Parallel Implementation of QRD Algorithms on the Fujitsu AP1000*

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

Abstract

This paper addresses several important aspects of parallel implementation of QR decomposition of a matrix on a distributed memory MIMD machine, the Fujitsu AP1000. They include: Among various QR decomposition algorithms, which one is most suitable for implementation on the AP1000? With the total number of cells given, what is the best aspect ratio of the array to achieve optimal performance? How efficient is the AP1000 in computing the QR decomposition of a matrix? To help answer these questions we have implemented various orthogonal factorisation algorithms on a 128-cell AP1000 located at the Australian National University. After extensive experiments some interesting results have been obtained and are presented in the paper.

1 Introduction

Orthogonal QR decomposition [11] has many useful applications, e.g., to the solution of linear least squares problems, or as a preliminary step in the singular value decomposition.

The widely used methods for QR decomposition of a matrix A fall into two classes – those using Householder transformations and those using Givens rotations. The QR algorithms based on Givens rotations require about twice the number of arithmetic operations as the algorithms using Householder transformations, but the latter require about twice as many memory references as the former. On a single processor, Householder transformations are usually cheaper than Givens rotations because the cost of a floating point operation dominates the cost of a memory reference. However, which of the two methods is superior in parallel computation is very much dependent upon the machine configuration. For example, on the Denelcor HEP, a shared memory multiprocessor, memory access times dominate floating point operation times, and it is reported in [8] that Givens-based algorithms are twice as fast as Householder-based algorithms. Different results obtained on distributed memory machines can be found in [13, 15].

When implementing QR decomposition algorithms on the AP1000, we are interested in:

 To appear in *Proc. 17th Australian Computer Science Conference*, Christchurch, New Zealand, Jan. 1994.

†E-mail addresses: {bing,rpb}@cslab.anu.edu.au rpb145 typeset using LaTeX

- Of the various QR decomposition algorithms, which one is most suitable for implementation on this machine?
- When the total number of cells (processors) is given, what is the best aspect ratio of the cell array to achieve optimal performance?
- How efficient is the AP1000 in solving this particular problem?

To answer these questions, we implemented various orthogonal decomposition algorithms on the AP1000. This paper presents some results obtained through extensive tests of these algorithms.

The manner in which matrices are stored is very important on a distributed memory machine such as the AP1000. In §2 we describe two methods adopted in our experiments. §3 is concerned with different QR decomposition algorithms. The data communication required for parallel implementation of these algorithms on machines like the AP1000 is also discussed. In §4 we present experimental results and analyze these results to give some theoretical explanations. §5 gives the overall conclusions.

2 Mapping Matrices over the AP1000

The AP1000 [3, 5, 12] is a distributed memory MIMD machine with up to 1024 independent SPARC processors which are called *cells*. Each processor has a 128 KByte direct-mapped copy-back cache, 16 MByte of memory, and a Weitek floating-point unit (FPU) of theoretical peak speed 8.33 MFLOP (single precision) and 5.67 MFLOP (double precision). The topology of the AP1000 is a torus, with hardware support for wormhole routing. The routing hardware (T-net) provides a theoretical bandwidth of 25 MByte/sec between any two cells; in practice, about 6 MByte/sec is attainable by user programs. The routing hardware supports row and column broadcasts, so a broadcast usually takes about the same time as a comparable cell to cell transfer.

The local memory in each cell is accessible to other cells only via explicit message passing. Thus the distribution of data over the AP1000 is a very important issue for efficient solution of a given problem.

Various matrix mapping schemes have been discussed in [2, 4, 6]. Two of them, which are used in our experiments, are described here. The first one is the *scattered* representation. Assume that the cells form an $ncely \times ncelx$ grid with $ncely$ and $ncelx$ the number of rows and columns, respectively. Then (assuming C conventions, so indices run from 0) the elements $a_{i,j}$ of the given matrix A are stored in cell ($i \bmod ncely$, $j \bmod ncelx$). The matrix stored locally on each cell has the same shape as the global matrix A, so the computational load on each cell is approximately the same. We use this mapping scheme to implement unblocked algorithms on the AP1000.

