Sparse Householder QR Factorization on a Mesh

Ramón Doallo, Juan Touriño

Dept. Electrónica y Sistemas
Universidad de La Coruña
Campus de Elviña s/n
15071 La Coruña, Spain
E-mail: {esdoallo,juan}@udc.es

Emilio L. Zapata

Dept. Arquitectura de Computadores
Universidad de Málaga
Plaza El Ejido s/n
29013 Málaga, Spain
E-mail: ezapata@atc.ctima.uma.es

Abstract

In this document we are going to analyze the parallelization of QR factorization by means of Householder transformations. This parallelization will be carried out on a machine with a mesh topology (a 2-D torus to be more precise). We use a cyclic distribution of the elements of the sparse matrix M we want to decompose over the processors. Each processor represents the nonzero elements of its part of the matrix by a one-dimensional doubly linked list data structure. Then, we describe the different procedures that constitute the parallel algorithm. As an application of QR factorization, we concentrate on the least squares problem and finally we present a evaluation of the efficiency of this algorithm for a set of test matrices from the Harwell-Boeing sparse matrix collection.

1 Introduction

In many scientific fields it is necessary to solve large systems of linear equations: fluid dynamics, molecular chemistry, aeronautic simulation... In many cases, these systems are sparse, feature we can use in order to reduce the computation time and the memory necessary for the solution of these problems. We can use iterative methods [3], such as the conjugate gradient algorithm or direct methods [7], mainly based on Gaussian elimination. In this document we consider a direct method based on orthogonalization: QR decomposition. It basically consists in the decomposition of a matrix M of dimensions $A * B$ (with $A \geq B$) into the product of two matrices: $Q * R$, where Q is an orthogonal matrix (that is, the columns of Q are orthogonal and of unit length; consequently, $Q^T * Q = I$, that is, $Q^T = Q^{-1}$) and R is upper triangular.

LU decomposition is more widely employed for the solution of sparse linear equation systems because it is less costly in computation time and memory despite the better numerical stability of QR decomposition. However, the main use of QR decomposition is in the various applications it has in linear algebra, such as eigenvalue calculation and the least squares problem. There are several algorithms for finding this decomposition: Householder reflections or transformations, Givens rotations and Modified Gram-Schmidt algorithm. Different results obtained for dense matrices can be found in [4, 18, 16, 17] on distributed-memory machines; or for sparse matrices in [15, 13, 9, 14].

The algorithm we will implement is the one based on Householder transformations [10, chapter 5] with column pivoting in order to contemplate those cases in which the rank of matrix M is not maximum (B). Pivoting will also be used in order to provide numerical stability. This algorithm obtains a matrix Q of dimensions $A * A$ and a matrix R of dimensions $A * B$, in which the elements of the last A-B rows of this matrix are zero.

The parallel algorithm has been programmed for Fujitsu's *AP1000* MIMD distributed-memory computer and previously debugged using the software simulator called CASIM [6]. This supercomputer has a 2-D torus topology. It consists of 64 to 1024 processing elements or cells (*SPARC* processors) and three independent communication networks: the *torus network (T-net)* for point to point communications between cells; the *broadcast network (B-net)* for 1 to N communications between the host and the cells as well as for the distribution and recollection of the data; and the *synchronization network (S-net)* for the barrier synchronization. For more details on this architecture consult [12]. To all this we must add a specific instruction set [1] for the distribution and recollection of the data (very adequate for working with dense matrices) and reduction instructions for obtaining sums, maxima and minima of the elements in different cells. We are going to assume, in the notation, a general configuration of $m * n$ processing elements or cells. Each cell is going to be identified by coordinates *(idx,idy)*, with $0 \leq idx < n$ and $0 \leq idy < m$; m identifies the number of cells in the Y axis and n the number of cells in the X axis.

