New Parallel (Rank-Revealing) *QR* Factorization Algorithms

Rudnei Dias da Cunha, Dulcenéia Becker, and James Carlton Patterson

Programa de Pós-Graduação em Matemática Aplicada, Instituto de Matemática,
Universidade Federal do Rio Grande do Sul, 91509-900 Porto Alegre – RS, Brazil
{rudnei,dubecker,patterson}@mat.ufrgs.br

Abstract. We present a new algorithm to compute the *QR* factorization of a matrix $A_{m \times n}$ intended for use when $m \gg n$. The algorithm uses a reduction strategy to perform the factorization which in turn allows a good degree of parallelism. It is then integrated into a parallel implementation of the *QR* factorization with column pivoting algorithm due to Golub and Van Loan, which allows the determination of the rank of A . The algorithms were coded in FORTRAN 90 using the MPI library. Results are presented for several different problem sizes on an IBM 9076 SP/2 parallel computer.

1 Introduction

The *QR* factorization of a matrix $A_{m \times n}$ is required in some situations, especially when solving a (block) linear least-squares problem,

$$A_{m \times n} X_{n \times k} = B_{m \times k}, \quad m \geq n, \quad k \geq 1. \quad (1)$$

This is also required inside some solvers for block systems of linear equations, like the Block GMRES method by Saad [12].

The *QR* factorization of A leads to a orthogonal matrix $Q_{m \times m}$ and an upper triangular matrix $R_{m \times n}$ such that

$$QR = Q \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = A, \quad (2)$$

where R_1 is $n \times n$ and X can easily be obtained solving the (block) triangular system

$$R_1 X = Q^T B. \quad (3)$$

To compute Q , one usually uses either Householder reflections or Givens's rotations, or both (see [6, pp. 193-220]); Q may not be stored explicitly, since its action over a matrix or vector may be obtained via a sequence of rank-1 updates or applying Givens's rotations to that matrix/vector.

However, if $r = \text{rank}(A) < n$, we can avoid doing unnecessary work if we can determine which columns of A are multiples of others. This can be accomplished via the *QR* factorization with column pivoting, from which one obtains

$$QR = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} = AP, \quad \|R_{22}\| \approx 0 \quad (4)$$

where P is a permutation matrix, Q has orthonormal columns, R is upper triangular and R_{11} is of order r , the numeric rank of A .

For large problems, one may need to use a parallel algorithm to perform either of these factorizations. Parallel algorithms to compute (2) are presented in [10] (which requires $O(n^2)$ processors), [4] (where a recursive strategy to obtain Q and R is presented) and [13]. The latter presents a number of different parallel algorithms, ranging from the modified Gram-Schmidt to hybrid algorithms using Householder reflections and Givens's rotations.

The algorithms available to compute the QR factorization with column pivoting (see, for instance, [5], [2], [6, pp. 233-236] and [1]) use the non-pivoting QR factorization as an inner-step. Our main aim in this paper is to present a new algorithm to compute the non-pivoting QR factorization of A and its use inside a parallel implementation of the QR factorization with column pivoting – a *rank-revealing* QR algorithm – presented in [6, pp. 233-236], for matrices where the number of columns is very small when compared to the number of rows. The parallel QR factorization algorithm presented here share some similarities with the row merging strategy proposed by [9] and multifrontal QR factorizations [11]. The novelty in our algorithm is to introduce the merging of the triangular factors in a tree-like reduction scheme.

Our motivation lies in the need to solve large, sparse block linear systems of equations using the Block GMRES method [12] in parallel, using message-passing constructs; an initial implementation of this method used Bischof's (sequential) rank-revealing QR package [1]. However this approach did not provide an adequate performance, leading us to develop the parallel algorithms presented here.

1.1 Parallel QR Factorization

In some applications, one will need the full Q matrix, whereas in others – like the solution of system (3) – just a few columns of Q are necessary. Our algorithm, which will be presented now, can cope with both situations, though for some larger problems the full Q matrix will not be stored explicitly.

