# Sparse Givens QR Factorization on a Multiprocessor

J. Tourino
R. Doallo
E.L. Zapata

May 1996
Technical Report No: UMA-DAC-96/08

Published in:

*2nd Int'l. Conf. on Massively Parallel Computing Systems
Ischia, Italy, May 1996, pp. 551-556*

# Sparse Givens QR Factorization on a Multiprocessor[*]

Juan Touriño, Ramón Doallo
Dept. Electrónica y Sistemas
University of La Coruña
Campus de Elviña s/n
15071 La Coruña, Spain
{juan,doallo}@udc.es

Emilio L. Zapata
Dept. Arquitectura de Computadores
University of Málaga
Campus de Teatinos
29071 Málaga, Spain
ezapata@atc.ctima.uma.es

## Abstract

*We present a parallel algorithm for the QR factorization with column pivoting of a sparse matrix by means of Givens rotations. Nonzero elements of the matrix* M *to be decomposed are stored in a one-dimensional doubly linked list data structure. We will discuss a strategy to reduce fill-in in order to gain memory savings and decrease the computation times. As an application of QR factorization, we will describe the least squares problem. This algorithm has been designed for a message passing multiprocessor and we have evaluated it on the Cray T3D supercomputer using the Harwell-Boeing sparse matrix collection.*

## 1. Introduction

QR factorization is a direct method in matrix algebra which involves the decomposition of a matrix $M$ of dimensions $A \times B$ ($A \geq B$) into the product of an orthogonal matrix $Q$ ($Q^T = Q^{-1}$) and an upper triangular matrix $R$. QR factorization has many applications in numerical linear algebra to solve linear systems of equations, least squares problems, linear programs, eigenvalue problems, coordinate transformations, projections and optimization problems. It is necessary to solve these problems in many scientific fields, such as fluid dynamics, molecular chemistry, aeronautic simulation ...

This factorization can be computed by several possible ways [6, Chapter 5]: using plane rotations (Givens method), the Modified Gram-Schmidt procedure or Householder reflections. Since these sequential algorithms have a high arithmetic complexity, the development of parallel algorithms is of considerable interest. Several parallel orthogonal factorization algorithms have been designed for various machines. We cite just a few: [3] for the *Intel iPSC/1*, [5] for the *nCUBE 10*, [9] for a network of transputers, [1] for the *nCUBE 2*, [2] for the *CM-200*, all of them for dense matrices; and [14] (*CM-2*), [13] (*Fujitsu AP1000*), [12] (*Cray T3D*) for sparse matrices.

We have implemented the Givens method with column pivoting for sparse matrices on the *Cray T3D MIMD* distributed memory computer. Although a sparse problem could be treated with a parallel program for dense factorization, the storage and time cost of ignoring sparsity would not benefit from parallel processing.

This paper is organized as follows. In §2 we describe the sequential and parallel Givens algorithm, as well as a strategy to reduce fill-in. The least squares problem, an application of QR factorization, is shown in §3 and experimental results are discussed in §4.

## 2. QR through Givens rotations

We obtain matrices $Q$ and $R$ of dimensions $A \times A$ and $B \times B$, respectively, using Givens rotations. We do not calculate matrix $Q$ because this matrix is not explicitly necessary in order to solve the least squares problem, described in §3. Matrix $M$ is overwritten by matrix $R$ (in-place algorithm). We present the sequential algorithm with column pivoting in order to consider those cases in which the rank of matrix $M$ is not maximum. The use of orthogonal transformations is numerically stable and, in practice, the rank of the matrix can be determined accurately when column permutations are performed during factorization.