To implement blocked algorithms, a *blocked panel-wrapped* representation is adopted. In this scheme an integer blocking factor r is chosen, the matrix A is partitioned into $r \times r$ blocks $A_{i,j}$, and the blocks $A_{i,j}$ are distributed as for our scattered representation, i.e., the block $A_{i,j}$ is stored on cell ($i \bmod ncely$, $j \bmod ncelx$). Like the elements $a_{i,j}$ in the scattered representation, the blocks $A_{i,j}$ are evenly distributed over the entire system, provided r is not too large.

In reporting our results we ignore the time required to send the data matrix A from the host processor to the cells. This is because, in most applications, the matrix A is already available in the cells, because it is known *a priori* or has been computed in earlier computations. Note that local disks are now available as an option on the AP1000, so data can be stored locally and transferred in parallel to the cells, and the host-cell communication link need no longer be a bottleneck.

3 Algorithms

In this section several algorithms for QR decomposition are described. Communication issues relating to implementation of the algorithms on distributed memory machines (such as the AP1000) with a two-dimensional grid or torus topology are also discussed.

3.1 Row and column Householder algorithms

The QR decomposition of an $m \times n$ matrix using Householder transformations takes n stages (assuming $m \geq n$). Each stage annihilates the subdiagonal elements of a column via two steps: the first step constructs a Householder vector v such that the first column of a sub-matrix is transformed into a multiple of the first coordinate vector e_1; the second step uses v to modify the remaining columns of the sub-matrix. The two steps may be written as

$$(I - 2\frac{vv^T}{v^Tv})A_{k-1} = \begin{pmatrix} \rho & s^T \\ 0 & A_k \end{pmatrix}$$

where the matrix $I - 2vv^T/v^Tv$ is a *Householder transformation* and the matrices A_j, of size $(m-j) \times (m-j)$, are the sub-matrices involved in the computation at the $(j+1)^{th}$ stage of the algorithm.

Application of a Householder transformation to a matrix A may be broken into the following two operations –

$$z^T = v^T A \text{ (vector-matrix multiplication)}$$

and

$$\hat{A} = A - \beta v z^T \text{ (rank-one modification)}$$

where $\beta = 2/v^Tv$. Thus, the Householder matrix $I - 2vv^T/v^Tv$ is never formed explicitly.

Let us consider the data communication required by a parallel implementation of the Householder algorithm on the AP1000. When constructing the Householder vector v in the first step of the k^{th} stage, we need to compute the 2-norm of the first column of the sub-matrix A_{k-1}. Since the matrix is distributed over the entire system, communication between cells in the leading column of the array is required. (Here the leading column of the array is a column of cells in which the first column of A_{k-1} resides. Similarly, a leading row of the array is a row of cells in which the first row of A_{k-1} resides.) Because operations to avoid overflow and to normalise the Householder vector v (so that $v_1 = 1$) are necessary in the implementation, communication in the vertical direction occurs three times for the construction of v at each stage.

To obtain the vector z^T in the second step, the Householder vector v is broadcast horizontally to all other columns of the array, local vector-matrix multiplications are performed in each cell, and then the partial results are accumulated in the corresponding cells in the leading row of the array (this is called the global summation procedure). To minimise the communication cost in the global summation, a minimum spanning tree communication pattern is adopted, that is, rows of the array cooperate pairwise to accumulate the partial results so that the final resulting vector is obtained in about $\log_2 ncely$ communication steps, where $ncely$ is the number of rows of the array. (For future reference, we note that in LU factorisation [2, 4] the communication pattern is simpler as the global summation procedure is not required.)

After z^T is obtained in the leading row of the array, it is broadcast vertically to all other rows of the array. The rank-one modification is then performed, without any further communication.