2 Data storage structure

In general, we will say that a matrix is sparse if it is advantageous to exploit the null elements. As in the sparse QR decomposition problem the structure of the matrices varies (due to the *fill-in*), it is convenient to use dynamic structures for the storage of the data in the cells that support the changes in the disposition of the nonzero elements. The main factor we have taken into account in the selection of the data storage structure has been the nature of the algorithm. Thus, as we will later see, we are only going to need column access to the matrix (except in the least squares problem). Consequently, we have chosen to use doubly linked lists, each one of which represents a column of the matrix. Their structure is given by the following code:

```
typedef struct item {
  index i;
  type mij;
  struct item *up;
  struct item *down; } item;
```

where *index* is the data type of the row index i of the matrix element (an integer, for example) and *type* is the data type of the element of the matrix. This list is arranged in growing order of the index i. We also need two pointers per list, one at the beginning and another one at the end:

```
item *startcr[MAXSIZE];
item *lastcr[MAXSIZE];
```

where *MAXSIZE* is a constant that represents the maximum number of local rows or columns.

An important fact is that we only require efficient access by columns in this algorithm; this implies large memory savings. Thus, in a LU decomposition for sparse matrices [5], we would need a data structure that facilitated access both by rows and by columns, such as a two-dimensional doubly linked list, and in order to do this we would need to store, in addition to what we have indicated, the j index of each element and two additional pointers (one to the previous row element and another to the next). On the other hand, the time needed for managing this structure would increase.

The parallel algorithm uses several routines for insertion and deletion in the lists in order to optimize their handling. As an example, there is an *insertion* procedure that contemplates an insertion at the end of the list or in the middle, and as a complement there is another procedure *fast_insertion* that only takes into account insertions at the beginning of the list. The division of insertion into two procedures is due to the fact that in certain steps of the algorithm we are only going to need insertions at the beginning of the lists and this way computation times can be reduced.

3 The parallel algorithm

The parallel algorithm developed has been generalized for any dimension of the mesh and any dimension of matrix M, so that the execution for a single processor is going to be equivalent to the sequential algorithm.

3.1 Data distribution

We have chosen a cyclic distribution of the data, also known as *grid* distribution and *scattered square* decomposition. Element (I,J) of the original matrix M is located in a cell $(idx,idy)=(J \; mod \; n, I \; mod \; m)$. The original matrix M is distributed among the local matrices (lists) and, after executing the corresponding parallel algorithm, we will obtain a piece of the matrix R in each one of them. It is therefore an *in-place* algorithm. From the global indices (I,J) that identify the elements of the matrix, we can obtain the local indices (i,j) in each cell: $i=\lfloor \frac{I}{m} \rfloor$, $j=\lfloor \frac{J}{n} \rfloor$. If the dimension of matrix M is $A*B$, the dimension of the local matrices once the distribution has been carried out would be $a*b$ (the dimension will vary depending on the cell we consider): $a = \lfloor \frac{A}{m} \rfloor + idy < A \; mod \; m$; $b= \lfloor \frac{B}{n} \rfloor + idx < B \; mod \; n$. The logical expressions $idy < A \; mod \; m$, $idx < B \; mod \; n$ will return one if they are fulfilled and zero otherwise. In a similar way, we can reconstruct the global indices of the matrix from the local indices: $I = (i*m) + idy, J = (j*n) + idx$. This is necessary for the result recollection stage (matrix R and the solution vector of the least squares problem). The selection of this distribution responds to two reasons: data balancing and load balancing.

So, this is a problem of reducing the index space. The algorithm is made up of as many steps as columns in the original matrix (unless the rank is smaller). Let us assume that in a given moment we are in step k of the algorithm. Then, as we will later see, elements (α,μ) of matrix R, will be updated with $k \leq \alpha < A$; $k \leq \mu < B$. It is clear that if we employed a consecutive distribution of the data, as the algorithm was executed, a large number of the cells would be inactive because the elements of the matrix that have to be updated would be concentrated in a few cells. This is prevented using a cyclic distribution.

3.2 Main procedure of the program

In what follows we present the main body of the parallel algorithm. This code is the same for all the cells of the computer.

cell_main()

```
1    Receiving_Data_from_Host();
2    rank=B;
3    Norms();
4    for (current_index=0;current_index<B;
           current_index++) {
5      Pivot_Element();
6      if (fabs(pivot)<EPSILON) {
7        rank=current_index;
8        break; }
9      Column_Swap();
10     Householder_Vector();
11     Householder_Product(); }
12   Sending_Data_to_Host();
```

(1-2) The cells receive the submatrices corresponding to the original matrix M according to a cyclic distribution scheme; *rank* indicates the rank of this original matrix. The maximum rank for M is B (number of columns), as $A \geq B$.