[*]This work was supported by the Spanish CICYT under contract TIC92-0942-C03 and by the TRACS Programme under the Human Capital and Mobility Programme of the European Union (grant number ERB-CHGE-CT92-0005)

```
rank = B;
for (j=0; j<B; j++)
    $norm_j = \sum_{i=0}^{A-1} m_{ij}^2;$                              (1)

for (cx=0; cx<B; cx++) {
    Obtain $px$, $cx \leq px < B$, such that
        $norm_{px} = \max_{cx \leq j < B} norm_j;$                      (2)
    if ($norm_{px} < \phi$) {                                          (3)
      $rank = cx;$
      $cx = B$ ;
    }
    else {
      swap ($norm_{cx}$, $col_{cx}$ of $M$) and
        ($norm_{px}$, $col_{px}$ of $M$);                               (4)

      for (i=A-1; i>cx; i--)
        Apply Givens rotation to subrows
            i-1 and i from $col_{cx}$ to $col_{B-1}$;                   (5)

      for (j=cx+1; j<B; j++)
        $norm_j = norm_j - m_{cx\,j}^2;$                                (6)
    }
}
```

In (1) the squares of the euclidean norms of the columns of matrix $M$ are calculated and stored in vector *norm*. Then, a procedure of $B$ iterations (if the rank of $M$ is maximum) is performed. It consists of the following actions: the pivot column ($px$) and the pivot element ($norm_{px}$) are selected (2). The pivot element is the maximum of the norms of the columns whose index is $\geq cx$. If *pivot* is close to *0* ($\phi$ is the required precision), the rank of the matrix is given by the value $cx$ and the factorization ends (3). Otherwise, a swap of column $cx$ with the pivot column of matrix $M$, as well as a swap of their norms, are performed (4).

Givens rotations are applied in (5) to zero the subdiagonal elements of column $cx$. A Givens rotation involving two rows $\alpha$, $\beta$ of matrix $M$ consists of calculating the following product:

$$\begin{pmatrix} m'_{\alpha\, cx} & m'_{\alpha\, cx+1} & \cdots & m'_{\alpha\, B-1} \\ 0 & m'_{\beta\, cx+1} & \cdots & m'_{\beta\, B-1} \end{pmatrix}$$

$$\uparrow \tag{7}$$

$$\begin{pmatrix} gcos & -gsin \\ gsin & gcos \end{pmatrix} \cdot \begin{pmatrix} m_{\alpha\, cx} & m_{\alpha\, cx+1} & \cdots & m_{\alpha\, B-1} \\ m_{\beta\, cx} & m_{\beta\, cx+1} & \cdots & m_{\beta\, B-1} \end{pmatrix}$$

where $gcos = \frac{m_{\alpha\, cx}}{\sqrt{m_{\alpha\, cx}^2 + m_{\beta\, cx}^2}}$, $gsin = \frac{-m_{\beta\, cx}}{\sqrt{m_{\alpha\, cx}^2 + m_{\beta\, cx}^2}}$

Givens rotations are clearly orthogonal and it is not necessary to perform inverse trigonometric functions. Figure 1 shows the sequential QR factorization by means of Givens rotations. It can be observed how zeros are introduced in matrix $M$ to achieve the upper triangular matrix $R$. Finally, the norms of the updated columns are calculated in (6).

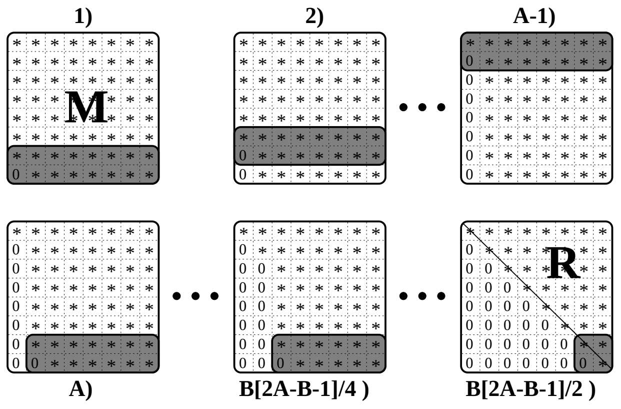

**Figure 1. Sequential Givens rotations**

Once the algorithm has ended, what we really get is a $M \times \Pi = Q \times R$ factorization. $\Pi$ is a permutation $B \times B$ made up by the product of *rank* elementary permutations, $\Pi = \pi_0 \times \pi_1 \times \ldots \times \pi_{rank-1}$, each $\pi_j$, with *j=0,...,rank-1*, being the identity matrix or a matrix resulting from swapping two of its columns. This is due to the pivoting carried out in (2).

