New Serial and Parallel Recursive *QR* Factorization Algorithms for SMP Systems

Erik Elmroth[1] and Fred Gustavson[2]

[1] Department of Computing Science and HPC2N, Umeå University, S–901 87 Umeå, Sweden.
elmroth@cs.umu.se
[2] IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.
gustav@watson.ibm.com

Abstract. We present a new recursive algorithm for the QR factorization of an m by n matrix A. The recursion leads to an automatic variable blocking that allow us to replace a level 2 part in a standard block algorithm by level 3 operations. However, there are some additional costs for performing the updates which prohibits the efficient use of the recursion for large n. This obstacle is overcome by using a hybrid recursive algorithm that outperforms the LAPACK algorithm DGEQRF by 78% to 21% as $m = n$ increases from 100 to 1000. A successful parallel implementation on a PowerPC 604 based IBM SMP node based on dynamic load balancing is presented. For 2, 3, 4 processors and $m = n = 2000$ it shows speedups of 1.96, 2.99, and 3.92 compared to our uniprocessor algorithm.

1 Introduction

LAPACK algorithm DGEQRF requires more floating point operations than LAPACK algorithm DGEQR2, see [1]. Yet, DGEQRF outperforms DGEQR2 on a RS/6000 workstation by nearly a factor of 3 on large matrices. Dongarra, Kaufman and Hammarling, in [7], and later Bishof and Van Loan, in [3], and still later, Schreiber and Van Loan, in [9], demonstrated why this is possible by aggregating the Householder transforms before applying them to a matrix C. The result of [3] and [9] was the k way aggregating WY Householder transform, and the k way aggregating storage efficient Householder transform. In the latter, the WY representation of $Q = I - YTY^T$. Here lower trapezoidal Y is m by k and upper triangular T is k by k.

Our recursive algorithm, RGEQR3, starts with a block size k of 1 and doubles k in each step. If we would allow this to continue for a large number of columns the performance would eventually degrade as the additional floating point operations (FLOPS) grow cubically in k. Thus, to avoid this occurring, RGEQR3 should not be used until $k = n/2$. Instead, we propose to use a hybrid recursive algorithm, RGEQRF, which is a slight modification of a standard level 3 block algorithm in that it calls RGEQR3 for factorizing block columns. Hence, RGEQRF is a modified version of Alg. 4.2 of Bishof and Van Loan, in [3] and RGEQR3 is a recursive level 3 counterpart of their Alg. 4.1.

In Section 2 we describe our new recursive serial algorithms RGEQR3 and RGEQRF and compare their performance with LAPACK's serial algorithms DGEQR2 and DGEQRF. In Section 3 we describe our new parallel recursive algorithm. Section 3.1 shows performance results of near perfect speedups for large matrices of size m = n = 2000. We also describe an anomaly (10% performance loss on one processor) with the `!SMP$ do parallel` construct.

2 Recursive QR Factorization

In our recursive algorithm, the QR factorization of an $m \times n$ matrix A

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix},$$

is initiated by a recursive factorization of the left-hand side $\lfloor n/2 \rfloor$ columns, i.e.,

$$Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}.$$

The remaining part of the matrix is updated,

$$\begin{pmatrix} R_{12} \\ \tilde{A}_{22} \end{pmatrix} \longleftarrow Q_1^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}, \tag{1}$$

and $\tilde{A}_{22}$ is recursively factorized into $Q_2 R_{22}$. The recursion stops when the matrix to be factorized consists of a single column. This column is factorized using an elementary Householder transformation. We use the storage efficient WY form, i.e., $Q = I - YTY^T$, to represent the orthogonal matrices, and the update by Q_1^T in (1) is performed as a series of matrix multiplications with general and triangular matrices (i.e., by calls to level 3 BLAS DGEMM and DTRMM).

In Figure 1, we give the details of the recursive algorithm, called RGEQR3.

