PARALLEL COMPUTATION OF THE SINGULAR VALUE DECOMPOSITION ON TREE ARCHITECTURES*

Zhou B. B. and Brent R. P.[1]
Computer Sciences Laboratory
Australian National University
Canberra, ACT 0200

Abstract *We describe a new Jacobi ordering for parallel computation of SVD problems. The ordering uses the high bandwidth of a perfect binary fat-tree to minimise global interprocessor communication costs. It can thus be implemented efficiently on fat-tree architectures.*

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 $n \times n$ nonnegative diagonal matrix, say $\Sigma = diag(\sigma_1, \cdots, \sigma_n)$.

There are various ways to compute the SVD [2]. To achieve efficient parallel SVD computation the best approach may be to adopt the Hestenes one-sided transformation method [3] as advocated in [1].

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. 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 [2].

Since one Jacobi plane rotation operation only involves two columns, 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.

In this paper we present a new parallel Jacobi ordering. This ordering may be called a *fat-tree ordering* because it uses the high bandwidth of a fat-tree to minimise global interprocessor communication costs. Thus it can be implemented efficiently on the fat-tree architectures.

The paper is organised as follows: Section 2 briefly describes fat-tree architectures. Our fat-tree ordering is described in Section 3 and compared with the (different) fat-tree ordering of [4]. Our conclusions are given in Section 4.

2 Fat-Tree Architectures

A fat-tree, based on a complete binary tree, is a routing network for parallel communication [5]. 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. If a factor of less than two is used (as in the CM5), we say that the tree is a *skinny* fat-tree.

A problem which is compute-bound on a serial computer may be communication-bound on a parallel com-

 To appear in *Proceedings 22nd International Conference on Parallel Processing*, St Charles, Illinois, August 1993.

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

rpb138 typeset using LaTeX

puter. 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 [6] 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 Ordering

In the following discussion 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.

A fat-tree ordering was recently introduced in [4]. In the ordering of [4], most communications are local, and global communications are minimised. However, the disadvantages of the scheme recommended in [4] 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. The logic to generate forward and backward sweeps is more involved than the logic to generate just a forward sweep.
3. On average an extra half-sweep has to be performed as 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 [4]. 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 all three problems noted above for the ordering of [4].

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.

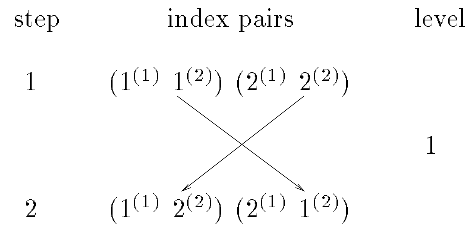

Figure 1: Basic module for two-block ordering.

The basic module for our two-block ordering is depicted in Fig. 1. 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 have assumed 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.

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. 1.

We only consider the ordering of size $4 = 2^2$. The idea can easily be extended to the general case. 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. 2.

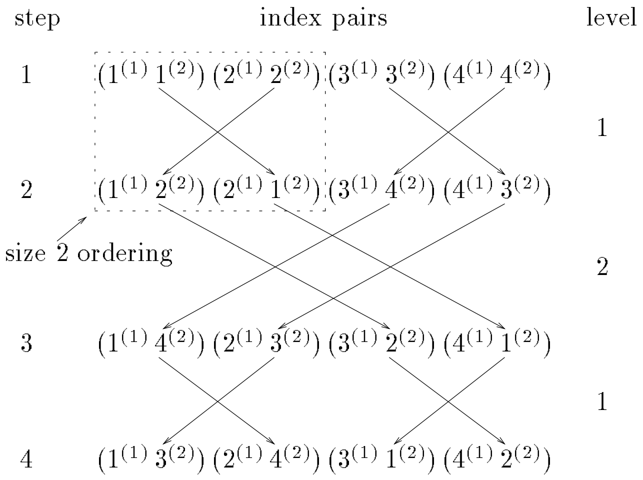

Figure 2: The two-block ordering of size 4.