## 2.1. Fill-in control

When working with sparse matrices an additional problem arises, which is the fill-in. The factors that influence the fill-in of a sparse matrix are the following: the dimensions and rank of the matrix, the sparsity degree (number of null elements) and another factor of great importance, but difficult to model, which is the matrix pattern or the location of the nonzero elements. Thus, fill-in may vary significantly for two matrices with the same dimensions, rank and sparsity degree, depending on how nonzero elements are placed. A high fill-in is an undesirable situation due to the increase in the storage cost and computation time. It would be of interest to implement a simple method to reduce fill-in. The most common heuristic strategy employed in LU factorization to maintain the sparsity degree is the Markowitz criterion [7, Chapter 7]. Moreover, numerical stability must be ensured in the LU factorization, by avoiding the selection of pivots with a low absolute value. A row ordering strategy for Givens

| *Matrix* | *Origin* | $A \times B$ | *Elem(M)* | *% Elem(M)* |
|---|---|---|---|---|
| JPWH991 | Circuit physics modelling | $991 \times 991$ | 6027 | 0.61% |
| BCSSTK14 | Structural engineering | $1806 \times 1806$ | 32630 | 1.00% |
| SHERMAN5 | Oil reservoir modelling | $3312 \times 3312$ | 20793 | 0.19% |

**Table 1. Harwell-Boeing sparse matrices**

| *Matrix* | $\epsilon = 0$ | | $\epsilon \approx 1$ | | *%Reduc.* |
|---|---|---|---|---|---|
| | *Elem(R)* | *%Elem(R)* | *Elem(R)* | *%Elem(R)* | |
| JPWH991 | 308149 | 62.69% | 140610 | 28.61% | 54.37% |
| BCSSTK14 | 1224979 | 75.07% | 254841 | 15.62% | 79.20% |
| SHERMAN5 | 481439 | 8.78% | 377134 | 6.87% | 21.67% |

**Table 2. Fill-in reduction**

rotations based on pairing rows to minimize fill-in is presented in [15].

We have implemented a method to reduce fill-in in the QR factorization by taking advantage of column pivoting. Instead of expression (2), we use a new criterion to select the pivot column:

$$\text{Obtain } px,\ cx \leq px < B, \text{ such that}$$
$$\left\{ \epsilon \left( \frac{zero_{px}}{\max\limits_{cx \leq j < B} zero_j} \right) + (1-\epsilon) \left( \frac{norm_{px}}{\max\limits_{cx \leq j < B} norm_j} \right) \right\} \quad (8)$$
$$\text{is maximum}$$

where $zero_j$ is the number of nonzero elements in subcolumn $j$ (from row $cx$ to row *A-1*), and $\epsilon$, $0 \leq \epsilon \leq 1$, is a prefixed parameter. The first term refers to fill-in reduction, while the second one relates to numerical stability. Obviously, for $\epsilon = 0$, the pivot column selection criterion is equivalent to the one described in (2). The strategy is to choose a pivot column with many zeroes in the subcolumn from row $cx$ to *A-1* in order to perform fewer rotations. Although we try to reduce fill-in as much as possible ($\epsilon \approx 1$), we shall always keep a minimum degree of numerical stability in the algorithm by discarding as pivot columns those with norm close to zero.

With the aim of testing this strategy, we have chosen three matrices from the Harwell-Boeing sparse matrix collection [8]. A description of these matrices is presented in table 1, where $A \times B$ are the dimensions of the matrix, *Elem(M)* is the number of nonzero elements of $M$ and *%Elem(M)* is the percentage of these elements. Table 2 shows fill-in in matrix $R$ after the factorization; *Elem(R)* and *%Elem(R)* are the number and percentage of nonzero elements, for $\epsilon = 0$ and $\epsilon \approx 1$ in expression (8); *%Reduc.* is the percentage of reduction in the number of nonzero elements obtained with $\epsilon \approx 1$ ($\epsilon = 0.999$). As we can see, fill-in has decreased by 50% on the average for this set of sparse matrices, which is a substantial reduction.

