Parallel Sparse Modified Gram-Schmidt QR Decomposition*

Ramón Doallo**[1], Basilio B. Fraguela[1], Juan Touriño[1] and Emilio L. Zapata[2]

[1] Dept. Electrónica y Sistemas, University of La Coruña, Campus de Elviña s/n, 15071 La Coruña (SPAIN) {doallo, basilio, juan}@udc.es

[2] Dept. Arquitectura de Computadores, University of Málaga, Campus de Teatinos s/n, 29071 Málaga (SPAIN) ezapata@atc.ctima.uma.es

Abstract. We present a parallel computational method for the QR decomposition with column pivoting of a sparse matrix by means of Modified Gram-Schmidt orthogonalization. Nonzero elements of the matrix M to be decomposed are stored in a one-dimensional doubly linked list data structure. We discuse a strategy to reduce fill-in in order to get memory savings and decrease the computation times. As an application of QR decomposition, we describe the least squares problem. This algorithm was designed for a message passing multiprocessor and has been evaluated on a Cray T3D, using the Harwell-Boeing sparse matrix collection.

1 Introduction

QR factorization is a computational 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 decomposition has many applications in linear algebra to solve linear systems of equations, Least Squares Problems (LSP), linear programs and eigenvalue problems. It is necessary to solve these problems in many scientific fields, such as fluid dynamics, structural analysis, circuit simulation ...

There are several methods for finding this factorization [1, Chap. 5]: Modified Gram-Schmidt algorithm (MGS), Householder reflections and Givens rotations. 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: [2] for the *Intel iPSC/1*, [3] for the *nCUBE/10*, [4] for a network of transputers, [5] for the *nCUBE 2*, [6] for the *CM-200*, all of which are for dense matrices; and [7] (*CM-2*), [8] (*Fujitsu AP1000*), [9] (*Cray T3D*) for sparse matrices. We have implemented the MGS procedure with column pivoting for sparse matrices on the *Cray T3D MIMD* distributed memory computer.

* 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)

** Authors are listed alphabetically

This paper is organized as follows: in Sect. 2 we describe the sequential and parallel MGS algorithm, as well as a strategy to reduce fill-in. An application of QR decomposition, the LSP, is shown in Sect. 3, and experimental results are discussed in Sect. 4.

2 The MGS Algorithm

MGS is a rearrangement of the Classical Gram-Schmidt algorithm with better numerical properties. This method obtains matrices Q and R of dimensions $A \times B$ and $B \times B$, respectively. Matrix M is overwritten by matrix Q (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, and to provide numerical stability:

$rank = B$;
for *(j=0; j<B; j++)*

$$norm_j = \sum_{i=0}^{A-1} m_{ij}^2; \quad (1)$$

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 ($norm_{px} < \alpha$) { (3)

$rank = cx$;

break;

}

else swap ($norm_{cx}$, col_{cx} of M and R), ($norm_{px}$, col_{px} of M and R); (4)

$$r_{cx\,cx} = \sqrt{norm_{cx}}; \quad (5)$$

for *(i=0; i<A; i++)*

$$m_{i\,cx} = m_{i\,cx} / r_{cx\,cx}; \quad (6)$$

for *(j=cx+1; j<B; j++)* {

$$r_{cx\,j} = \sum_{i=0}^{A-1} m_{i\,cx} \cdot m_{ij}; \quad (7)$$

$$norm_j = norm_j - r_{cx\,j}^2; \quad (8)$$

for *(i=0; i<A; i++)*

$$m_{ij} = m_{ij} - m_{i\,cx} \cdot r_{cx\,j}; \quad (9)$$

}

}

In step (1) the squares of the euclidean norms of the columns of matrix M are calculated. Then, a loop of B iterations (if the rank of M is maximum) is

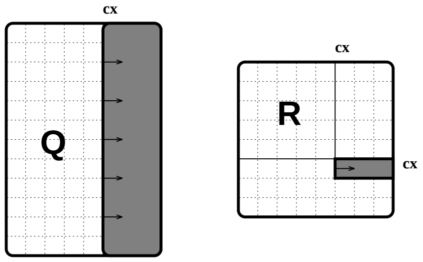

Fig. 1. Process of updating matrices Q and R

performed, which consists of the following steps: the pivot column (px) and the pivot element ($norm_{px}$) are selected (2). The pivot element is the maximum of the norms of those columns with an index $\geq cx$. If this pivot is close to 0 (α 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 in matrices Q and R, as well as a swap of their norms are performed (4). 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. Elements m_{ij} of matrix Q and elements $r_{cx\,j}$ of matrix R, $0 \leq i < A$, $cx \leq j < B$ are updated in (5-9) (see Fig. 1); the corresponding norms are updated too.