The parallel QR algorithm (to which we will refer as PQR) we have developed shares some concepts with those presented in [13]. In that paper, a number of parallel QR algorithms are presented, their main differences being in the way data is partitioned (rows or columns); the most efficient one uses a row partitioning and is made up of two parts, one completely parallel, and the other involving communication between the processors, as if they were interconnected in a ring. Our algorithm differs from this exactly in the latter part. We have also been inspired by the recursive algorithm presented in [4].

The PQR algorithm is written using the *Single-Program, Multiple-Data, SPMD* model. We partition the m rows of the matrix to be factored (of size $m \times n$) among p processors, each having at most $\hat{m} = \lfloor m/p \rfloor + 1$ rows. For a processor q , we call $[i_q, f_q]$ the range of rows of A stored on it ($\hat{m}_q = f_q - i_q + 1$).

A restriction we place on the algorithm is that $\hat{m} \geq n$. This restriction is made since we assume that $m \gg n$ and n is, typically, $2 \leq n \leq 10$; therefore, the triangular system (3) is very small and if one intended to solve it in parallel,

it would yield a very poor performance (see, for instance, [7], [8] and [3]). The matrix R_1 may be made available in all processors at the end of PQR; this would allow for an easy solution of (3) by all processors involved.

Before applying the PQR algorithm we copy the factoring matrix into a matrix R ; the algorithm then proceeds in two parts. At first, every processor performs a *local* Householder factorization, obtaining as a result a triangular block of order n ; the Householder reflectors are stored in the strictly lower triangular portion of R . This part of the algorithm is perfectly parallel and has a *flop* count of $(2\hat{m}n + \hat{m} + n)\gamma$, where γ is the time to perform either a multiplication or addition.

Now one must annihilate all the triangular blocks but the first; these are stored in processors $1, 2, \dots, p-1$. The key idea here is to note that those blocks *do not* need to be annihilated in an orderly fashion, say $p-1$ with $p-2$; then $p-2$ with $p-3$, etc. In fact, it is possible to perform this elimination phase using pairs of blocks, as if one were traversing a binary tree upwards; all the annihilations at the same level in that tree can be made in parallel, independently. This is commonly called a *reduction* operation in parallel computing terminology.

In this latter phase we use Givens's rotations to update the topmost triangular block of a given pair of processors, each processor exchanging with the other its triangular block, and applying the rotations locally. The sine/co-sine values are stored using a single value τ as given in [6, pp. 202]; between each pair of processors, there will be $(n^2 + n)/2$ τ elements computed; they are stored in an array, to be used later for computing Q . The total operation cost for this phase is $\lceil \log_2 p \rceil (T_c((n^2 + n)/2) + 3(n^2 + n)\gamma)$, where $T_c(w) = \alpha + w\beta$ is the time taken to transfer w words between two processors, α being the latency and β the rate of transfer. Figure 1 gives an example of the workings of the PQR algorithm for a matrix with 8 rows and 2 columns, partitioned across 4 processors.

The pairs of processors used in the second part of the PQR algorithm are given by simply grouping together, initially, a processor and its neighbour, e.g. $(0, 1), (2, 3), \dots, (p-2, p-1)$; then in the next recursion level we group the leftmost processor of a pair, $(0, 2), (4, 6), \dots$, and proceed in this fashion until the last pair is $(0, \lfloor p/2 \rfloor)$, with the result that the triangular block stored in processor 0 contains the actual triangular matrix R_1 in the QR factorization. If

Fig. 1. The PQR algorithm: obtaining the R matrix; $m = 8, n = 2, p = 4$.

the number of active processors in each level of recursion is not even, we simply add another pair including it and another suitable processor. We note that other groupings could be used to take advantage of a given topology of a network of processors, minimizing the length of the path needed for the messages to traverse.