RGEQR3 $A(1\!:\!m, 1\!:\!n) = (Y, R, T)$! Note, Y and R replaces A
if $(n = 1)$ **then**
  compute Householder transform $Q = I - \tau u u^T$ such that $QA = (x, 0)^T$
  return (u, x, τ) ! Note, $u = Y$, $x = R$, and $\tau = T$
else
  $n1 = n/2$ and $j1 = n1 + 1$
  call RGEQR3 $A(1\!:\!m, 1\!:\!n1) = (Y_1, R_1, T_1)$ where $Q_1 = I - Y_1 T_1 Y_1^T$
  compute $A(1\!:\!m, j1\!:\!n) = Q_1^T A(1\!:\!m, j1\!:\!n)$
  call RGEQR3 $A(j1\!:\!m, j1\!:\!n) = (Y_2, R_2, T_2)$ where $Q_2 = I - Y_2 T_2 Y_2^T$
  compute $T_3 = T(1\!:\!n1, j1\!:\!n) = -T_1 (Y_1^T Y_2) T_2$.
  set $Y = (Y_1, Y_2)$! Y is m by n unit lower trapezoidal
  return (Y, R, T), where $R = \begin{pmatrix} R_1 & A(1\!:\!n1, j1\!:\!n) \\ 0 & R_2 \end{pmatrix}$ and $T = \begin{pmatrix} T_1 & T_3 \\ 0 & T_2 \end{pmatrix}$
endif

Fig. 1. Recursive QR factorization routine RGEQR3.

We assume A is m by n where $m \geq n$. In the "else" clause there are two recursive calls, one on matrix $A(1{:}m, 1{:}n1)$, the other on matrix $A(j1{:}m, j1{:}n)$, and the computations $Q_1^T A(1{:}m, j1{:}n)$ and $-T_1(Y_1^T Y_2)T_2$. These two computations consist mostly of calls to either DGEMM or DTRMM. Our implementation of RGEQR3 is made in standard Fortran 77 which requires the explicit handling of the recursion.

In Figure 2 we give annotated descriptions of algorithms DGEQRF and DGEQR2 of LAPACK. See [1] for full details. The routine DGEQRF calls DGEQR2 which is a level 2 version of DGEQRF. In the annotation we have assumed $m \geq n$.

```
DGEQRF(m, n, A, τ, work)
do j = 1, n, nb      ! nb is the block size
   jb = min(n − j + 1, nb)
   call DGEQR2(m − j + 1, jb, A(j, j), τ(j) )
   if (j + jb .LE. n) then
      compute T(1:jb, 1:jb) in work via a call to DLARFT
      compute (I − Y T^T Y^T)A(j:m, j + jb:n) using work and T via
      a call to DLARFB
   endif
enddo
```

```
DGEQR2(m, n, A, τ)
do j = 1, n
   compute Householder transform Q(j) = I − τuu^T such that
   Q(j)^T A(j:m, j) = (x, 0)^T via a call to DLARFG
   if (j .LT. n) then
      apply Q(j)^T to A(j:m, j + 1:n) from the left by calling DLARF
   endif
enddo
```

Fig. 2. DGEQRF and DGEQR2 of LAPACK.

2.1 Remarks on the recursive algorithm RGEQR3

The algorithm RGEQR3 requires more floating point operations than algorithm DGEQRF which requires more floating point operations than DGEQR2. Dongarra, Kaufman and Hammarling, in [7], showed how to increase performance by increasing the FLOP count when they aggregated two Householder transforms before they were applied to a matrix C. The computation they considered was

$$C = Q_1 Q_2 C, \tag{2}$$

where C is m by n and Q_1 and Q_2 are Householder matrices of the form $I - \tau_i u_i u_i^T$, $i = 1, 2$. Their idea was to better use high speed vector operations and thereby gain a decrease in execution time. Bishof and Van Loan, in [3],

generalized (2) by using the WY transform. They represented the product of k Householder transforms $Q_i, i = 1, \ldots, k$, as

$$Q = Q_1 Q_2 \cdots Q_k = I - WY^T. \tag{3}$$

They used (3) to compute $Q^T C = C - YW^T C$. Later on, Schreiber and Van Loan, in [9], introduced a storage-efficient WY representation for Q:

$$Q = Q_1 Q_2 \cdots Q_k = I - YTY^T, \tag{4}$$

where T is an upper triangular k by k matrix. In both of these cases performance was enhanced by increasing the FLOP count. Here the idea was to replace the matrix-vector type computations by matrix-matrix type computations. The decrease in execution time occurred because the new code, despite requiring more floating point operations, made better use of the memory hierarchy.