It is clear from the description above that in each stage all communications but one (the broadcast of v) take place in the vertical direction. Thus the communication is predominantly row oriented. Premultiplying a matrix by a Householder transformation is often called a row Householder update procedure [11], so we call the algorithm described above the *row Householder algorithm.* If the transpose of the matrix A is stored, the QR decomposition will require post-multiplication of Householder transformations, which is called the column Householder update procedure [11]. Thus, we call the corresponding algorithm the *column Householder algorithm.* It is easy to verify that the communication involved in parallel implementation of the column Householder algorithm is predominantly column oriented.

3.2 Block Householder algorithms

The standard Householder algorithm, as described above, is rich in matrix-vector multiplications. On many serial machines, including the AP1000 cells, it is impossible to achieve peak performance if the QR decomposition is performed via the standard Householder transformations. This is because matrix-vector products suffer from the need for one memory reference per multiply-add. The performance may be limited by memory accesses rather than by floating-point arithmetic. Closer to peak performance can be obtained for matrix-matrix multiplication [6, 7, 9].

The main idea of the block Householder QR method is as follows. The columns of a matrix to be decomposed are divided into blocks, each holding r columns. At each stage of the block algorithm the standard Householder algorithm is applied to the first block of the sub-matrix to generate r Householder transformations $H_i = I - 2v_iv_i^T/v_i^Tv_i$, $0 < i \leq r$. These Householder matrices are combined and represented in block (WY) form [11]:

$$H_1H_2\cdots H_r = I + WY^T$$

where W is an $m \times r$ matrix and Y is an $m \times r$ trapezoidal matrix. After this is accomplished the block Householder matrix is applied to update the rest of the sub-matrix. This updating procedure can be broken into two steps:

$$Z^T = W^TA \text{ (matrix-matrix multiplication)}$$

and

$$\hat{A} = A + YZ^T \text{ (rank-r modification).}$$

These steps are rich in matrix-matrix multiplications.

The total number of floating point operations in the block algorithm is larger than in the unblocked one, because extra computations are required to generate the matrix W in the block algorithm. Experimental results given in [7] show that the block algorithm is more efficient only if the size of block is much smaller than the size of the matrix to be decomposed.

There is another block representation of Householder matrices in which the product of Householder transformations is written in the form [16]:

$$H_1H_2\cdots H_r = I - YTY^T$$

where T is an $r \times r$ upper triangular matrix. The only advantage of this representation over the WY form is that it requires less storage for the matrix T. Because the block size r needs to be very small in order to obtain optimal performance on the AP1000 (see §4.3), the difference in storage requirements is not significant, so we use the simpler WY form.

The communication pattern for our implementation of the block Householder algorithm on the AP1000 is similar to that for our implementation of the standard Householder algorithm. However, in addition to extra floating point operations, extra communication is required to generate the matrix W (or T) if the cells are configured in a two-dimensional mesh. Thus, the block Householder algorithm involves a tradeoff between reduced memory access and increased communication. A similar tradeoff occurs in a blocked implementation of LU factorisation [2].

3.3 Givens and hybrid algorithms

The idea of the Givens algorithms is to apply a sequence of elementary plane rotations G_{ij} which are constructed to annihilate the ij^{th} element of a given matrix. A plane rotation is defined by a 2×2 orthogonal matrix of the form

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $c^2 + s^2 = 1$. If a $2 \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \end{pmatrix}$$

is premultiplied by G, a zero can be introduced into the a_{21} position.

Several parallel algorithms for pipelined implementation of Givens rotations have been introduced in the literature. However, most of the algorithms are designed for implementation on application-specific machines, e.g., those described in [1, 10]. Each time a plane rotation is applied, there is a requirement for data exchange between the corresponding two rows of the array. The high cost in data communication would significantly degrade the overall performance on a machine where a communication takes much longer than a floating-point operation. Thus, the pipelined Givens algorithms are not suitable for machines like the AP1000.