Once the algorithm has been concluded, what we actually obtain is a $M \times \Pi = Q \times R$ factorization, where Π is a permutation $B \times B$ due to the pivoting.

2.1 Reducing Fill-in

The factors that influence the fill-in of a sparse matrix are the dimensions and rank of the matrix, the degree of sparsity and the pattern of the matrix (the location of the nonzero elements). Thus, fill-in may vary significantly depending on how the nonzero elements are placed. Given the pattern of the matrix M to be decomposed by means of the MGS procedure, the structure of the resulting matrices Q and R is determined in [10]. A high fill-in is an undesirable situation due to increased storage cost and computation time. The most habitual heuristic strategy employed in the LU factorization for controlling the degree of sparsity is the Markowitz criterion [11, Chap. 7]. Furthermore, numerical stability must be ensured in the LU factorization, by avoiding the selection of pivots with a low absolute value.

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

Obtain px, $cx \leq px < B$, such that

$$\left\{ \epsilon \left(\frac{zero_{px}}{\max\limits_{cx \leq j < B} zero_j} \right) + (1-\epsilon) \left(\frac{norm_{px}}{\max\limits_{cx \leq j < B} norm_j} \right) \right\} \text{ is maximum} \qquad (10)$$

Table 1. Harwell-Boeing sparse matrices

Matrix	$A \times B$	*El(M)*	*% El(M)*
BCSSTK14	1806 × 1806	32630	1.00%
BCSSTK21	3600 × 3600	15100	0.12%
BCSSTK27	1224 × 1224	28675	1.91%

Table 2. Fill-in reduction

Matrix	$\epsilon = 0$		$\epsilon \approx 1$		*%Red.*
	El(R)	*%El(R)*	*El(R)*	*%El(R)*	
BCSSTK14	926767	56.80%	178559	10.94%	80.73%
BCSSTK21	957374	14.77%	137833	2.13%	85.60%
BCSSTK27	301530	40.22%	72732	9.70%	75.88%

where $zero_j$ is the number of nonzero elements in column j, and ϵ, $0 \leq \epsilon \leq 1$, is a prefixed parameter. Obviously, for $\epsilon = 0$, the pivot column selection criterion is equivalent to the one described in (2). Although we try to reduce fill-in as much as possible by choosing a pivot column with few nonzero elements ($\epsilon \approx 1$), we shall always keep a minimum degree of numerical stability in the algorithm by discarding in the selection process those columns with a norm close to zero.

In order to test this strategy, we have chosen three matrices used in structural engineering from the Harwell-Boeing sparse matrix collection [12]. A description of these matrices is presented in table 1, where $A \times B$ are the dimensions of the matrix, *El(M)* is the number of nonzero elements of M and *%El(M)* is the percentage of these elements. Table 2 shows the fill-in in matrix R after factorization; *El(R)* and *%El(R)* are the number and percentage of nonzero elements, for $\epsilon = 0$ and $\epsilon \approx 1$ ($\epsilon = 0.999$) in expression (10); *%Red.* is the percentage of reduction in the number of nonzero elements achieved with $\epsilon \approx 1$. Fill-in decreased by 80%, on the average, for this set of sparse matrices, which is a very noticeable reduction.

2.2 The Parallel Algorithm

The parallel algorithm developed has been generalized for any number of processing elements (PEs) and for any dimension of matrix M, so that its execution for a single processor is equivalent to the sequential algorithm. We find MGS parallel algorithms for dense matrices in [3] and [4].

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 [13], but these elements are stored in one-dimensional doubly linked lists instead of vectors. This distribution provides data and load balancing. Each list 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 according to the growing order of the row index. In this algorithm only efficient access by columns is necessary; which implies large memory and computation savings. Nevertheless, in a LU decomposition for sparse matrices, we require a data structure to access both by rows and by columns, such as a two-dimensional doubly linked list [14].