In (4) Y is a trapezoidal matrix consisting of k consecutive Householder vectors, $u_i, i = 1, \ldots, k$. The first component of each u_i is one, where the one is implicit and not stored. These vectors are scaled with τ_i. For $k = 2$, the T of equation (4), is

$$T = \begin{pmatrix} \tau_1 & -\tau_1 u_1^T u_2 \tau_2 \\ 0 & \tau_2 \end{pmatrix}. \tag{5}$$

Suppose $k_1 + k_2 = k$ and T_1 and T_2 are the associated triangular matrices in (4). We have

$$Q = (Q_1 \cdots Q_{k_1})(Q_{k_1+1} \cdots Q_k) = (I - Y_1 T_1 Y_1^T)(I - Y_2 T_2 Y_2^T) = I - YTY^T, \tag{6}$$

where $Y = (Y_1, Y_2)$ is formed by concatenation. Thus a generalization of (5) is

$$T = \begin{pmatrix} T_1 & -T_1 Y_1^T Y_2 T_2 \\ 0 & T_2 \end{pmatrix}, \tag{7}$$

which is essentially a level 3 formulation of (5), (6).

Schreiber and Van Loan and LAPACK's DGEQRF compute (6) via a bordering technique consisting of a series of level 2 operations. For each $k_1 = 1, \ldots, k-1$, k_2 is chosen to be 1. However, as (6) and (7) suggests, $Q = I - YTY^T$ can be done recursively as a series of $k-1$ matrix-matrix computations. Also, the FLOP count in doing the T computation in (6) by this matrix-matrix computation is the same as the FLOP count of doing the bordering computation to compute T.

Algorithm RGEQR3 can be viewed as starting with with $k = 1$ and doubling k until $k = n/2$. If this doubling was allowed to continue performance would drastically degrade due to the cubically increasing FLOP count in the variable k. To avoid this occurring RGEQR3 should *not* be used for large n. Instead, the current algorithm, DGEQRF, should be revised and used with RGEQR3. In Figure 3 we give our hybrid recursive algorithm which we name RGEQRF. The routine DQTC applies $Q^T = I - YT^T Y^T$ to a matrix C as a series of DGEMM and DTRMM operations, i.e., $Q^T C = C - Y(T^T(Y^T C))$. Note that RGEQRF has no call to LAPACK routine DLARFT. The DLARFT routine computes the upper triangular matrix T via level 2 calls. Instead, routine RGEQR3 computes T via our matrix-matrix approach in addition to computing τ, Y and R.

RGEQRF($m, n, A, lda, \tau, work$)
do $j = 1, n, nb$! nb is the block size
$jb = \min(m - j + 1, nb)$
call RGEQR3 $A(j : m, j + jb - 1) = (Y, R, T)$! note T is stored in work
if $(j + jb$ .LE. $n)$ **then**
compute $(I - YT^TY^T)A(j{:}m, j + jb{:}n)$ using *work* and T via
a call to DQTC
endif
enddo

Fig. 3. Hybrid algorithm RGEQRF.

2.2 Uniprocessor performance results

Figures 4 and 5 show that RGEQRF outperforms DGEQRF of LAPACK on one processor by 20% for large matrices and up to 78% for small matrices. The gain in performance is partially explained by the increased performance of RGEQR3 compared to DGEQR2. Additional benefits occur because the faster RGEQR3 leads to an increased optimal block size in RGEQRF compared to DGEQRF, which in turn improves the performance of the level 3 BLAS updates in DQTC.

The performance results are obtained on a 4-way 112 MHz PowerPC 604 based IBM SMP High Node, using IBM XL Fortran 5.1.