To avoid the high communication cost, a *greedy* scheme suggested in [14] can be adopted. Each stage of the algorithm takes two steps. In the internal elimination step of the k^{th} stage, each cell in the leading column of the array applies Givens rotations to introduce zeros in the subcolumn corresponding to the first column of the sub-matrix A_{k-1}, until only one nonzero element remains. The rotations are then broadcast horizontally to all other columns of the array to update the remaining columns of A_{k-1}. In the recursive elimination step the remaining subdiagonal elements (one in each cell of the leading column) are annihilated. To minimize the communication cost in this step, a minimum spanning tree communication pattern similar to that described in §3.1 can be adopted. However, the operations between each two vertical communication steps in the recursive elimination procedure involve a horizontal data broadcast and a plane rotation application, so the procedure is much more expensive than the global summation procedure in the Householder algorithm.

It is well known that the complexity of the Householder method in terms of floating point operations is roughly half that of the Givens method. The overall computational cost of the greedy Givens algorithm can be decreased if Householder transformations, instead of Givens rotations, are applied in the internal elimination step. This leads to the *hybrid* algorithm [15]. The hybrid algorithm should be more efficient than either the pipelined Givens or the greedy Givens algorithms on the AP1000.

cells	128					64				
aspect ratio	128	32	8	2	0.5	64	16	4	1	0.25
time (sec)	10.4	8.51	7.75	7.58	9.16	14.6	12.8	11.2	11.2	14.8

cells	32				16			
aspect ratio	32	8	2	0.5	16	4	1	0.25
time (sec)	26.1	23.7	23.5	25.1	48.9	48.7	47.3	52.1

cells	8			4			2	
aspect ratio	8	2	0.5	4	1	0.25	2	0.5
time (sec)	96.1	95.4	97.1	190	189	195	379	386

Table 1: Execution time of the column Householder algorithm for a 1000×1000 matrix.

4 Experimental Results and Discussion

Several algorithms for QR decomposition have been implemented on the 128-cell AP1000 located at the Australian National University. Many results have been obtained through extensive tests. In the following some of these experimental results are presented together with a theoretical discussion. In all cases the matrices are dense (our algorithms do not take advantage of sparsity) and the runtimes are independent of the input data (since the algorithms do not involve pivoting). For simplicity we only present results for the QR decomposition of square matrices and only count the time for the computation of the triangular factor R, not for accumulation of the orthogonal transformations. In all cases double-precision (64-bit) floating-point arithmetic is used.

4.1 Effect of array configuration

Suppose that the total number of cells is $P = ncely \times ncelx$. The aspect ratio is the ratio $ncely : ncelx$. For simplicity, we assume that P is a power of two; it follows that the aspect ratio is also a power of two. The AP1000 can be logically configured with any positive integer $ncely$ and $ncelx$ provided P does not exceed the maximum number of physical cells (128 for our machine). However, there is some loss of efficiency if $ncely > 16$ or $ncelx > 8$ since the logical configuration is simulated on a 16×8 hardware configuration.

We estimate the effect of array configuration by fixing the problem size and varying the aspect ratio of the array. In this experiment the unblocked column Householder algorithm is applied to factorise a dense 1000×1000 matrix. It can be seen from Table 1 that there is a decrease in the total computational time when the aspect ratio increases from less than one to one or two. This is due to the nature of the communication pattern of the algorithm. Since the communication required in the column Householder algorithm is column oriented, that is, the horizontal communication volume is higher than the vertical communication volume, the number of columns in the cell array should be decreased so that the overall communication cost may be reduced. However, Table 1 shows that further increasing the aspect ratio (even to P, that is, using a one dimensional array to eliminate all the horizontal communication), the performance gets worse. There are two reasons for this –

1. When the aspect ratio is increased, the height of the array is also increased, which results in a longer communication distance for data broadcast in the vertical direction. The longer the distance between the sending cell and the receiving cells, the more expensive is the communication, especially if the broadcast is performed using a binary tree rather

cells	128	64	32	16	8	4	2
hybrid	8.07	12.0	24.0	47.9	96.1	192	391
row Householder	8.09	12.0	23.8	47.6	95.8	190	391
column Householder	7.58	11.2	23.5	47.3	95.4	189	379

Table 2: Execution times of different algorithms for a 1000×1000 matrix.

than by the T-net hardware.

