Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance

Erik Elmroth* Fred Gustavson†

January 12, 2000

Abstract

We present new recursive serial and parallel algorithms for QR factorization of an m by n matrix. They improve performance. The recursion leads to an automatic variable blocking and it also replaces a Level 2 part in a standard block algorithm by Level 3 operations. However, there are significant additional costs for creating and performing the updates which prohibits the efficient use of the recursion for large n. We present a quantitative analysis of these extra costs. This analysis leads us to introduce a hybrid recursive algorithm that outperforms the LAPACK algorithm DGEQRF by around 20% for large square matrices and up to almost a factor of 3 for tall thin matrices. Uniprocessor performance results are presented for two IBM RS/6000 SP nodes; a 120 MHz IBM POWER2 node and one processor of a 4-way 332 MHz IBM PPC604e SMP node. The hybrid recursive algorithm reaches over 90% of the theoretical peak performance of the POWER2 node. Compared to standard type block algorithms, the recursive approach also shows a significant advantage in the automatic tuning obtained from its automatic variable blocking. A successful parallel implementation on a 4-way 332 MHz IBM PPC604e SMP node based on dynamic load balancing is presented. For 2, 3, 4 processors it shows speedups up to 1.97, 2.99, and 3.97.

Keywords: Serial and parallel QR factorization, recursion, automatic variable blocking, SMP system, dynamic load balancing.

1 Introduction

LAPACK algorithm DGEQRF requires more floating point operations than LAPACK algorithm DGEQR2, see [2]. Yet, DGEQRF outperforms DGEQR2 on a RS/6000 workstation by nearly a factor of 3 on large matrices. Dongarra, Kaufman and Hammarling, in [9], and later Bishof and Van Loan, in [4], and

*Department of Computing Science and HPC2N, Umeå University, SE–901 87 Umeå, Sweden. `elmroth@cs.umu.se`

†IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A. `gustav@watson.ibm.com`

still later, Schreiber and Van Loan, in [15], demonstrated why this is possible by aggregating the Householder transforms before applying them to a matrix C. The result of [4] and [15] was the k way aggregating WY Householder transform, and the k way aggregating storage efficient Householder transform. In the latter, the aggregated representation of $Q = I - YTY^T$. Here lower trapezoidal Y, is m by k consisting of k Householder vectors, and upper triangular T is k by k.

Our recursive algorithm, RGEQR3, starts with a block size $k = 1$ 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. The cost of RGEQR3 on an n by n matrix is $(13/6)n^3 + \ldots$ whereas DGEQR2 cost is $(4/3)n^3 + \ldots$. 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 modification of a standard Level 3 block algorithm in that it calls RGEQR3 for factorizing block columns instead of calling DGEQR2 and DLARFT.[1] Hence, RGEQRF is a modified version of Algorithm 4.2 of Bishof and Van Loan, in [4] and RGEQR3 is a recursive Level 3 counterpart of their Algorithm 4.1. This work is a continuation of the work presented in [10].

In Section 2 we describe our new recursive serial algorithms RGEQR3 and RGEQRF and give a proof of correctness. In Section 2.1 we discuss aspects of increasing the FLOP count in order to gain performance. In Section 2.2 we give an explanation why the FLOP count increases for QR factorization when one goes from a Level 2 to a Level 3 factorization. Next we derive FLOP counts for LAPACK algorithms DGEQR2 and DGEQRF and our new algorithms RGEQR3 and RGEQRF. These counts are functions of the three parameters m, n, and k, the block size. Based on relations between these various counts one can devise and tune various QR factorization algorithms; the above four are examples. Specifically, we demonstrate the correctness of the new recursive algorithm by using mathematical induction on $k = \log_2 n$, $k = 0, 1, 2, \ldots$. Additionally, a quantitative analysis is presented that details how the FLOP counts of the various standard and new recursive algorithms relate to one another. At the end of Section 2.2 we detail savings for RGEQR3 in computing the T matrix when one does not have to later update with T. These savings are especially important for tall thin matrices. We also discuss alternative approaches for performing the updates of the matrix, including a description of the routine DQTC (equivalent to DLARFB) used in our implementations.