1.2 Obtaining Q

The orthogonal matrix Q (or some of its first columns) is obtained after A is triangularized. Again, Q is partitioned by rows, in the same way as A was partitioned. The computation of Q is done in two phases, using backward accumulation. The Householder reflectors, which are stored in each processor in the strictly lower portion of R , are applied to Q . As they were computed for each block $\hat{m} \times n$ of R , the overall effect of this part is that Q ends up with a very definite block structure, as shown in Figure 2. Note the relationship between the sizes of the blocks and the number of rows of A stored in each processor (see the end of the last section).

The Givens's rotations are now to be applied to Q , again pairing two processors as was done for triangularizing A . For a pair (s, t) , $s < t$ (storing rows $[i_s, f_s]$ and $[i_t, f_t]$ of A), this will lead to $f_t - i_t + 1$ rows being added, *below* the Q block on s , to the first m columns of that block (therefore, due to the partitioning of Q , these new elements will be stored in t); and, conversely, $f_s - i_s + 1$ rows will be added, *above* the Q block on t , to the first m columns of that block (but these will be stored in s). Of course, those elements of Q already stored in a processor will be modified when applying the rotations. Figure 3-(a) shows Q as a result of the pairs $(0, 1)$ and $(2, 3)$ having applied their Givens's rotations.

Following this scheme, consider the case when we move upwards one level on the recursion tree and have, now, to operate on a pair (s, u) , having previously applied the rotations to pairs (s, t) and (u, v) . Then, on processor s , the number of rows created will be $f_v - i_u + 1$ on the first m columns of the Q block respective to s ; and there will be $f_t - i_s + 1$ new rows on the first m columns of the block corresponding to u on Q .

$$\begin{bmatrix} \times & \times & & & & \\ \times & \times & & & & \\ & & \times & \times & & \\ & & \times & \times & & \\ & & & & \times & \times \\ & & & & \times & \times \\ & & & & & \times & \times \\ & & & & & \times & \times \end{bmatrix}$$

Fig. 2. Computing the Q matrix: the result after each processor has applied its own Householder reflectors.

$$(a) \begin{matrix} i_s \rightarrow \\ f_s \rightarrow \\ i_t \rightarrow \\ f_t \rightarrow \\ i_u \rightarrow \\ f_u \rightarrow \\ i_v \rightarrow \\ f_v \rightarrow \end{matrix} \begin{bmatrix} \times & \times & \diamond & \diamond & & & \\ & \times & \times & \diamond & \diamond & & \\ & & \diamond & \diamond & \times & \times & \\ & & \diamond & \diamond & \times & \times & \\ & & & & \times & \times & \diamond & \diamond \\ & & & & \times & \times & \diamond & \diamond \\ & & & & \diamond & \diamond & \times & \times \\ & & & & \diamond & \diamond & \times & \times \end{bmatrix}, \quad (b) \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{bmatrix} \times & \times & \diamond & \diamond & + & + & \\ & \times & \times & \diamond & \diamond & + & + \\ & & \diamond & \diamond & \times & \times & + & + \\ & & \diamond & \diamond & \times & \times & + & + \\ & & \diamond & \diamond & \times & \times & + & + \\ + & + & & & \times & \times & \diamond & \diamond \\ + & + & & & \times & \times & \diamond & \diamond \\ + & + & & & \diamond & \diamond & \times & \times \\ + & + & & & \diamond & \diamond & \times & \times \end{bmatrix}$$

Fig. 3. Computing the Q matrix: (a) Q after the pairs of processors $(s, t) = (0, 1)$ and $(u, v) = (2, 3)$ have applied Givens's rotations to their columns of Q ; the symbol $\diamond$ indicates those elements of Q created as a result. (b) Q after the pair of processors $(s, u) = (0, 2)$ have applied Givens's rotations to their columns of Q , where those elements of Q created during the process are indicated by $+$.

When we reach the root of the recursion tree, we will have the pair $(0, \lfloor p/2 \rfloor)$. Therefore, if we proceed with the above procedure, the first m columns of Q will be filled-in completely, as in Figure 3-(b), where we show Q after the pair $(0, 2)$ applies their rotations to Q . The total cost of computing Q , simultaneously in each processor, is $(4(\hat{m}^2 n - \hat{m} n^2 + n^3/3) + 3\hat{m}(n^2 + n))\gamma$.