## 2.2. Parallel Givens Rotations

The parallel algorithm developed has been generalized for any number of processing elements (PEs) and any dimension of matrix *M*. We find parallel Givens algorithms in [2] and [14].

Matrix *M* is distributed onto a mesh with $m \times n$ PEs. Each PE is identified by coordinates *(idx,idy)*, with $0 \leq idx < n$ and $0 \leq idy < m$. Nonzero elements of *M* are mapped over PEs using a Block Column Scatter (BCS) scheme [10], but these elements are stored in doubly linked lists instead of vectors. This distribution provides data and load balancing. The algorithm requires access both by rows and by columns; a data structure such as a two-dimensional doubly linked list (used in [11] for a LU factorization) would be suitable. But this structure is costly to manage and a great amount of memory is required. Therefore, we use one-dimensional doubly linked lists; each one represents one column of the matrix and each item of the list stores the row index, the matrix element and two pointers. These lists are arranged in growing order of the row index and they provide efficient access by columns. Row access is achieved using an auxiliary pointer vector with as many components as columns in the matrix. We go with this pointer vector through the linked lists corresponding to the columns of the matrix from bottom to top to get row access.

Let us consider the sequential algorithm to see how it can be executed in parallel. First, each PE obtains the local norms corresponding to the column segments (local lists) it contains. By means of a reduction instruction (sum by columns), the vector *norm* of each column of PEs will contain the norms of the corres-

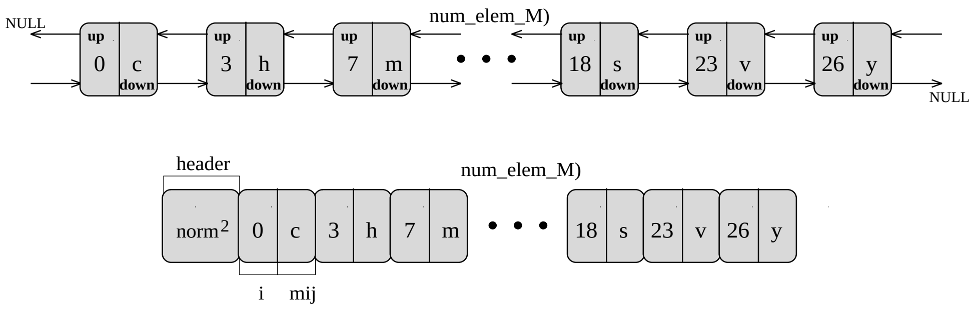

**Figure 2. Buffer for sending local lists**

ponding global columns (1). Then, the local maximum norm of each PE is obtained (this value is the same for each column of PEs). The global maximum is obtained by means of a reduction instruction which finds the maximum norm by rows of PEs. As a result, the pivot element, as well as $px$ (the index of the pivot column (2)), will be contained in all the PEs. The parallelization of the strategy to reduce fill-in (8) requires more communications than (2), but it is not very costly from the computational point of view. In order to perform the pivoting described in (4), if columns $cx$ and $px$ are located in different PEs, we use a packed vector [7, Chapter 2] that acts as a buffer for exchanging data. As figure 2 shows, information of the lists is placed in consecutive memory positions; the corresponding column of $M$ and the square of its norm are sent in a single message.

If the rotations are parallelized according to the sequential algorithm, neither the outer loop, which goes through the columns of the matrix, nor the inner loop (5), which annihilates each element of the column, could be executed in parallel due to data dependencies. Thus, the parallel algorithm would need B(2A-B-1)/2 iterations. Nevertheless, a Givens rotation can be applied to any two rows, not necessarily consecutive; therefore, each row of PEs can, independently and in parallel, apply the rotations to the $M$ rows which they store and, thus, reduce the execution times. Figure 3a illustrates how this is done for a matrix *16*$\times$*4* distributed on a mesh *4*$\times$*1*. In this example, the four PEs apply Givens rotations in parallel to the first column of the matrix, but the first row of each PE is not rotated. This is solved as shown in figure 3b. The PE containing the row $cx$ (in this example, row 0) sends it to the southern PE *(i)*, and then a Givens rotation is applied to row $cx$ and to the non-rotated row of this PE in order to zero the corresponding element. The new row $cx$ is sent again to the southern PE *(iii)* and we proceed in the same way as in *(ii)*. Once all the rows are rotated, row $cx$ is updated *(vii)*; the final result is shown in 3c. We have chosen this approach because it is easy to generalize for any dimension of the mesh.