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) that 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 corresponding 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 looks for the maximum norm by rows of PEs. As a result, the pivot element will be contained in all the PEs, as well as px, the index of the pivot column (2). The parallelization of the strategy to reduce fill-in, (10), 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 [11, Chap. 2] that acts as a buffer for exchanging data. Therefore, the corresponding column of Q and R, as well as the square of the norm, are sent in a single message.

After swapping is carried out, the current column cx of matrix Q is normalized by dividing it by its norm (6). This normalized column is stored in a packed vector called *vcol*. The column of PEs which contains the normalized column, broadcasts it to all the corresponding PEs on the X axis. For each column with a global index $>cx$, the vector *vsum* (not packed), stores the local dot product of this column and the normalized current column. Finally, the global dot product for each column is obtained in *vsum* by means of a reduction instruction. The elements of *vsum* will be the row cx of matrix R (7), from column cx to the last column, as R is upper triangular. Row access to matrix R is not necessary in order to insert this new row because this insertion is made at the end of the R lists, which means column access. Now, the corresponding elements, m_{ij}, of Q are updated (9). Thus, m_{ij} is overwritten with $m_{ij} - vcol_i \cdot vsum_j$. This is a local operation and communications are not required. When this operation is carried out, it may happen that $m_{ij} = 0$ and $vcol_i, vsum_j \neq 0$, so that an element of matrix Q, which was initially null, now takes a nonzero value (fill-in); we, therefore, insert it into the corresponding local list. It may also happen that element m_{ij} takes a value close to zero and consequently, must be erased from its local list. The updating of the norms, (8), is also local.

3 Solving the Least Squares Problem

The LSP consists of calculating 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 LSP has a unique solution (x_{LS}). Otherwise, it has an infinite number of solutions x_{SOL}, one of them having 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 LSP is equivalent to solving a linear equation system $Mx = z$, as $\|Mx - z\|_2 = 0$.

The solution of this problem can be approached by adapting the parallel algorithm that carries out the QR decomposition of matrix M. In particular, the LSP problem is equivalent to solving the upper triangular system: $R\Pi^T x = Q^T z$. If $rank(M) = B$, this algorithm calculates the unique solution to the LSP. 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 Obtaining Vector $Q^T z$

Product $Q^T z$ is calculated at the same time that QR factorization is performed. Previously, vector z (we assume it is dense) was distributed in each column of PEs, so that the global component I of z is replicated in the row of PEs with *idy*=*I mod m*. Vector *qtz* will store the product $Q^T z$, so that one element of *qtz* is obtained for each iteration of the algorithm. Each element is a dot product between vector z and the column of Q obtained in that iteration. Once all the iterations of the algorithm have ended, product $Q^T z$ is stored in vector *qtz*, from index *0* to global index *B-1*.

3.2 Back-Substitution and Permutation

We solve the upper triangular system $Rx = Q^T z$ by means of a back-substitution. The corresponding sequential algorithm is the following:

for *(i=rank-1;i≥0;i--)*

$$x_i = (qtz_i - \sum_{j=i+1}^{rank-1} r_{ij} \cdot x_j)/r_{ii}; \qquad (11)$$

This loop has data dependencies, and thus it must be maintained in the 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 using an auxiliary pointer vector with as many components as columns in the matrix. In this way we can go through the linked lists corresponding to the columns of the matrix only once in order to gain access to matrix R (from bottom to top) by rows. Another option is to apply the column version of back-substitution [1, Chap. 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, Π permutation must be applied to the components of vector x, so that x is overwritten by vector Πx. 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 pivot column in each iteration (px) and, by applying these swaps starting from the end, the elements of vector x are obtained in the correct order.

Table 3. Execution times (in seconds) for $\epsilon = 0$ and $\epsilon \approx 1$

	$\epsilon = 0$					$\epsilon \approx 1$				
Matrix	*1*	*4*	*16*	*64*	*128*	*1*	*4*	*16*	*64*	*128*
BCSSTK14	1976.36	508.43	137.64	41.96	26.83	207.98	60.05	24.33	12.30	10.08
BCSSTK21	3459.67	931.44	250.83	79.71	53.93	179.63	70.24	32.96	22.99	19.69
BCSSTK27	368.66	103.93	29.24	10.04	7.35	35.26	12.41	7.10	4.40	3.99