2. Since there are $ncelx$ columns in the array, the communication bandwidth for vertical broadcasts is (at most) $25ncelx$ MByte/sec. Similarly, the communication bandwidth for horizontal broadcasts is (at most) $25ncely$ MByte/sec. Thus, the time for v vertical and h horizontal broadcasts is roughly proportional to

$$T = \frac{v}{ncelx} + \frac{h}{ncely}.$$

If $P = ncelx \times ncely$ is fixed, T is minimised when the aspect ratio is h/v. Since the column Householder algorithm has $h > v$, we expect the optimal aspect ratio to be slightly (but not too much) greater than unity.

The results given in Table 1 show that the optimal ratio is either one or two.

4.2 The Householder method versus the hybrid method

As described in §3.3, the pipelined Givens and the greedy Givens algorithms are not as efficient as the hybrid algorithm on a two-dimensional distributed memory machine. We now consider the (row) hybrid algorithm. Since the unblocked row Householder method is applied for the internal elimination steps, we compare the results with those obtained using the pure unblocked row (and column) Householder algorithms.

Since the programs are written in the C programming language, matrices are stored in row-major order. Thus, access to a column in the row Householder algorithm will not be contiguous. This is why the performance of the row Householder algorithm is slightly worse than the performance of the column Householder algorithm, as shown by the last two rows of Table 2. If the programs were written in Fortran, which stores matrices in column-major order, the ranking would be reversed.

A theoretical complexity analysis of various QR decomposition algorithms on distributed memory machines [15] predicts that the hybrid algorithm should be more efficient than the Householder algorithm. However, our experimental results on the AP1000 do not agree with this conclusion. Table 2 shows that the two algorithms take about the same time to solve a 1000×1000 problem on the AP1000.

The recursive elimination procedure in the hybrid algorithm and the global summation in the Householder algorithm both act as synchronisation points during the computation, and this is one of the key factors in determining the active/idle ratio of cells in the system. As described in §3.3, the computation involved in a recursive elimination procedure is much more expensive than that in a simple global vector summation. The former will take a longer time to complete and thus decrease the active/idle ratio. Therefore, there is no obvious reason to choose the hybrid algorithm instead of the row or column Householder algorithm as the preferred algorithm for solving the QR decomposition problem on machines like the AP1000.

matrix size	256 × 256					
block width	1	2	4	8	16	32
time (sec)	.741	.994	.887	.887	1.02	1.38
matrix size	512 × 512					
block width	1	2	4	8	16	32
time (sec)	2.64	3.26	2.88	2.93	3.30	4.41
matrix size	1024 × 1024					
block width	1	2	4	8	16	32
time (sec)	12.7	14.4	13.1	13.3	14.5	18.0
matrix size	1536 × 1536					
block width	1	2	4	8	16	32
time (sec)	44.1	45.2	39.5	37.8	39.9	46.4
matrix size	2048 × 2048					
block width	1	2	4	8	16	32
time (sec)	115	116	103	82.3	84.5	95.0
matrix size	4096 × 4096					
block width	1	2	4	8	16	32
time (sec)	867	862	754	580	571	604

Table 3: Execution time of the block Householder algorithm on an 8×8 array.

4.3 Effect of the block width and matrix size

To measure the effect of block width on the performance, we have implemented the block WY Householder algorithm. In the experiment the number of cells involved in the computation is fixed at 64, i.e., an 8×8 array is used. To solve a problem of a given size, different block widths are applied, with experimental results as given in Table 3. Two points should be stressed. First, the block Householder algorithm is more efficient than the unblocked algorithm only if the size of the matrix to be decomposed on each cell is larger than the local cache size. Since the AP1000 has a 128 KByte cache, a 128×128 double-precision matrix will fit into the cache. Thus, on a configuration of $64 = 8 \times 8$ cells, a (distributed) 1024×1024 matrix will fit into the local caches on each cell. This explains why the unblocked algorithm is more efficient than the blocked algorithms for matrices of size 1024×1024 and smaller.