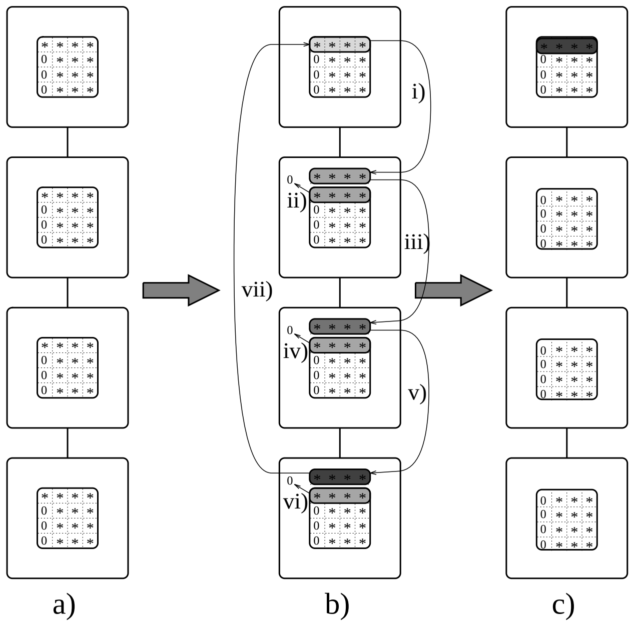

**Figure 3. Parallel Givens Rotations**

Rows are rotated to zero the corresponding elements. Clearly, it is not necessary to rotate those rows whose first element is zero. As our matrix is sparse and null elements are not stored in the lists, we go through the elements of list (column) $cx$ and rotate the rows of those elements; fill-in may appear at this stage.

## 3. The least squares problem

The least squares problem calculates a vector $x$ of length $B$ that minimizes $||Mx - z||_2$, where $z$ is a vector of length $A$. If the rank of $M$ is maximum ($B$), the least squares problem has one unique solution ($x_{LS}$). Otherwise, it has an infinite number of solutions $x_{SOL}$, one of which has a minimum norm and which we will also denote as $x_{LS}$, $x_{LS} = x_{SOL}$ *such that* $||x_{SOL}||_2$ *is minimum.* If $A$=$B$, the least squares problem is equivalent to solving a linear equation system $Mx = z$, since $||Mx - z||_2 = 0$.

This problem can be solved adapting the parallel algorithm that carries out the Givens QR factorization of matrix $M$. In particular, the least squares problem is equivalent to solving the upper triangular system $R\Pi^T x = Q^T z$. This approach is adequate due to the good numerical stability of the QR factorization. If $rank(M) = B$, this algorithm calculates the one unique solution to the least squares problem. If $rank(M) < B$, only one of the infinite solutions is obtained, the one called basic solution, which has a maximum of rank nonzero elements and that, in general, will not coincide with the minimum norm solution $x_{LS}$.

### 3.1. Calculation of $Q^T z$

Product $Q^T z$ is obtained at the same time that QR factorization is performed (we assume vector $z$ is dense). First, vector $z$ is stored in a vector called *qtz*, which is distributed in each column of PEs, so that the global component $I$ of *qtz* is replicated in the row of PEs with *idy=I mod m.* Then, Givens rotations are applied to the corresponding elements of *qtz* at the same time we apply the rotations to the rows of matrix $M$. For instance, if rows $\alpha$ and $\beta$ are being processed, this product is calculated (it is similar to (7)):

$$\begin{pmatrix} qtz'_\alpha \\ qtz'_\beta \end{pmatrix} \leftarrow \begin{pmatrix} gcos & -gsin \\ gsin & gcos \end{pmatrix} \cdot \begin{pmatrix} qtz_\alpha \\ qtz_\beta \end{pmatrix} \quad (9)$$

Once all the rotations of the factorization have ended, vector *qtz* stores the product $Q^T z$ from index *0* to the global index *B-1*. As we can see, matrix $Q$ is not necessary.