(3) This procedure finds the square of the euclidean norms of the columns of matrix M and stores them in vector *norm*. The fact that each list represents one column of the matrix favours the calculation. Each cell obtains the local norms corresponding to the column segments (local lists) it contains. By means of the reduction instruction *y_fsum* of the *AP1000* (sum by columns), the *norm* vector of a given cell is going to contain the norms of the global columns corresponding to the original matrix M.

We go through the columns of the matrix and perform all the following actions:

(5-8) Obtain in variable *pivot* the pivot element, which is the maximum of the norms of the columns whose order is higher than or equal to *current_index* (iteration we are currently processing) of our global matrix. The index of the global column containing the pivot element is called *pivot_index*. Both values (*pivot* and *pivot_index*) will be contained in all the cells. If the pivot element is zero (*EPSILON* is the required precision), the rank of the matrix is given by the value *current_index* and the algorithm ends. Initially, we calculate the local maximum of each cell (this value is going to be the same for each cell column). The global maximum is obtained by means of a reduction instruction (*x_fmax*) which finds the maximum by cell rows.

(9) If the pivot is different from zero, we perform a swap of the column we are processing with the column of the pivot element in matrix R and of their corresponding norms.

(10-11) Once the pivoting has been carried out, we apply the Householder transformations, updating the appropriate elements of matrix R, as well as the corresponding norms. In the following sections we describe the last three procedures in more detail.

(12) The local results of the cells are sent to the host in order to reconstruct the global results.

3.3 Column_Swap procedure

A swap of the square of the norm and of the column of matrix R with global index *current_index* (current column we are processing) with the norm and column corresponding to the global index *pivot_index* (column of the pivot element) is carried out. This can be observed in figure 1.

The cell column that contains the current column exchanges this column of matrix R (as well as the corresponding norm) with the cell column that contains the pivot column. Consequently, we have to send a local list to another cell (which can be the same one). In order to carry out this process of sending information to another cell, we have to indicate the start memory address where this information is located and the size. This implies that the information we send has to occupy consecutive memory positions, situation that obviously does not happen with the information contained in a data structure such as a list. This would imply a great temporal cost, since we would

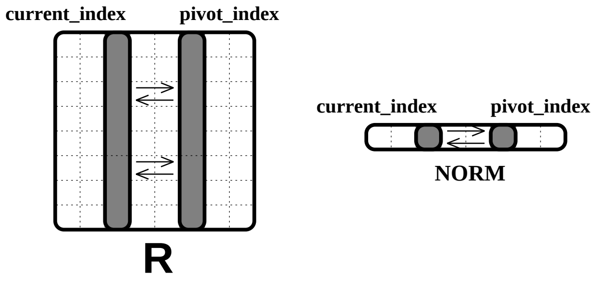

Figure 1: Column swap (pivoting)

have to establish as many connections (for sending the data) as components of the list. In the *AP1000* computer, the communication time T can be modelled by an affine function so that for a message of N bytes, $T = t_0 + t_b * N \mu s$, where t_0 is a fixed overhead in establishing the communication and t_b is the transfer time per byte. In point to point communications (using the *T-net*), $t_0 = 6.9 \mu s$ and $t_b = 0.069 \mu s$. In the *B-net* these values vary depending on several conditions: $t_0 = 43 - 60 \mu s$ and $t_b = 0.026 - 0.042 \mu s$. In [11] we find numerous experiments and measures of the communication times of the *AP1000*. Therefore, it would be interesting to send all the components of the list in a single message and this way we would save the fixed overhead in each one of the communications. In order to implement this solution, we will use a special structure, a *packed vector* [7, chapter 2], that is going to act as a buffer for sending the necessary information.