Uniprocessor performance results for square and tall thin matrices are presented for the 120 MHz IBM POWER2 node and the 332 MHz IBM PPC 604e node in Section 3. Roughly speaking, the algorithm RGEQR3 is α times faster than DGEQRF where α decreases from 3 to 1.2 as n ranges from 50 to m and $m \geq 300$. We remark that significantly more effort was needed to tune the parameters for DGEQRF than what was needed for RGEQRF, since DGEQRF has two parameters that are to be set (defining a 2-dimensional parameter space), whereas only one parameter is to be set in RGEQRF. The fact that less tuning

[1] Routines starting with DLA... are LAPACK auxiliary routines.

is needed to obtain optimal performance is another feature of the automatic variable blocking of the recursive algorithm. We also mention when recursion fails to produce good performance and its remedy.

In Section 4 we describe our new parallel recursive algorithm. It is related to the LU factorization algorithm described in [1] and the dynamic load balancing versions of LU and *Cholesky* factorization in [8]. The algorithm is based on a dynamic load balancing strategy, implemented using the pool-of-task principle where each processor enters a critical section to assign itself more work as soon as it has completed its last task. This process is fully asynchronous as there are no fixed synchronization points. The amount of work performed in each task is large enough to make overhead in the work distribution process negligible. Section 5 shows performance results for the parallel algorithm on 1, 2, 3, and 4 processors of a 4-way 332 MHz IBM PPC604e SMP node. The uniprocessor performance of the parallel algorithm is basically the same as for the serial algorithm. The parallel results show near perfect speedups, up to 1.97, 2.99, and 3.97 for 2, 3, and 4 processors, respectively.

2 Recursive QR Factorization

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

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}, \tag{1}$$

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

$$Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}. \tag{2}$$

The remaining part of the matrix is updated,

$$\begin{pmatrix} R_{12} \\ \tilde{A}_{22} \end{pmatrix} \longleftarrow Q_1^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}, \tag{3}$$

and then $\tilde{A}_{22}$ is recursively factorized

$$\tilde{Q}_2 R_{22} = \tilde{A}_{22}. \tag{4}$$

The recursion stops when the matrix to be factorized consists of a single column. This column is factorized by applying an elementary Householder transformation to it.

Proof of correctness. Our method of proof will be mathematical induction. We only consider the case $m \geq n$. The recursion take place on the column dimension n. We want to prove the correctness for $n = 1, 2, \ldots$. However, recursion breaks the problem into nearly two equal pieces, $n_1 = \lfloor n/2 \rfloor$ and $n_2 = n - n_1$. This suggests we use mathematical induction on $k = \log_2 n$, $k = 0, 1, \ldots$. Based on this observation we have the following form of mathematical induction. Suppose the result is true for $1 \le n \le 2^k$. Then we establish the results for all j, $2^k < j \le 2^{k+1}$. Additionally, we need to establish the result for (m, n) where $n = 1$.

Now we give the proof: For $n = 1$ we compute the Householder reflector $H = I - \tau u u^T$ where u is computed from $A(1:m,1)$. Thus the result is true for $n = 1$. Now suppose the result is true for $1 \le n \le 2^k$. Let $2^k < j \le 2^{k+1}$, $j_1 = \lfloor j/2 \rfloor$ and $j_2 = j - j_1$. Since $j > 1$ we do the computations indicated by equations (2), (3), and (4) in that order.

Equation (2) is satisfied by the induction hypothesis; Equation (3) is a calculation and equation (4) is satisfied by the induction hypothesis. Using equations (2)–(4), we want to show that equation (1) is true, where $Q = Q_1 Q_2$ and Q, Q_1, and Q_2 are $m \times m$. That Q is orthogonal follows by induction, using the induction hypothesis that Q_1 and Q_2 are orthogonal and the facts that Q_1 and Q_2 are orthogonal Householder transformations for the cases $n_1 = 1$ and $n_2 = 1$, respectively, and the fact that the product of two orthogonal matrices is an orthogonal matrix.

Partition Q_1 and Q_2 as follows:

$$Q_1 = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} \quad Q_2 = \begin{pmatrix} I & 0 \\ 0 & \tilde{Q}_2 \end{pmatrix}, \tag{5}$$

where $\tilde{Q}_2$ is $m - n_1$ by $m - n_1$. Since Q is orthogonal, $Q^T Q = I$ and it is sufficient to show that $Q^T A = R$.