### 3.2. Back-substitution and permutation

The upper triangular system $Rx = Q^T z$ is solved by means of a back-substitution. The corresponding sequential algorithm is as follows:

```
for (i=rank-1;i≥0;i--)
    $x_i = (qtz_i - \sum_{j=i+1}^{rank-1} r_{ij} \cdot x_j)/r_{ii};$        (10)
```

This loop has data dependencies, and thus it must be maintained in parallel code without any possibility of being distributed among the PEs. In addition, it is necessary to access the elements of matrix $R$ by rows (matrix $R$ is stored by columns). This is solved by using an auxiliary pointer vector as we saw in §2.2. Another option is to apply the column version of back-substitution [6, Chapter 3]. Once the back-substitution is carried out we get the solution vector $x$ of length B distributed in each row of PEs, so that the global component $J$ of vector $x$ is replicated in the column of PEs with *idx=J mod n*.

Due to the column pivoting carried out in the QR factorization, $\Pi$ permutation must be applied to the components of vector $x$, so that $x$ is overwritten with vector $\Pi x$. All the PEs contain a vector called *permut* of length $B$. It is the only vector whose components are not distributed among the PEs. This vector stores the index of the column swapped in each iteration (*px*) and, by applying these swaps starting from the end, elements of vector $x$ are obtained in the correct order.

## 4. Experimental results and conclusions

The algorithm has been implemented on a *Cray T3D* supercomputer [4] with a *Cray Y-MP* host and 320 DEC-Alpha processors connected by a tridimensional torus topology, using *C* language and *PVM* routines for message passsing. The parallel code is *SPMD* (Simple Program Multiple Data). We have used low latency communication functions, such as *pvm_fastsend* and *pvm_fastrecv* (non-standard PVM functions) for messages of length less than 256 bytes. We have also developed reduction instructions suitable for our algorithm to reduce communications.

We have tested the performance of our parallel algorithm using the Harwell-Boeing matrices described in table 1. Table 3 shows the execution times (in seconds) for 1, 4, 16, 64 and 128 PEs without taking into account our strategy to reduce fill-in ($\epsilon = 0$) and applying the fill-in control approach ($\epsilon \approx 1$). These times include the QR factorization as well as the resolution of the least squares problem. The time required for data distribution and collection of results is not included because we assume that this program is a possible subproblem within a wider program.

Execution times are substantially reduced with $\epsilon \approx 1$ because fewer nonzero elements appear and, therefore, computations savings are achieved. For instance, execution times decrease for matrices *JPWH991*, *BCSSTK14* and *SHERMAN5* using 64 PEs by 60%, 87% and 67%, respectively. These times decrease even further as the number of PEs increases,

| *Matrix* | *1* | | *4* | | *16* | | *64* | | *128* | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\epsilon = 0$ | $\epsilon \approx 1$ | $\epsilon = 0$ | $\epsilon \approx 1$ | $\epsilon = 0$ | $\epsilon \approx 1$ | $\epsilon = 0$ | $\epsilon \approx 1$ | $\epsilon = 0$ | $\epsilon \approx 1$ |
| JPWH991 | 614.55 | 180.85 | 208.72 | 63.73 | 68.13 | 23.31 | 24.56 | 9.79 | 18.23 | 10.32 |
| BCSSTK14 | 5724.35 | 326.60 | 1593.82 | 101.73 | 466.84 | 34.01 | 129.50 | 17.25 | 79.21 | 17.08 |
| SHERMAN5 | 5308.59 | 815.49 | 1370.68 | 292.32 | 389.38 | 107.50 | 139.07 | 45.24 | 93.17 | 40.21 |

**Table 3. MGS: execution times (in seconds) for $\epsilon = 0$ and $\epsilon \approx 1$**

due to the fact that, when data are distributed among more PEs, the number of computations that each PE carries out is lower, whereas the number of communications tends to increase. Since many communications are required to update the non-rotated row of each PE in each step of the algorithm (see figure 3b), even the running time is higher using 128 PEs than with 64 PEs for matrix *JPWH991*.