As we show in figure 2, the procedure is the following: we go through the local list corresponding to the column of matrix R from the end; as we go through it, we store each one of the elements in the buffer and erase the element from the list. The management of the erasure has been optimized: we use a specific routine for erasing the last element of the list. As header of this exchange buffer, we indicate the value of the square of the norm of that column. As a conclusion, with a single pass over the local list, it is completely erased and its content written in the swap buffer. As a result, we have our information in consecutive memory positions and we send the corresponding column of R, as well as the square of the norm in a single message. The receiving cell will reconstruct the new local list for R by means of a function for insertion. This insertion has also been optimized: we always insert at the beginning of the local list we are constructing, as the buffer is arranged according to a decreasing order of the row index.

3.4 Householder_Vector procedure

By means of this procedure we are going to get the *Householder vector (v)*, of length *dim*=*A*-*current_index*, so that *v[current_index]* (the first component of the vector) is 1 and the product $(I - 2vv^T/v^T v)\delta$ is zero in all of the components except the first one. I is the identity matrix, of dimensions

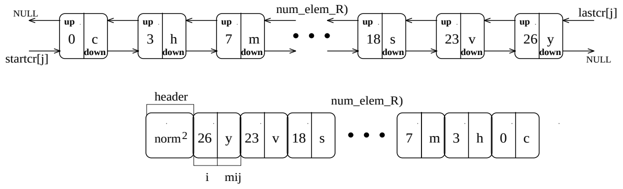

Figure 2: Buffer for sending local lists

dim∗dim and δ is the subcolumn *(α,current_index)* of our matrix, *current_index* $\leq \alpha < A$. We call matrix $P = I - 2vv^T/v^T v$, of dimensions *dim∗dim*, Householder matrix.

In order to get vector v we only require accessing the current column (access by columns) and divide it by a given value. Once this is completed, each cell column will contain vector v in a cyclic distribution. As the *Householder vector* is going to contain many null elements and is going to be broadcast to all the cells of the mesh, we are going to store it in a packed vector. Nonetheless, so as not to complicate the notation, we reference it as if it was a conventional vector.

In this procedure, only the processor column that contains the current column we are processing is going to be working.

3.5 Householder_Product procedure

In this procedure we are going to update matrix M, in particular, the submatrix S of dimensions *(A-current_index)∗(B-current_index)*, made up of elements (α, μ), *current_index* $\leq \alpha < A$ and *current_index* $\leq \mu < B$, so that we will substitute the original submatrix S by the product $P{*}S$, being P the Householder matrix. This product $P{*}S$ is equivalent to performing the operation $S + vw^T$, with $w = \beta S^T v$ and β a floating point number equal to $-2/v^T v$. Remember that the global vector v starts in index *current_index* and ends in *A-1*. On the other hand, the global vector w also starts in position *current_index* and ends in *B-1*.

As we have calculated the Householder vector in the previous procedure, the updating of S as $S{=}P{*}S$ would make all the elements of subcolumn *(α,current_index)*, *current_index* $< \alpha < A$ of our matrix zero. Thus, once all the iterations (B iterations) of the algorithm have been carried out, we will get the upper triangular matrix R, as shown in figure 3.

The cyclic distribution of the data is going to allow us to follow an optimal path through submatrix S distributed in the cells. The strategy consists in going through the columns (local lists) corresponding to this submatrix, starting from the last column. When we reach a column whose global index is less than *current_index*, we end the process. Also, in order to process each column, we go through the corresponding list from the end until we reach a row whose global index is less than *current_index*.

In the first place, we obtain in all the cells the value

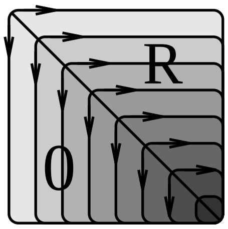

Figure 3: Process of obtaining matrix R from the original matrix M

β. After this, we calculate vector $w = \beta S^T v$, from element *w[current_index]* to *w[B-1]* (its length is therefore, *B-current_index*). It is not necessary to transpose S due to the disposition of the data. Each cell row will contain vector w which is distributed in a cyclic manner. We will update submatrix S, as $S = S + vw^T$. For this, we place ourselves in element S_{ij} of matrix S which is the one we are going to update and make $S_{ij} = S_{ij} + v_i w_j$. When this sum is carried out, it may happen that $S_{ij} = 0$ and $v_i, w_j \neq 0$, so that an element of submatrix S which was initially null, now takes a nonzero value (fill-in); we therefore insert it in the corresponding local list. It may also happen that element S_{ij} takes a value of zero and consequently, we will have to eliminate it from the corresponding local list (the opposite phenomenon to the fill-in).