Now,

$$Q_1^T A = \begin{pmatrix} Q_{11}^T A_{11} + Q_{21}^T A_{21} & Q_{11}^T A_{12} + Q_{21}^T A_{22} \\ Q_{12}^T A_{11} + Q_{22}^T A_{21} & Q_{12}^T A_{12} + Q_{22}^T A_{22} \end{pmatrix}, \tag{6}$$

and by substituting (2) and (3) into (6) we get

$$Q_1^T A = \begin{pmatrix} R_{11} & R_{12} \\ 0 & \tilde{A}_{22} \end{pmatrix}. \tag{7}$$

Now $Q^T A = Q_2^T (Q_1^T A)$ becomes equal to R by substituting (7) for $Q_1^T A$, then carrying out the multiplication by Q_2^T (see (5)) and finally substituting R_{22} for $\tilde{Q}_2^T \tilde{A}_{22}$. (This latter substitution is valid by equation (4)). $\square$

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 (3) 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 our recursive algorithm, called RGEQR3.

We assume A is m by n where $m \ge n$. In the "else" clause there are two recursive calls, one on matrix $A(1:m, 1:n_1)$, the other on matrix $A(j_1:m, j_1:n)$, and the computations $Q_1^T A(1:m, j_1:n)$ 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.

RGEQR3 $A(1{:}m, 1{:}n) = (Y, R, T)$! Note, Y and R replaces A, $m \geq n$
if $(n = 1)$ **then**
  compute Householder transform $Q = I - \tau uu^T$ such that $QA = (x, 0)^T$
  return (u, x, τ) ! Note, $u = Y$, $x = R$, and $\tau = T$
else
  $n_1 = \lfloor n/2 \rfloor$ and $j_1 = n_1 + 1$
  call RGEQR3 $A(1{:}m, 1{:}n_1) = (Y_1, R_1, T_1)$ where $Q_1 = I - Y_1 T_1 Y_1^T$
  compute $A(1{:}m, j_1{:}n) = Q_1^T A(1{:}m, j_1{:}n)$
  call RGEQR3 $A(j_1{:}m, j_1{:}n) = (Y_2, R_2, T_2)$ where $Q_2 = I - Y_2 T_2 Y_2^T$
  compute $T_3 = T(1{:}n_1, j_1{:}n) = -T_1(Y_1^T Y_2)T_2$.
  set $Y = (Y_1, Y_2)$! Y is m by n unit lower trapezoidal
  return (Y, R, T), where
$$R = \begin{pmatrix} R_1 & A(1{:}n_1, j_1{:}n) \\ 0 & R_2 \end{pmatrix} \text{ and } T = \begin{pmatrix} T_1 & T_3 \\ 0 & T_2 \end{pmatrix}$$
endif

Figure 1: Recursive QR factorization routine RGEQR3.

Algorithm RGEQR3 can be proved correct by following and modifying the above correctness proof. In this regard we note that RGEQR3 uses the storage efficient representation $Q = I - YTY^T$ and so T also needs to be additionally computed. Furthermore, RGEQR3 computes T recursively. The modification in the proof would include the correctness of the recursive T computation. Since it also follows in a similar way we omit it.

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

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 [9], 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

$$C = Q^T C, \tag{8}$$

where C is m by n, $Q = Q_1 Q_2$, 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 [4], generalized (8) by using the WY transform. They represented the product of k Householder transforms $Q_i, i = 1, \ldots, k$, as

$$Q = Q_1 Q_2 \cdots Q_k = I - WY^T. \tag{9}$$

```
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 − τ u u^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
```

Figure 2: DGEQRF and DGEQR2 of LAPACK.

They used (9) to compute $Q^T C = C - YW^T C$. Later on, Schreiber and Van Loan, in [15], introduced a storage-efficient WY representation for Q:

$$Q = Q_1 Q_2 \cdots Q_k = I - YTY^T, \tag{10}$$

where T is an upper triangular k by k matrix. In all three 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 (9) and (10) 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 hence is not stored. These vectors are scaled with τ_i. For $k = 2$, the T of equation (10), is

$$T = \begin{pmatrix} \tau_1 & -\tau_1 u_1^T u_2 \tau_2 \\ 0 & \tau_2 \end{pmatrix}. \tag{11}$$

Suppose $k_1 + k_2 = k$ and T_1 and T_2 are the associated triangular matrices in (10). We have

$$Q = (Q_1 \cdots Q_{k_1})(Q_{k_1+1} \cdots Q_k) = (I - Y_1 T_1 Y_1^T)(I - Y_2 T_2 Y_2^T) = I - YTY^T, \tag{12}$$

where $Y = (Y_1, Y_2)$ is formed by concatenation. Thus a generalization of (11) is

$$T = \begin{pmatrix} T_1 & -T_1 Y_1^T Y_2 T_2 \\ 0 & T_2 \end{pmatrix}, \tag{13}$$

which is essentially a Level 3 formulation of (11), (12).