Now, if $H^{(q)} = H_1^{(q)} H_2^{(q)} \dots H_m^{(q)}$ is the matrix resulting from applying all necessary Householder transformations to triangularize a block of A stored in processor q ; and $G = G_1 G_2 \dots G_h$, where $h = \lceil \log_2 p \rceil$ is the number of levels of recursion in the tree, then $Q = H^{(1)} H^{(2)} \dots H^{(p)} G$.

The PQR algorithm is outlined below, according to the previous description. Note that step 1 is completely parallel, without any communication whatsoever between the processors; steps 2 to 5 entail communication between the processors, with messages of *constant size* $(n^2 + n)/2$ words (single- or double-precision). Step 6 does not require communication as the Givens's rotations are applied with the sine/co-sine pairs already computed in steps 2 to 5.

ALGORITHM PQR:

1. Every processor triangularizes its block $A_{\hat{m} \times n}$ using Householder reflections;
2. Compute pairs of processors (s, t) ;
3. for all pairs (s, t) along the reduction tree do
4. processor s receive triangular block from t (of size n);
5. processor s retriangularize its own block using the block from t (as if the latter had been annihilated);
- endfor
6. Compute Q as in §1.2.

1.3 The PQR Algorithm and the Least-Squares Problem

If one wishes to solve (1), then the solution to (3) must be obtained. Since B is $m \times k$, we need only the first m columns of Q to compute $Q^T B$, whose rows are partitioned across the processors; this product can be easily computed then.

We note that while in our explanation we have shown the full matrix Q , the actual implementation of the algorithm stores only the number of columns of Q (in each block of rows per processor) needed by the application, plus a small number of auxiliary arrays. Of course, if one needs the full Q matrix, it may not fit in the memory available; in this case, the above accumulation of Q must be modified to compute the action of the individual transformation matrices into a vector or a matrix.

2 The Rank-Revealing QR Algorithm

The PQR algorithm is used inside a parallel version of the RRQR algorithm by Golub and Van Loan [6, pp. 233-236], called PRRQR, which is also written using the SPMD model. This RRQR algorithm is easily parallelizable, requiring just parallel inner-products; we do not require a parallel swapping procedure for the columns, since the matrix to be factored is partitioned by rows (therefore, all processors have the same columns and any column swappings are made locally in every processor simultaneously).

A change made to that algorithm is that the QR factorization – performed by PQR – is done only once. After PQR is completed, the matrix R_1 , which ends up in processor 0, is broadcast to all other processors. Afterwards, if a column swapping is performed by the PRRQR algorithm, then only the upper $n \times n$ block needs to be re-triangularized (and Q must be updated); this re-triangularization is done using Givens's rotations. However, now all processors hold a copy of R_1 and their own blocks of rows of Q , allowing the *local* use of Givens's rotations to re-triangularize R_1 and update of Q . We note that just a few of the rows of R_1 will need to be updated and our implementation performs only the necessary work, keeping track of the rows changed. The algorithm is outlined below; communication occurs only in steps 1 (inside PQR) and 2 (a broadcast of a message $(n^2 + n)/2$ words long).

ALGORITHM PRRQR:

1. *Every processor calls PQR;*
2. *The root processor, 0, broadcasts R_1 to the other processors;*
3. *In every processor, independently:*
4. *for all columns of R_1*
5. *if near-dependency is detected, perform:*
6. *column swapping in R_1 ;*
7. *retriangularize its own copy of R_1 ;*
8. *update its block of rows of Q .*

3 Experiments

The PQR and RRQR algorithms were implemented using FORTRAN90 and MPI, and there are versions in real and complex arithmetic, each of which in single- and double-precision.