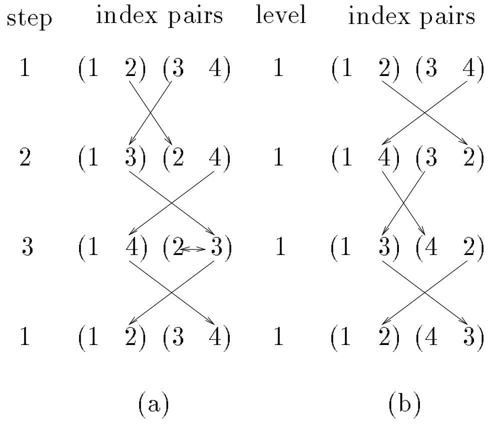

Figure 3: Basic modules for four-block ordering.

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. 2 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 their positions during the computation.

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.

We now 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. 3.

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. 3(a). However, with the second ordering depicted in Fig. 3(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. 3(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.

Note that in Fig. 3(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. However, this problem can easily be avoided. (See [8] for details.)

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.

It should be noted that our objective is to let the 2^n indices meet each other exactly once in a sweep. Thus the two indices are not allowed to meet if they have met at a previous stage of the same sweep. In the following we give an example to illustrate the merge procedure. The method is easily extensible to problems of larger sizes.

Consider the case $n = 3$, or $2^n = 8$. 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 Fig. 4). 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. 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. 4. It is clear that the order of the indices is unchanged by the merge procedure.

4 Conclusions

A new Jacobi ordering algorithm for parallel computation of SVD problems on fat-tree architectures has been introduced. It is currently being implemented on a 32-node CM5 at the Australian National University. Since the CM5 has a skinny fat-tree architecture, it is expected that the hybrid ordering described in [8], which is a combination of our fat-tree ordering and a ring ordering, will be the most efficient one, since that ordering does not cause any contention and reduces the number of global communications required by the ring ordering. If the CM5 used a perfect fat-tree, then our fat-tree ordering would be more attractive.

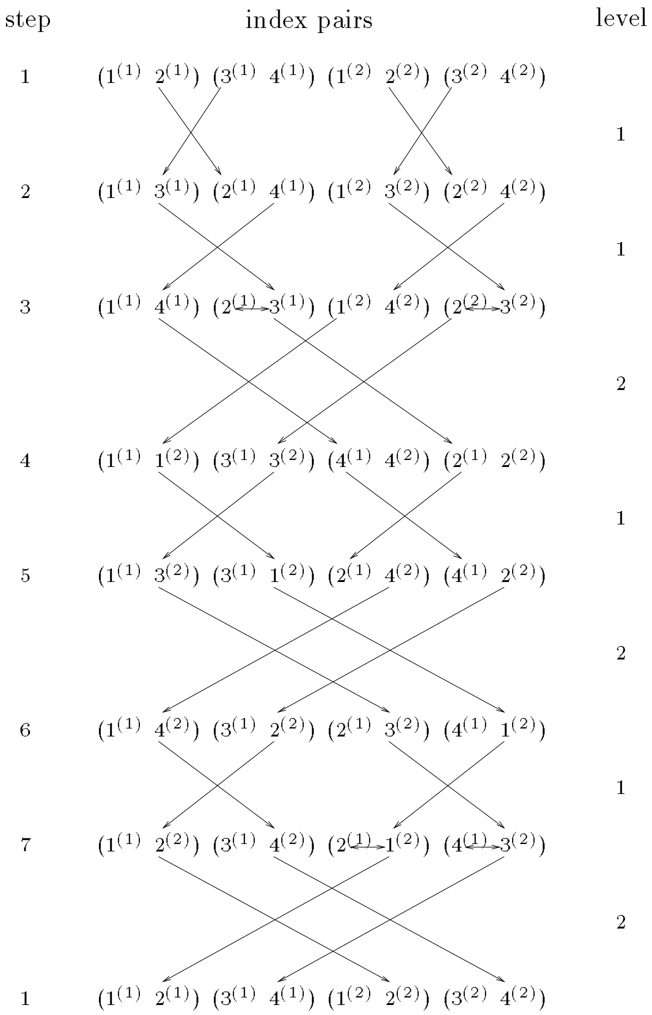

Figure 4: The four-block ordering for eight indices.

References

[1] 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.

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

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

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

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

[6] 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

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

[8] B. B. Zhou and R. P. Brent, *Parallel Computation of the Singular Value Decomposition on Tree Architectures*, Report TR-CS-93-05, Computer Sciences Laboratory, Australian National University, January 1993 (revised May 1993), 14 pp.