Schreiber and Van Loan and LAPACK's DGEQRF compute (12) (by LAPACK algorithm DLARFT) 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 (12) and (13) 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 (12) by this matrix-matrix computation is the same as the FLOP count of doing the bordering computation to compute T (See Section 2.2.)

Algorithm RGEQR3 can be viewed as starting 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 (see Section 2.2 for details). To avoid this occurring RGEQR3 should *not* be used for large n. Instead, the LAPACK 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 described in Sections 2.2 and 2.3 applies $Q^T = I - YT^TY^T$ to a matrix C as a series of DGEMM and DTRMM operations.

RGEQRF($m, n, A, lda, \tau, work$)
do $j = 1, n, nb$! nb is the block size
    $jb = \min(n - j + 1, nb)$
    call RGEQR3 $A(j:m, j+jb-1) = (Y, R, T)$! 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

Figure 3: Hybrid algorithm RGEQRF.

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 Level 3 (matrix-matrix) approach in addition to computing τ, Y and R. DGEQRF calls DGEQR2 to compute τ, Y and R; there is no need to compute T and so it is not done. However, the "if" clause of RGEQRF is not invoked after the last and sometimes only column block $A(j:m, j:j+jb-1)$ is factored (this occurs when and only when $j+jb-1 = n$). In that case, some of the Level 3 T computations can be avoided. An example is given in Figure 4 where matrices $T_i, i = 0, \ldots, 4$ need to be computed whereas zero matrices $Z_i, i = 0, \ldots, 3$ are not needed and hence, their computation can be avoided. Our algorithm RGEQR3 has an additional parameter ISW (a switch or flag), which when set will avoid the computation of the Z_i matrices. ISW is set on in RGEQRF just before the last column block is factored. In RGEQR3, when ISW is set on, the T_3 computation is avoided when and only when T_3's right boundary is equal to n. In figures 1 and 3 we did *not* include the switch logic as it was a detail that would detract from clarity of these algorithms. It should be clear from the description above what modifications are necessary. In Section 2.2 we detail the FLOP count saving $S(m,n)$ when ISW is set on. All performance tests have been made with ISW set on.

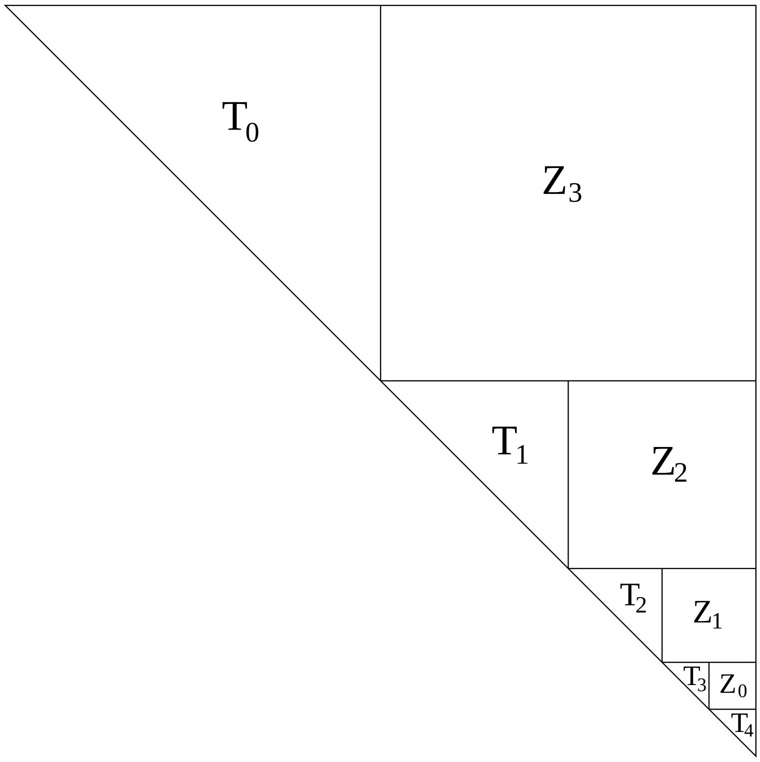

Figure 4: The matrix T, with T_i, $i = 0, \ldots, 4$ representing blocks that need to be generated and Z_i, $i = 0, \ldots, 3$, representing blocks that are not needed.