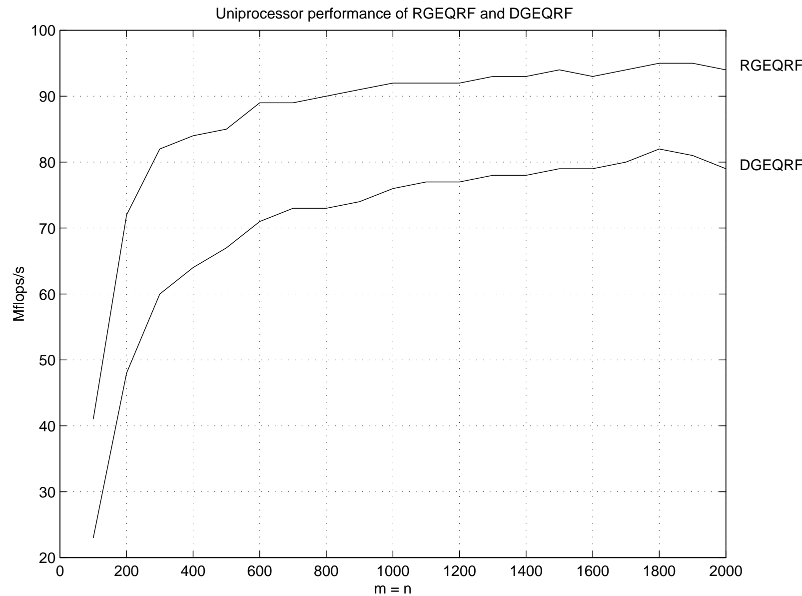

Fig. 4. Uniprocessor performance results in *Mflops* for the hybrid recursive algorithm RGEQRF and DGEQRF of LAPACK, using optimal block sizes 48 and 32, respectively.

$m = n$	100	200	300	400	500	600	700	800	900	1000	1500	2000
Ratio	1.78	1.50	1.37	1.31	1.27	1.25	1.22	1.23	1.23	1.21	1.19	1.19

Fig. 5. Ratio of uniprocessor performance results for RGEQRF and DGEQRF.

3 Parallel Hybrid Recursive QR Factorization

In the parallel version of RGEQRF, a "node program", NODEQR, is called once for each available processor, as shown in Alternative 1 in Figure 6. In the NODEQR routine, each processor repeatedly performs three operations. It finds a new block to update through a call to routine GETJOB, it updates that block through a call to routine DQTC, and it factorizes the block through a call to RGEQR3, if required. The processors returns from NODEQR when the QR factorization is completed. When using one processor, one might argue that Alternative 2 in Figure 6 should be equivalent to Alternative 1. However, as we will see in Section 3.1, the `parallel do` construction appears to introduce a significant amount of system overhead when run on one processor.

Alternative 1

```
!SMP$ do parallel
      do me = 1, NUMPROCS
         call NODEQR(me, A, ...)
      end do
```

Alternative 2

```
      if (NUMPROCS .EQ. 1) then
         call NODEQR(me, A, ...)
      end if
```

Fig. 6. Code segments for calling the routine NODEQR.

The critical part of the parallel QR factorization is done by the GETJOB routine. GETJOB implements a distributed version of the pool-of-tasks principle [4] for dynamically assigning work to the processors. The GETJOB algorithm is sketched in Figure 7, and is described in the next paragraph.

