conclusions

Three new Jacobi ordering algorithms for parallel computation of SVD problems on tree architectures have been introduced. They are currently being implemented on a 32-node CM5 at the Australian National University. It is expected that the hybrid ordering will be the most efficient one on the CM5 since it does not cause any contention and reduces the number of global communications required by the ring orderings. If communication-handling capability is increased, then our fat-tree ordering will become more attractive.

Acknowledgement

We thank Franklin Luk for showing us a preprint of [8], and for his helpful comments on an early version of this paper.

References

[1] C. H. Bischof, "The two-sided block Jacobi method on a hypercube", in *Hypercube Multiprocessors*, M. T. Heath, ed., SIAM, 1988, pp. 612-618.

[2] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays", *SIAM J. Sci. and Statist. Comput.*, 6, 1985, pp. 69-84.

[3] P. J. Eberlein and H. Park, "Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures", *J. Par. Distrib. Comput.*, 8, 1990, pp. 358-366.

[4] L. M. Ewerbring and F. T. Luk, "Computing the singular value decomposition on the Connection Machine", *IEEE Trans. Computers*, 39, 1990, pp. 152-155.

[5] G. R. Gao and S. J. Thomas, "An optimal parallel Jacobi-like solution method for the singular value decomposition", in *Proc. Internat. Conf. Parallel Proc.*, 1988, pp. 47-53.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, second ed., 1989.

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

[8] T. J. Lee, F. T. Luk and D. L. Boley, *Computing the SVD on a fat-tree architecture*, Report 92-33, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, November 1992. Also *Proc. NATO ASI on Linear Algebra for Large Scale and Real-Time Applications*, Volume 232, Leuven, Belgium, August 1992, pp. 231-240.

[9] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing", *IEEE Trans. Computers*, C-34, 1985, pp. 892-901.

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

[11] F. T. Luk, "A triangular processor array for computing singular values", *Lin. Alg. Applics.*, 77, 1986, pp. 259-273.

[12] F. T. Luk and H. Park, "On parallel Jacobi orderings", *SIAM J. Sci. and Statist. Comput.*, 10, 1989, pp. 18-26.

[13] R. Ponnusamy, A. Choudhary and G. Fox, "Communication overhead on CM5: an experimental performance evaluation", in *Frontiers '92*, Proc. Fourth Symp. on the Frontiers of Massively Parallel Computation, IEEE, 1992, pp. 108-115.

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

[15] Thinking Machines Corporation, *The Connection Machine CM5 Technical Summary*, October 1991.

[16] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965, pp. 277-278.