2.2 Comparative FLOP counts for algorithms DGEQR2, DGEQRF, RGEQRF, and RGEQR3

Some dense linear algebra algorithms have the same FLOP counts for both Level 2 and Level 3 versions. General *LU* factorization and *Cholesky* factorization are two examples. Others, *QR* factorization is an example, exhibit increasing FLOP count for their Level 2 and Level 3 versions as a function of increasing block size. We briefly describe why this is so. Let L be the lower triangular factor of LU factorization with partial pivoting of a general matrix, i.e., $LU = PA$. In LAPACK, the routine that produces L, U and P, given A is called DGETRF. One can represent L as the product of n rank one corrections of the identity matrix, i.e.,

$$L = L_1 L_2 \cdots L_n,$$

where $L_i = I + l_i e_i^T$. Here, vector $l_i^T = (0, \ldots, 0, 1, l_{i+1,i}, \ldots, l_{m,i})$ is the ith column of L. The point that we make here is that the product of the n elementary matrices L_i requires no arithmetic to produce L, i.e., the product L is only a concatenation of the n vectors l_i, $1 \leq i \leq n$ (see [5] for details). On the other hand, $Q = Q_1 Q_2 \cdots Q_n$, where each $Q_i = I - \tau_i u_i u_i^T$ is *not* equal to the concatenation of the n Q_i matrices. To form Q we must multiply these n Q_i matrices together. The actual representation we use is $Q = I - YTY^T$ and the extra work required to produce Q is the work needed to produce the T matrix.

In the Level 2 implementations DGETF2 and DGEQR2 we only work with matrices L_i and Q_i. In the Level 3 implementations DGETRF and DGEQRF we work with block matrices L and Q. It is clear from the discussion above that the Level 3 implementation of general LU factorization requires no additional FLOPS as L can be formed from the L_i by concatenation whereas the Level 3 implementation of general QR factorization requires additional FLOPS, namely those needed to produce T in the representation $Q = I - YTY^T$.

Our recursive algorithms RGEQRF and RGEQR3 also exhibit increasing FLOP counts. Furthermore, the FLOP count of RGEQRF is greater than the FLOP count of DGEQRF. In this section we compute FLOP counts of these four algorithms. Specifically, we derive FLOP counts FG, F, FT, as functions of m and n, and $FB(m, n, k)$. The four functions refer to LAPACK auxiliary routines DLAR{FG, F, FT, FB}. These routines work on a matrix of size m by n; DLARFB computes $Q^T C$ where C is m by n and Q is m by k. Additionally, we compute FLOP count functions $T(m, k)$, $Y(m, k)$, $R(m, k)$, $\mathit{Diff}(k)$ and $M(m, k)$, $S(m, k)$. Here we use k instead of n. These latter six functions relate RGEQR3 and RGEQRF to DGEQR2 and DGEQRF. The function $T(m, k) = FT(m, k)$ computes the FLOP count of producing the T matrix which we need to compute the block matrix $Q = I - YTY^T$. $Y(m, k)$ is the cost of doing the $k-1$ update computations $Q^T C$ in RGEQR3. We were unable to find an explicit solution to $Y(m, k)$ and so computed $W(m, k) (= Y(m, k) + \Delta W(k))$ explicitly. $R(m, k)$, standing for the cost of RGEQR3 was computed as the sum of $FG(m, k)$, $Y(m, k)$ and $T(m, k)$. $\mathit{Diff}(k) = Y(m, k) - F(m, k)$ represents the difference in FLOP count between RGEQRF and DGEQRF during one block step; i.e., the FLOP count of one factor and update step of RGEQRF/DGEQRF. $T(m, k) = M(m, k) + S(m, k)$. $M(m, k)$ is the modified FLOP count needed to compute only those parts of the T matrix that are necessary to factor a single (and last) block column; i.e., the T_i submatrices of T in Figure 4. $S(m, k)$ is the FLOP count saved by not computing the Z_i submatrices of T in Figure 4. We were not able to explicitly compute M and S. However, we produce implicit expressions of M and S and also produce explicit approximations of M and S that satisfies $T = M + S$.