In the pool-of-tasks implementation only one critical section is entered per job (to find a new task, to tell what submatrix is reserved for this task, to update variables after the task is completed, etc). There are three variables associated with each processor that keep track of an iteration index and two indices equal to the first and last column the processor is working on. The size of the blocks to update depends on the size of the remaining matrix and the number of processors available. Near the end of the computation, a form of adaptive blocking is used [2]. As the remaining problem size decreases, both the number of processors and the sizes of the blocks being updated are decreased. A processor that will both update and factorize a block will only update the columns it will factor. There is no fixed synchronization in the algorithm; it is an asynchronous algorithm. This means that sometimes processors may be working on different iterations. Therefore more than one T matrix is needed. At least three T matrices are required to avoid contention. However, using 4 appears to further improve performance.

```
while (I have not yet found a new task, i.e., a new block to update)
    Enter Critical section
    If I did factor in my last task, update global variables
    If (remaining problem is too small for the current # processors) then
          Update global variables and terminate
    else
          Find the next matrix block to update
          Test if it is OK to start working on this block, i.e. test that:
                - no one is writing on any column in this block
                - the block I will read is computed
                - if I will factor: is it safe to overwrite one of the T matrices?
          If (it is OK to start working on this block) then
                Update global variables
          else
                Update variables to show that no block is reserved
          endif
    endif
    Leave Critical section
end while
```

Fig. 7. Algorithm GETJOB does the pool-of-tasks implementation.

3.1 Parallel Performance Results

As mentioned above, there seem to be a significant amount of system overhead introduced by the `parallel do` construction in Figure 6 (Alternative 1). In performance comparisons on one processor, Alternative 2 outperforms Alternative 1 by around 10% for large matrices. Since the `parallel do` construction has to be used for running on more than one processor, this is a bit discouraging, i.e., it suggests that $4 \times 90\%$ of the best uniprocessor performance is the best we can achieve on 4 processors. With the Alternative 2 approach, the parallel RGEQRF shows approximately the same performance as the serial variant, indicating that the overhead costs for additional blocking etc is negligible. The only significant extra cost in the parallel algorithm is associated with the 10% overhead of the `parallel do` construction.

Fortunately, it seems that the amount of overhead for the `parallel do` construction is reduced when the number of processors is increased. Performance results for 1 to 4 processors are shown in Table 1. *Mflops* are given for the Alternative 1 approach. The speedup is shown both relative to the uniprocessor results using the Alternative 1 approach, denoted S_1, and using the Alternative 2 approach, denoted S_2.

The speedup S_2 which is relative to the best uniprocessor implementation of the algorithm, i.e., using Alternative 2 in Figure 6, is almost optimal for large matrices. The fact that the speedup increases by more than one when adding another processor (from one to two and from two to three processors) indicates that the amount of overhead per processor is decreased when the number of processors is increased. We think the speedup S_1 (up to 4.29) is remarkable and the speedup S_2 (up to 3.92) is excellent.

Table 1. Parallel performance results and speedup for PRGEQRF.

	#proc = 1			#proc = 2			#proc = 3			#proc = 4		
$m = n$	*Mflops*	S_1	S_2	*Mflops*	S_1	S_2	*Mflops*	S_1	S_2	*Mflops*	S_1	S_2
100	40	*1.00*	*0.95*	55	*1.38*	*1.31*	55	*1.38*	*1.31*	54	*1.35*	*1.29*
200	65	*1.00*	*0.96*	99	*1.52*	*1.46*	126	*1.94*	*1.85*	139	*2.14*	*2.04*
300	74	*1.00*	*0.94*	128	*1.73*	*1.62*	170	*2.30*	*2.15*	201	*2.72*	*2.54*
400	77	*1.00*	*0.93*	145	*1.88*	*1.75*	194	*2.52*	*2.34*	235	*3.05*	*2.83*
500	78	*1.00*	*0.92*	149	*1.91*	*1.75*	209	*2.68*	*2.46*	255	*3.27*	*3.00*
600	81	*1.00*	*0.92*	161	*1.99*	*1.83*	227	*2.80*	*2.58*	288	*3.56*	*3.27*
700	81	*1.00*	*0.91*	166	*2.05*	*1.87*	238	*2.94*	*2.67*	307	*3.79*	*3.45*
800	82	*1.00*	*0.91*	169	*2.06*	*1.88*	248	*3.02*	*2.76*	318	*3.88*	*3.53*
900	83	*1.00*	*0.91*	173	*2.08*	*1.90*	253	*3.05*	*2.78*	330	*3.98*	*3.63*
1000	83	*1.00*	*0.90*	175	*2.11*	*1.90*	258	*3.11*	*2.80*	334	*4.02*	*3.63*
1100	83	*1.00*	*0.91*	178	*2.14*	*1.96*	262	*3.16*	*2.88*	341	*4.11*	*3.75*
1200	85	*1.00*	*0.92*	179	*2.11*	*1.95*	267	*3.14*	*2.90*	349	*4.11*	*3.79*
1300	84	*1.00*	*0.89*	181	*2.15*	*1.93*	268	*3.19*	*2.85*	352	*4.19*	*3.74*
1400	84	*1.00*	*0.88*	179	*2.13*	*1.88*	271	*3.23*	*2.85*	352	*4.19*	*3.71*
1500	84	*1.00*	*0.90*	183	*2.18*	*1.97*	273	*3.25*	*2.94*	357	*4.25*	*3.84*
1600	84	*1.00*	*0.90*	182	*2.17*	*1.96*	275	*3.27*	*2.96*	360	*4.29*	*3.87*
1700	85	*1.00*	*0.90*	183	*2.15*	*1.95*	275	*3.24*	*2.93*	362	*4.26*	*3.85*
1800	86	*1.00*	*0.91*	186	*2.16*	*1.96*	278	*3.23*	*2.93*	364	*4.23*	*3.83*
1900	85	*1.00*	*0.90*	183	*2.15*	*1.95*	279	*3.28*	*2.97*	365	*4.29*	*3.88*
2000	85	*1.00*	*0.91*	182	*2.14*	*1.96*	278	*3.27*	*2.99*	365	*4.29*	*3.92*