This section of code is optimized because when it updates a column of submatrix S, the corresponding local list is going to suffer a single pass, always starting from the last one of its elements and until an element with a global row index lower than *current_index* is found. In addition, we have taken into account that when we perform an insertion at the beginning of a local list (due to the fill-in), all the rest of the insertions will also be at the beginning of the list. In this procedure the norms of the columns will also be updated.

We must point out that once the algorithm has ended, what we really get is a $M * \Pi = Q * R$ factorization, where Π is a permutation $B * B$, made up by the product of *rank* elementary permutations: $\Pi = \pi_0 * \pi_1 * \ldots * \pi_{rank-1}$, being each π_i, with *i=0,...,rank-1*, the identity matrix or a matrix resulting from swapping two of its columns. This is due to the pivoting we carry out in the *Column_Swap* procedure. In general, obtaining matrix Q in an explicit manner is not going to be necessary. It can be obtained by previously storing it in a factorized format. It consists in storing the Householder vectors as they are obtained, in the lower triangular part of matrix R. Each Householder vector will have a length of *A-current_index*. However, the first element (*v[current_index]*), as we have obtained it, will always be one and it will not be necessary to store it. Consequently, we will use column *current_index* of matrix R, from row *current_index+1* to row *A-1* in order to store the corresponding Householder vector. And, from this factorized format, an algorithm can be applied (*backward accumulation*) in order to get Q in an explicit way (see [10, chapter 5] for more details).

4 An application: the least squares problem

As an application example of the QR decomposition, we are going to approach a standard problem in linear algebra: the least squares problem. It consists in calculating a vector x of length B that minimizes $\|Mx - z\|_2$ (euclidean norm), where M is a matrix of dimensions $A * B$ (with $A \geq B$) and z is a vector of length A. If the rank of M is maximum (B), the least squares problem is going to have a unique solution (x_{LS}). In any other case, it is going to have an infinite number of solutions x_{SOL}, out of which there will only be one whose norm is minimum and which we will also denote as x_{LS}: $x_{LS} = x_{SOL} / \|x_{SOL}\|_2$ *is minimum*. In the case where A=B, the least squares problem is equivalent to solving a linear equation system $Mx = z$ as, obviously, $\|Mx - z\|_2 = 0$ (minimum norm).

The solution of this problem can be approached adapting the parallel algorithm that carries out the QR decomposition of matrix M. In particular, the least squares problem is going to be 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. With this algorithm we will get the unique solution to the least squares problem when $rank(M) = B$. In the case where $rank(M) < B$, we will get one of the infinite solutions: the one called basic solution, which will have a maximum of *rank* nonzero elements and that in general will not coincide with the minimum norm solution x_{LS}.

4.1 Obtaining vector $Q^T z$

Once vector z has been cyclicly distributed in each cell column, the product $Q^T z$ can be calculated at the same time we perform the Householder transformations. In order to get this product it is not necessary to have physically a vector z, as we store it directly in vector *qtz* and through an iterative process, in the end we will obtain in *qtz* the $Q^T z$ product we desire (in-place algorithm). Consequently, initially, vector *qtz* will contain vector z (of length A). And, in each one of the iterations (B, if the rank is maximum) of our parallel algorithm we will carry out the following actions for the elements of vector *qtz* and v, from component *current_index* to component *A-1*: $\lambda = \beta qtz^T v$; $qtz = qtz + v\lambda$.

Once all the iterations of the algorithm have ended, we will get the product $Q^T z$ in vector *qtz*, from index *0* to index *B-1*. Now we will add two procedures to the main body of the program: *Back_Substitution()* and *Permutation()*.