The purpose of producing these FLOP counts is to be able to quantify and predict the performance properties of our new recursive algorithms. For example, we will show that the FLOP count of RGEQR3 is about equal to the sum of the FLOP counts for DGEQR2 and DLARFT for tall thin matrices. Thus for tall thin matrices we can expect about a three fold speed up of our algorithm over LAPACK algorithm DGEQRF (see second paragraph below for an explanation). Our experimental results in Section 3 verify this.

It is counter-intuitive that an algorithm with higher FLOP count will execute faster. The reason is that the FLOP rate depends crucially on the FLOP operands being in the higher levels of memory, e.g., cache. Our recursive algorithms automatically blocks for the memory hierarchy. Additionally, recursion produces a Level 3 implementation of the "factorization part of the code". LAPACK implementations, on the other hand, compute the "factorization part of the code" as Level 2.

For RISC type processors one usually needs a Level 3 type code to maintain peak FLOP rates even if the operands are in cache. To first approximation, execution time is equal to FLOP rate times FLOP count. The main result of this paper shows that recursion improves performance most for tall thin matrices. The reason is the difference between RGEQR3 and the sum of DGEQR2 and DLARFT. For tall thin matrices RGEQRF and DGEQRF spend the majority of their time in RGEQR3 and DGEQR2 plus DLARFT, respectively. We assume that, if appropriate, the switch ISW has been set on. (If we compare RGEQR3 with the switch on to just DGEQR2, the leading term mk^2 of the two FLOP counts are 7/3 and 2. Thus the FLOP ratio of the two codes, for tall thin matrices, is about 7/6.) Now RGEQR3 has a higher FLOP count than DGEQR2 plus DLARFT, but not drastically so. For tall thin matrices they are essentially equal. On the other hand, the FLOP rate ratio of RGEQR3 to DGEQR2 plus DLARFT is, very loosely speaking, about *three* which explains why the performance results are so good for tall thin matrices.

We start with DGEQR2 and determine its FLOP count. We see from Figure 2 that DLARFG is called n times to compute n Householder transforms, $Q(j) = I - \tau uu^T$. This routine computes the 2 (Euclidean) norm of a vector, avoiding overflow. It then scales the vector. The jth vector has size $m - j + 1$. We give the FLOP count as $3(m - j + 2) + 2$. This is an underestimate because we designate the 2 norm cost as $2(m - j)$ and count square root as one FLOP. Summing from 1 to n we get

$$FG(m, n) = n(6m - 3n + 13)/2.$$

Also, for $1 \leq j < n$, DGEQR2 applies $Q(j)^T$ to $A(j : m, j+1 : n)$ from the left by calling DLARF. Let $C = A(j : m, j+1 : n)$. DLARF computes $(I - \tau uu^T)C = C - \tau uw^T$, where $w = C^T u$. Now $w = C^T u$ is computed by calling DGEMV and $C = C - \tau uw^T$ is computed by calling DGER. The computation $w = C^T u$ followed by the scaling $w = \tau w$ costs $2(m+1-j)(n-j)$ FLOPS. The computation $C = C - \tau uw^T$ also costs the same; so, the FLOP count is $4(m+1-j)(n-j)$. Summing j from 1 to $n-1$ we get

$$F(m,n) = 2n(n-1)(m-(n-2)/3).$$

Now we turn to DGEQRF. The routine in Figure 2 is a simplified version in that no mention is made of blocking parameter nx. However, it is sufficient for our purposes. There are two routines we want to analyze, DLARFT and DLARFB. DLARFT computes T via a Level 2 bordering computation. At the jth step, $1 \leq j \leq n$ we have T as an order $j-1$ upper triangular matrix and we want to compute the jth column of T. Letting

$$Y_1 = (Y, v), \quad T_1 = \begin{pmatrix} T & w \\ 0 & z \end{pmatrix},$$

we have

$$(I - YTY^T)(I - \tau vv^T) = I - Y_1 T_1 Y_1^T,$$

and so the cost of computing column j consists of computing $-\tau TY^T v$, where Y is a m by j unit lower trapezoidal matrix and $v = (1,\ u^T)^T$ is an $m-j+1$ vector. DLARFT computes $w = T(1:j-1, j) = -\tau Y^T v$ by calling DGEMV and then calling DTRMV to compute $w = Tw$. The DGEMV computation, including the scaling by τ costs $2(j-1)(m-j+1)$ FLOPS. The DTRMV computation costs $(j-1)^2$ FLOPS. Thus the total costs is $(j-1)(2m-(j-1))$. Summing j from 1 to n we get

$$FT(m,n) = n(n-1)(m-(2n-1)/6).$$

Now we turn to DLARFB. This routine does the bulk of the computation. DLARFB is a general routine. In one specific instance it has the same function as routine DQTC. For our purposes we want to compute $Q^T C$, where C is m by n and $Q = I - YTY^T$. Here Y is m by k lower unit trapezoidal and T is order k upper triangular. Both DLARFB and DQTC have same the FLOP count and use an auxiliary work array of size k by n.