We have tested our double-precision implementations of both algorithms on a number of random matrices of different sizes. The tests were carried out on an IBM 9076 SP/2 parallel computer housed at the Brazilian “*Laboratório Nacional de Computação Científica, LNCC*” (National Scientific Computing Laboratory). The computer was not entirely dedicated to our testings and therefore there was some fluctuation in the traffic across its network.

Tables 1 and 2 show the running times (in seconds) and the respective speed-ups ($S_p = T_1/T_p$) of PQR and PRRQR for some large matrices, with $m \gg n$.

Table 1. Timings (in seconds) and speed-ups for PQR; the column labelled “LAPACK” shows the timings obtained with solving the same problem using LAPACK’s DGEQPF/DORGQR routines.

m	n	LAPACK	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
20000	5	0.07	0.16	0.15	0.07	0.05	0.05
				1.07	2.29	3.20	3.20
	10	0.22	0.52	0.32	0.19	0.12	0.09
				1.63	2.74	4.33	5.78
40000	5	0.15	0.31	0.30	0.12	0.09	0.07
				1.03	2.58	3.44	4.43
	10	0.47	1.00	0.65	0.38	0.23	0.23
				1.54	2.63	4.35	4.35
80000	5	0.28	0.63	0.59	0.24	0.17	0.10
				1.07	2.63	3.71	6.30
	10	0.94	1.93	1.25	0.75	0.46	0.26
				1.54	2.57	4.20	7.42
200000	5	0.72	1.63	0.97	0.57	0.34	0.24
				1.68	2.86	4.79	6.79
	10	2.35	4.77	2.93	1.76	1.01	0.70
				1.63	2.71	4.72	6.81
400000	5	1.44	3.00	1.78	1.07	0.62	0.45
				1.69	2.80	4.84	6.67
	10	4.69	9.50	5.57	3.43	2.01	1.35
				1.71	2.77	4.73	7.04
800000	5	2.88	6.56	3.86	2.30	1.33	0.85
				1.70	2.85	4.93	7.72
	10	9.38	19.06	11.38	6.71	4.07	2.71
				1.67	2.84	4.68	7.03

Table 2. Timings (in seconds) and speed-ups for PRRQR.

<i>m</i>	<i>n</i>	<i>p</i> = 1	<i>p</i> = 2	<i>p</i> = 4	<i>p</i> = 8	<i>p</i> = 16
20000	5	0.21	0.12	0.08	0.06	0.05
			1.75	2.63	3.50	4.20
	10	0.73	0.43	0.25	0.16	0.10
40000	5	0.40	0.24	0.13	0.09	0.07
	10	1.55	0.93	0.52	0.29	0.17
80000	5	0.75	0.47	0.26	0.25	0.16
	10	2.83	1.83	1.04	0.59	0.33
200000	5	2.02	1.09	0.61	0.37	0.27
	10	7.65	4.36	2.39	1.29	0.83
400000	5	3.86	2.08	1.19	0.70	0.50
	10	13.89	7.55	4.76	2.25	1.63
800000	5	7.76	4.15	2.35	1.40	0.94
	10	28.39	14.79	7.56	4.55	3.30
			1.92	3.76	6.24	8.60

Here T_1 is the time taken by a sequential implementation of either the QR or RRQR algorithm where the QR factorization is obtained via Householder reflections. On Table 1 we also show the timings obtained when solving the same problem with the LAPACK DGEQPF and DORGQR routines, on a single processor.

The entries on the tables show that up to four processors, the speed-up decreases for fixed m and increasing n ; afterwards, it increases. When there are few processors available, the amount of work done in each processor during the second phase, to triangularize A and compute Q as a result, is considerable and this imposes a reduction in the speed-up for a larger n ; also, increasing n leads to larger triangular blocks being exchanged between the processors.