The second point is that the optimal block width is about 8 (or 16 for larger matrices). During the generation of the matrices W and Y in each stage of the block column Householder algorithm, only the cells in the leading row of the array are active; all other cells remain idle, requesting data from the corresponding cells in the leading row. The greater the block width, the longer the time for those cells to wait, and the lower the active/idle ratio. This is why the optimal block width is fairly small. (For simplicity, our Householder implementation uses the same block width in the algorithm as in the blocked panel-wrapped data distribution. Logically, these two parameters could be different. In our implementation [3] of LU factorisation we distributed data with block width 1, so we were able to use a larger blocking factor in the LU decomposition without decreasing the active/idle ratio.)

Table 4 gives the results obtained when the number of cells involved in the computation varies, but the size of the matrix is fixed at 1000×1000. It can be seen that the block algorithm is more efficient if the number of cells is less than 64 and that the optimal block width is about 8,

cells	4 (2×2)					
block width	1	2	4	8	16	32
time (sec)	189	172	143	130	131	143
cells	8 (4×2)					
block width	1	2	4	8	16	32
time (sec)	95.4	87.4	73.8	69.3	71.7	82.9
cells	16 (4×4)					
block width	1	2	4	8	16	32
time (sec)	47.3	45.4	39.1	36.9	38.6	45.2
cells	32 (8×4)					
block width	1	2	4	8	16	32
time (sec)	23.5	23.9	21.3	20.9	23.0	29.1
cells	64 (8×8)					
block width	1	2	4	8	16	32
time (sec)	11.2	13.2	12.1	12.2	13.8	17.9
cells	128 (16×8)					
block width	1	2	4	8	16	32
time (sec)	7.58	8.72	7.93	8.17	9.50	13.1

Table 4: Execution time of the block Householder algorithm for a 1000×1000 matrix using different numbers of cells

which is consistent with the observations above.

As described in §3.2, extra computations and communications are required to generate the matrix W in the block Householder algorithm. Thus the algorithm will not be as efficient as its unblocked counterpart when the size of a matrix to be decomposed is small. It is interesting to note that Tables 3 and 4 illustrate two different views about algorithms such as the block Householder algorithm on massively parallel supercomputers. In a pessimistic view, the advantage of using this kind of block algorithm to solve a problem of a given size diminishes as the number of cells and the cache size increase with advances in technology (Table 4). However, massively parallel supercomputers are intended to be used for solving large problems. In an optimistic view, this kind of block algorithm becomes advantageous as the problem size increases (Table 3).

4.4 Speedup and efficiency

Let t_P denote the time to execute a given job on P processors. The speedup S_P and the efficiency E_P for a system of P processors are defined as $S_P = t_1/t_P$ and $E_P = S_P/P$. Since Householder algorithms are preferred for the AP1000, the column Householder algorithm is applied to decompose a 1000×1000 matrix in measuring the speedup and the efficiency. The results are given in Table 5. The efficiency is close to 0.5 for $P = 128$ and higher for small P. This is similar to results for the Linpack benchmark [2], although the efficiency of the Householder algorithms is slightly (up to about 8%) lower. The results confirm the assertion [2] that the AP1000 is a good machine for the solution of dense numerical linear algebra problems such as the solution of linear equations, least squares, eigenvalue and singular value problems.

time for one cell	cells	time (sec)	speedup (S_P)	efficiency (E_P)
467	128	7.58	61.6	0.48
467	64	11.2	41.7	0.65
467	32	20.9	22.3	0.70
467	16	36.9	12.7	0.79
467	8	69.3	6.74	0.84
467	4	130	3.60	0.90
467	2	254	1.84	0.92

Table 5: Speedup and Efficiency of the Householder algorithm for a 1000×1000 matrix.