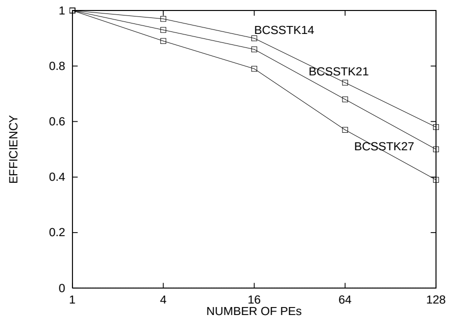

Fig. 2. Efficiencies for $\epsilon = 0$

4 Experimental Results and Conclusions

The algorithm was implemented on a *Cray T3D* supercomputer [15] 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*. We used low latency communication functions, such as *pvm_fastsend* and *pvm_fastrecv* (non-standard PVM functions) for messages with a length of less than 256 bytes. We also developed reduction instructions suitable for our algorithm to reduce communications.

We tested the performance of our parallel algorithm using the Harwell-Boeing matrices described in table 1. Table 3 shows the execution times obtained (in seconds) for 1, 4, 16, 64 and 128 PEs for: $\epsilon = 0$ (without taking into account our strategy to reduce fill-in), and $\epsilon \approx 1$ (applying the fill-in control approach). These times include the QR decomposition as well as the resolution of the LSP. Nevertheless, the time required for data distribution and collection 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 by 71% for matrices *BCSSTK14* and *BCSSTK21* using 64 PEs. This decreasing is lower as the number of PEs increases. This is 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 be higher.

Fig. 2 shows the efficiencies for $\epsilon = 0$ attained for each matrix. For example, the efficiencies for *BCSSTK14*, *BCSSTK21* and *BCSSTK27* using 128 PEs are 0.58, 0.50 and 0.39, respectively. As we 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 low 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. Golub, G.H., Van Loan, C.F.: Matrix Computations. The Johns Hopkins University Press, second edition (1989)
2. Bischof, C.H.: Adaptive Blocking in the QR Factorization. The Journal of Supercomputing, **3** (1989) 193–208
3. Zapata, E.L., Lamas, J.A., Rivera, F.F., Plata, O.G.: Modified Gram-Schmidt QR Factorization on Hypercube SIMD Computers. Journal of Parallel and Distributed Computing, **12** (1991) 60–69
4. Waring, L.C., Clint, M.: Parallel Gram-Schmidt Orthogonalisation on a Network of Transputers. Parallel Computing, **17** (1991) 1043–1050
5. Hendrickson, B.: Parallel QR Factorization Using the Torus-Wrap Mapping. Parallel Computing, **19** (1993) 1259–1271
6. Bendtsen, C., Hansen, P.C., Madsen, K., Nielsen, H.B., Pinar, M.: Implementation of QR Up and Downdating on a Massively Parallel Computer. Parallel Computing, **21** (1995) 49–61
7. Kratzer, S.G.: Sparse QR Factorization on a Massively Parallel Computer. The Journal of Supercomputing, **6** (1992) 237–255
8. Doallo, R., Touriño, J., Zapata, E.L.: Sparse Householder QR Factorization on a Mesh. In Fourth Euromicro Workshop on Parallel and Distributed Processing, Braga (Portugal), IEEE Computer Society Press (January 1996) 33–39
9. Touriño, J., Doallo, R., Zapata, E.L.: Sparse Givens QR Factorization on a Multiprocessor. In MPCS'96 (submitted)
10. Hare, D.R., Johnson, C.R., Olesky, D.D., Van Den Driessche, P.: Sparsity Analysis of the QR Factorization. SIAM J. Matrix Anal. Appl., **14(3)** (July 1993) 655–669
11. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Clarendon Press (1986)
12. Duff, I.S., Grimes, R.G., Lewis, J.G.: User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR-PA-92-96, CERFACS (October 1992)
13. Romero, L.F., Zapata, E.L.: Data Distributions for Sparse Matrix Vector Multiplication. Parallel Computing, **21(4)** (1995) 583–605
14. Asenjo, R., Zapata, E.L.: Sparse LU Factorization on the Cray T3D. In Int'l Conference on High-Performance Computing and Networking, Milan (Italy), Springer-Verlag LNCS no.919 (May 1995) 690–696
15. Cray Research, Inc: Cray T3D. Technical Summary, (September 1993)

This article was processed using the LaTeX macro package with LLNCS style