4.2 Back_Substitution procedure

By means of this procedure we will solve the upper triangular system $Rx = Q^T z$. The corresponding sequential algorithm is the following:

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

being r_{ij} element *(i,j)* of matrix R. It is a loop with data dependencies, and thus this loop must be maintained in the parallel code without any possibility of distributing it among the cells. In addition, it is necessary to access the elements of matrix R by rows. This implies a big drawback, as matrix R is stored by columns. We solve it using an auxiliary pointer vector with as many components as columns in the matrix. This would permit access to matrix R (from bottom to top) by rows going through the linked lists corresponding to the columns of the matrix only once.

Once the backsubstitution is carried out we get, in each cell row, the solution vector x of global length B cyclicly distributed, so that the global component J of vector x will be replicated in the cell column with *idx=J mod n*.

4.3 Permutation procedure

Due to the column swap carried out in the QR factorization, in order to obtain the final solution to the least squares problem we will have to apply the Π permutation to the components of vector x obtained in the previous procedure, so that we overwrite x with vector: Πx. All the cells will have a vector called *permut*, of local length B (it is the only global vector whose components are not distributed among the cells, but is completely stored in them). This vector will contain the index of the swapped column (pivot column) in each iteration. For this, we will add as first command of the *Column_Swap* procedure:

```
1       permut[current_index]=pivot_index;
```

This way, applying the swaps stored in vector *permut* starting from the end, we obtain the elements of vector x in the correct order.

5 Evaluation of the algorithm and conclusions

In order to carry out several temporal measurements of the parallel algorithm we have made use of the *Performance Analyzer* of the *AP1000* computer [2], which analyzes performance during execution and provides several temporal measures: times of the task, of inactivity of the cells, of interruptions, of calls to libraries, average, maximum and minimum values of these times for all the cells, information on various events, etc.

The sparse matrices we have used for the evaluation of the algorithm were obtained from the *Harwell-Boeing* collection [8]. Table 1 shows a description of the matrices we have selected. $A*B$ indicates the dimensions of the matrix; *El(M)* the number of nonzero elements and *El(R)* the number of nonzero elements of matrix R. The selection of these matrices is due to the fact that all of them include vector z, necessary for solving the least squares problem and,

Matrix	Origin	A*B	El(M)	El(R)
WELL1033	Least squares problems in surveying	1033*320	4732	13832
WELL1850	Least squares problems in surveying	1850*712	8758	52303
SHERMAN1	Oil reservoir simulation	1000*1000	3750	66187

Table 1: Harwell-Boeing sparse matrices for the evaluation

Matrix	1*1	2*2	4*4	8*8
WELL1033	112.31	47.13	16.10	6.41
WELL1850	1515.07	408.13	120.71	35.93
SHERMAN1	574.14	185.91	62.78	26.70

Table 2: Execution times (in seconds)

in particular, matrices *WELL1033* and *WELL1850* are specific for the solution of this problem.

Table 2 shows the execution times (in seconds) for some of the configurations of the mesh. All the measures depicted include the time required for carrying out the QR factorization and solving the least squares problem. The times required for the distribution and recollection of the data are not included because we assume that the problem of solving least squares is a possible subproblem within a wider program.

Figure 4 shows the efficiency and the speed-up obtained for these sparse matrices. Observe that the results are better for larger sizes of the matrix. This is because with larger matrices there are more calculations and thus the parallelism is more efficiently used. Small sizes of matrix M result in low efficiencies because the calculation time of the task itself is small with respect to the additional time required by communications (message passing) and other factors. In addition, a larger number of nonzero elements (as is the case of matrix *WELL1850*) also provides longer effective calculation times and the parallelism applied motivates that the calculation time of the task in each processor is large with respect to the times that are not related with the task itself.

The factors that are going to influence the fill-in of a given sparse matrix are the following: the dimension and rank of the matrix, the degree of sparsity (number of null elements) and a factor of great importance but difficult to model, the pattern of the matrix, that is, the location of the nonzero elements. Thus, with the same dimension, rank and degree of dispersion for two matrices, the fill-in may vary significantly, depending on how the nonzero elements are placed. Consequently, there will also be large variations in the execution times and efficiencies. The reduction of the fill-in is the work we are currently carrying out. As a conclusion, the sparse approach to the QR factorization we have presented is more adequate with respect to the dense approach the bigger the dimension of matrix M and the smaller the number of nonzero elements (high degree of sparsity).