5 Conclusions

The results of our experiments and theoretical analysis can be summarised as follows:

- Of the various orthogonal factorisation methods considered, the Householder transformation techniques are the most effective for QR decomposition of a matrix on the AP1000.
- The aspect ratio of the array significantly affects the performance, especially when the total number of cells involved in the computation is large. Although communication in the column (or row) Householder algorithm is predominantly column (or row) oriented, the best aspect ratio of the array is either one or two.
- Because of extra computations and communications required in forming the matrix W (or T) in the block Householder algorithm, the block width needs to be small. The block algorithm is more efficient than its unblocked counterpart only if the size of a matrix to be decomposed is larger than 1024×1024 on the 128-cell AP1000.
- QR decomposition of a matrix is more complicated than LU decomposition in terms of both arithmetic operations and communications required for parallel implementation (see §3.1). Thus, it is not surprising that the efficiency achieved for QR decomposition on the AP1000 is slightly lower than for the Linpack Benchmark [2]. However, the degradation in efficiency is less than 10%. This supports claims that the AP1000 is a good machine for a variety of dense numerical linear algebra problems.

Acknowledgements

Support by Fujitsu Laboratories, Fujitsu Limited, and Fujitsu Australia Limited via the Fujitsu-ANU CAP Project is gratefully acknowledged. Thanks are due to Iain Macleod for helpful discussions and for assistance in improving the exposition.

References

[1] A. W. Bojanczyk, R. P. Brent and H. T. Kung, "Numerically stable solution of dense systems of linear equations using mesh-connected processors", *SIAM J. Sci. and Statist. Computing* 5 (1984), 95–104.

[2] R. P. Brent, "The Linpack Benchmark on the Fujitsu AP1000", *Proc. Frontiers '92* (McLean, Virginia, USA, October 1992), IEEE Press, 1992, 128–135.

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

[4] R. P. Brent, "Parallel algorithms in linear algebra", *Proc. Second NEC Research Symposium* (Tsukuba, Japan, August 1991), invited paper, to appear. Also Report TR-CS-91-06, Computer Sciences Laboratory, ANU, August 1991, 17 pp. Available by anonymous ftp from `dcssoft.anu.edu.au` in the directory `pub/Brent` .

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

[6] R. P. Brent and P. E. Strazdins, "Implementation of the BLAS level 3 and Linpack benchmark on the AP1000", *Fujitsu Sci. Tech. J.* 29, 1 (March 1993), 61–70.

[7] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1990.

[8] J. J. Dongarra, A. H. Sameh and Y. Robert, "Implementation of some concurrent algorithms for matrix factorization", *Parallel Computing* 3 (1986), 25–34.

[9] K. A. Gallivan, W. Jalby and U. Meier, "The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory", *SIAM J. Sci. and Statist. Computing* 8 (1987), 1079–1084.

[10] W. M. Gentleman and H. T. Kung, "Matrix triangularization by systolic arrays", *Proc. SPIE, Volume 298, Real-Time Signal Processing IV*, Society of Photo-Optical Instrumentation Engineers, 1981, 19–26.

[11] G. H. Golub and C. Van Loan, *Matrix Computations*, second edition, Johns Hopkins Press, Baltimore, Maryland, 1989.

[12] H. Ishihata, T. Horie and T. Shimizu, "An architecture for the AP1000 highly parallel computer", *Fujitsu Sci. Tech. J.* 29, 1 (March 1993), 6–14.

[13] S. Kim, D. Agrawal and R. Plemmons, *Recursive least squares filtering for signal processing on distributed memory multiprocessors*, Tech. Rep., Dept. of Computer Science, North Carolina State University, Raleigh, NC, 1988.

[14] J. J. Modi and M. R. B. Clarke, "An alternative Givens ordering", *Numer. Math.* 43 (1984), 83–90.

[15] A. Pothen and P. Raghavan, "Distributed orthogonal factorization: Givens and Householder algorithms", *SIAM J. Sci. Statist. Computing* 10 (1989), 1113–1134.

[16] R. Schreiber and C. Van Loan, "A storage efficient WY representation for products of Householder transformations", *SIAM J. Sci. and Statist. Computing* 10 (1989), 53–57.