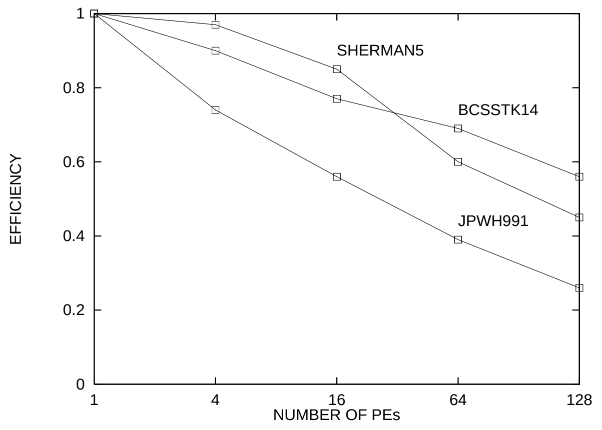

**Figure 4. Efficiencies for $\epsilon = 0$**

Figure 4 shows the efficiencies for $\epsilon = 0$ attained for each matrix. For example, the efficiencies for *JPWH991, BCSSTK14* and *SHERMAN5* using 128 PEs are 0.26, 0.56 and 0.45, respectively. As we can see, the algorithm scales rather well. It is clear that efficiencies will be lower for $\epsilon \approx 1$ because the execution of the algorithm with fill-in reduction has lower running times and therefore the communication term is a relatively more significant fraction of the running time. Nevertheless, better efficiencies could be achieved with larger matrices.

## References

[1] B.Hendrickson. Parallel QR Factorization using the Torus-wrap Mapping. *Parallel Computing*, 19:1259–1271, 1993.

[2] C.Bendtsen, P.C.Hansen, K.Madsen, H.B.Nielsen, and M.Pinar. Implementation of QR Up and Downdating on a Massively Parallel Computer. *Parallel Computing*, 21:49–61, 1995.

[3] C.H.Bischof. Adaptive Blocking in the QR Factorization. *The Journal of Supercomputing*, 3:193–208, 1989.

[4] Cray Research, Inc. *Cray T3D. Technical Summary*, September 1993.

[5] E.L.Zapata, J.A.Lamas, F.F.Rivera, and O.G.Plata. Modified Gram-Schmidt QR Factorization on Hypercube SIMD computers. *Journal of Parallel and Distributed Computing*, 12:60–69, 1991.

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

[7] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, 1986.

[8] I.S.Duff, R.G.Grimes, and J.G.Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR-PA-92-96, CERFACS, October 1992.

[9] L.C.Waring and M.Clint. Parallel Gram-Schmidt Orthogonalisation on a Network of Transputers. *Parallel Computing*, 17:1043–1050, 1991.

[10] L.F.Romero and E.L.Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *Parallel Computing*, 21(4):583–605, 1995.

[11] R.Asenjo and E.L.Zapata. Sparse LU Factorization on the Cray T3D. In *Int'l Conference on High-Performance Computing and Networking, Milan, Italy*, pages 690–696. Springer-Verlag LNCS no.919, May 1995.

[12] R.Doallo, B.B.Fraguela, J.Touriño, and E.L.Zapata. Parallel Sparse Modified Gram-Schmidt QR Decomposition. In *Int'l Conference on High-Performance Computing and Networking (accepted), Brussels, Belgium*, April 1996.

[13] R.Doallo, J.Touriño, and E.L.Zapata. Sparse Householder QR Factorization on a Mesh. In *Fourth Euromicro Workshop on Parallel and Distributed Processing, Braga, Portugal*, pages 33–39. IEEE Computer Society Press, January 1996.

[14] S.G.Kratzer. Sparse QR Factorization on a Massively Parallel Computer. *The Journal of Supercomputing*, 6:237–255, 1992.

[15] T.H.Robey and D.L.Sulsky. Row Ordering for a Sparse QR Decomposition. *SIAM J. Matrix Anal. Appl.*, 15(4):1208–1225, October 1994.