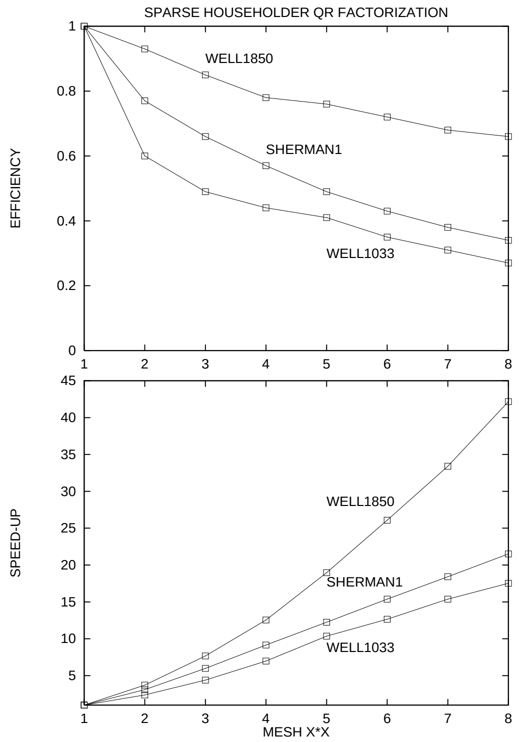

Figure 4: Efficiency and speed-up measures

Acknowledgements

We gratefully thank to the Fujitsu Parallel Computing Research Centre (Japan) and the Galician Supercomputing Centre (Spain) for giving us access to the AP1000 machine.

References

[1] *AP1000 Program Development Guide*, Fujitsu Laboratories Ltd., 3rd edition, July 1993.

[2] *AP1000 User's Guide*, Fujitsu Laboratories Ltd., June 1993.

[3] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 1993.

[4] C.H. Bischof, Adaptive Blocking in the QR Factorization, *The Journal of Supercomputing*, Vol.3, pp.193-208, 1989.

[5] R.H. Bisseling, J.G.G. Van de Vorst and A.F. Van der Stappen, Parallel Sparse LU Decomposition on a Mesh Network of Transputers, *SIAM J. Matrix Anal. Appl.*, Vol.14, n.3, pp.853-879, July 1993.

[6] *CASIM User's Guide*, Fujitsu Laboratories Ltd., 4th edition, August 1991.

[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] J.A. George and M.T. Heath, Solution of Sparse Linear Least Squares Problems using Givens Rotations, *Linear Algebra Appl.*, Vol.34, pp.69-83, 1980.

[10] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 2nd edition, 1989.

[11] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato and M. Ikesaka, Third Generation Message Passing Computer AP1000, *Proceedings of ISS'92*, pp.46-55, November 1992.

[12] H. Ishihata, T. Horie and T. Shimizu, Architecture for the AP1000 Highly Parallel Computer, *Fujitsu Sci. Tech. Journal*, Vol.29, n.1, pp.6-14, March 1993.

[13] S.G. Kratzer, Sparse QR Factorization on a Massively Parallel Computer, *The Journal of Supercomputing*, Vol.6, pp.237-255, 1992.

[14] J.H.W. Liu, On General Row Merging Schemes for Sparse Givens Transformations, *SIAM J. Sci. Statist. Computing*, Vol.7, pp.1190-1211, 1986.

[15] P. Matstoms, *Sparse QR Factorization with Applications to Linear Least Squares Problems*, PhD thesis, Department of Mathemathics, Linköping University, Sweden, 1994.

[16] A. Pothen and P. Raghavan, Distributed Orthogonal Factorization: Givens and Householder Algorithms, *SIAM J. Sci. Statist. Computing*, Vol.10, pp.1113-1134, 1989.

[17] 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*, Vol.12, pp.60-69, 1991.

[18] B.B. Zhou and R.P. Brent, *Parallel Implementation of QRD Algorithms on the Fujitsu AP1000*, Technical Report TR-CS-93-12, Computer Sciences Laboratory, The Australian National University, Canberra, ACT 0200, Australia, November 1993.