However, as p increases, with fixed m and n , less rows are stored in each processor and the binary-tree reduction mechanism used in PQR allows for a greater level of parallelism, since more and more data exchanges are made si-

Table 3. Ratios T_H/T_p , T_G/T_p and speed-ups for PQR, with $m = 4000$.

p	$n = 5$				$n = 10$			
	T_H/T_p	T_G/T_p	S_p		T_H/T_p	T_G/T_p	S_p	
2	0.98	0.02	1.94		0.96	0.04	1.93	
4	0.92	0.08	3.74		0.86	0.14	3.49	
8	0.80	0.20	6.64		0.70	0.30	5.90	
16	0.63	0.37	10.85		0.49	0.51	8.47	

multaneously and less work is done in each processor. The data confirms that both algorithms scale very well for a large number of processors, with PRRQR having a little better performance since the updates of Q and R_1 during any retriangularization caused by a column swapping is made locally, without communication between the processors.

Table 3 shows the ratios between the time taken for the first phase (Householder triangularization, perfectly parallel) and the total time (T_H/T_p); and also for the second phase (Givens's triangularization, with communication between the processors) and the total time (T_G/T_p), for two problems of size $m = 4000$ and $n = 5$, $n = 10$, using the PQR algorithm. It is interesting to note that the first phase is responsible for nearly all the execution time and it is the main reason for the good performance of the algorithm, as it does not involve any communication between the processors; also note that that the ratio for the first phase decreases with increasing p . However, as mentioned above, increasing n means that the second phase will have a larger contribution on the execution. As the time for this phase grows by a factor $C \log_2 p$, for some value of p both ratios will be approximately equal; the speed-up attained after this value of p will be less than $p/2$.

We should mention that we have tested our implementations for some situations where even if A has full rank, a block allocated to a processor may be deficient in rank (for instance, two or more of its columns may be linearly dependent; this has also been noted in [13]). The results obtained, with regards to those test matrices, have shown that our algorithms can cope with this problem.

4 Final Remarks

We have presented a new parallel algorithm, based on a reduction strategy, for computing the QR and rank-revealing QR factorizations of large matrices $A_{m \times n}$, $m \gg n$.

The performance obtained with our actual implementations in FORTRAN90 and MPI has been shown to present a good scalability when increasing the number of processors.

We would like to note that using PQR and RRQR in our parallel implementations of the Block GMRES method has shown to provide similar performance as that presented here.

Acknowledgements

The authors would like to thank LNCC for allowing our using its computing facilities. We also thank the referees for their very useful comments which have helped improve this paper.

References

1. C.H. Bischof and G. Quintana-Ortí. Computing rank-revealing QR factorizations of dense matrices. *ACM Transactions on Mathematical Software*, 24(2):226–253, June 1998.
2. T.F. Chan. Rank revealing QR factorizations. *Linear Algebra and its Applications*, 88/89:67–82, 1987.
3. R.D. da Cunha and T.R. Hopkins. The parallel solution of triangular systems of linear equations. In Durand, M. and El Dabaghi, F., editor, *High-Performance Computing II – Proceedings of the Second Symposium on High Performance Computing*, pages 245–256, Amsterdam, October 1991. North-Holland. Also as Report No. 86, Computing Laboratory, University of Kent at Canterbury, U.K.
4. E. Elmroth and F.G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, July 2000.
5. L.V. Foster. Rank and null space calculations using matrix decomposition without column interchanges. *Linear Algebra and its Applications*, 74:47–71, 1986.
6. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 2nd edition, 1989.
7. M.T. Heath and C.H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 9:558–588, 1988.
8. G. Li and T.F. Coleman. A new method for solving triangular systems on distributed-memory message-passing multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 10:382–396, 1989.
9. J.W.H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM Journal of Scientific and Statistical Computing*, 7:1190–1211, 1986.
10. F.T. Luk. A rotation method for computing the QR-decomposition. *SIAM Journal on Scientific and Statistical Computing*, 7(2):452–459, 1986.
11. D.J. Pierce and J.G. Lewis. Sparse multifrontal rank revealing QR factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):159–180, 1997.
12. Youcef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, Boston, 1995.
13. R.B. Sidje. Alternatives for parallel Krylov subspace basis computation. *Numerical Linear Algebra with Applications*, 4(4):305–331, 1997.