4 Conclusions

We have shown that a significant increase of performance can be obtained by replacing the level 2 algorithm in a LAPACK-style block algorithm for QR factorization by a recursive algorithm that essentially performs level 3 operations.

Recursion has previously been successfully applied to LU factorization [8, 10]. For the LU case, no extra FLOPS are introduced from recursion and the performance significantly exceeds that of LAPACK block algorithms. In this contribution we show that despite the extra flops, the hybrid QR factorization benefits even more from using recursion than the LU factorization does.

The parallel speedup compares well with previously published results for QR factorization routines for shared memory systems. The algorithm presented here show better speedup then what is presented for the IBM 3090 VF/600J in [6] and the Alliant fx2816 in [5] and it is similar to what is presented for IBM 3090 VF/600J in [5].

5 Acknowledgements

We thank Olov Gustavsson for sharing his experiences in parallel program development for the PowerPC 604 based IBM SMP High Nodes which, e.g., led

to our decision to only use one critical section per iteration due to the overhead associated with the implementations of critical sections.

This research was conducted using the resources of High Performance Computing Center North (HPC2N).

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, S. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* SIAM, Philadelphia, 1994.
2. C. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3:193–208, 1989.
3. C. Bischof and C. Van Loan. The *WY* representation for products of householder matrices. *SIAM J. Scientific and Statistical Computing*, 8(1):s2–s13, 1987.
4. A. Chalmers and J. Tidmus. *Practical Parallel Processing.* International Thomson Computer Press, UK, 1996.
5. K. Dackland, E. Elmroth, and B. Kågström. A ring-oriented approach for block matrix factorizations on shared and distributed memory architectures. In R. F. Sincovec et al, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 330–338, Norfolk, 1993. SIAM Publications.
6. K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan. Parallel block matrix factorizations on the shared memory multiprocessor IBM 3090 VF/600J. *International Journal of Supercomputer Applications*, 6(1):69–97, 1992.
7. J. Dongarra, L. Kaufman, and S. Hammarling. Squeezing the most out of eigenvalue solvers on high performance computers. *Lin. Alg. and its Applic.*, 77:113–136, 1986.
8. F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.
9. R. Schreiber and C. Van Loan. A storage efficient *WY* representation for products of householder transformations. *SIAM J. Scientific and Statistical Computing*, 10(1):53–57, 1989.
10. S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM J. Matrix. Anal. Appl.*, 18(4):1065–1081, 1997.