Let

$$Y = \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix},$$

where Y_1 is an order k unit lower triangular matrix and Y_2 is an $m-k$ by k rectangular matrix. Also, let

$$C = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix},$$

where C_1 is k by n and C_2 is $m-k$ by n. Six computations are performed. They are summarized in Table 1.

From Table 1 we have

$$FB(m,n,k) = kn(4m-k-1).$$

This concludes the analysis of DGEQRF.

Table 1: FLOP counts for operations used to perform $C = Q^T C$, where $Q = I - YTY^T$.

Routine	Computation	# floating point operations
DTRMM	$W = Y_1^T * C_1$	$k(k-1)n$
DGEMM	$W = W + Y_2 * C_2$	$2k(m-k)n$
DTRMM	$W = T^T * W$	$k^2 n$
DGEMM	$C_2 = C_2 - Y_2 * W$	$2k(m-k)n$
DTRMM	$W = C_1 * W$	$k(k-1)n$
Matrix Subtract	$C_1 = C_1 - W$	kn
Total Computation	$C = Q^T C$	$kn(4m - k - 1)$

We next turn to RGEQRF. Routines DGEQRF and RGEQRF differ (assuming both use the same blocking strategy) in FLOP count as follows. DGEQRF calls DGEQR2 and DLARFT whereas RGEQRF calls RGEQR3. Both routines use the DQTC computation for the bulk of their operations. Hence to compare the FLOP count difference we need to study the difference between DGEQR2 + DLARFT and RGEQR3. Turning to RGEQR3 we now show that its T computation can be separated out and directly compared to DLARFT. The functional equation governing the T FLOP count is

$$T(m,k) = T(m,k_1) + T(m-k_1,k_2) + k_1 k_2 (2m - k_1), \tag{14}$$

with $T(m,1) = 0$. The $k_1 k_2 (2m - k_1)$ FLOP amount is the cost of computing $-T_1(Y_1^T Y_2)T_2$, where T_1 and T_2 are k_1 and k_2 upper triangular, respectively, Y_1 is m by k_1 unit lower trapezoidal, and Y_2 is $m - k_1$ by k_2 unit lower trapezoidal.

The computation proceed as follows. Let us write

$$Y_1 = \begin{pmatrix} Y_{11} \\ Y_{21} \\ Y_{31} \end{pmatrix}, \quad Y_2 = \begin{pmatrix} Y_{12} \\ Y_{22} \end{pmatrix},$$

where Y_{11} and Y_{12} are order k_1 and k_2 unit lower triangular, respectively, Y_{21} is k_2 by k_1, Y_{31} is $m-k$ by k_1, Y_{22} is $m-k$ by k_2, and $k = k_1 + k_2$. The computation is summarized in Table 2. Note that Y_{11} is not used. First set $W = T(1:k_1, k_1+1:k) = Y_{21}^T$.

We claim $T(m,k) = FT(m,k)$. To show this we substitute $FT(m,k)$ into (14) and verify that (14) holds.

Now we want to remove the DLARFG part of the RGEQR3 computation. This computation is done when $n = 1$. The remaining part of RGEQR3 consists of computing $Q = I - YTY^T$ minus the T computation, i.e., the Y matrix. The FLOP count is

$$Y(m,k) = Y(m,k_1) + Y(m-k_1,k_2) + k_1 k_2 (4m - k_1 - 1). \tag{15}$$

Table 2: FLOP counts for operations used to compute T_3.

Routine	Computation	# floating point operations
DTRMM	$W = W * Y_{12}$	$k_1 k_2^2$
DGEMM	$W = W + Y_{31}^T * Y_{22}$	$2k_1 k_2(m-k)$
DTRMM	$W = -T_1 * W$	$k_1^2 k_2$
DTRMM	$W = W * T_2$	$k_1 k_2^2$
Total Computation	$W = -T_1(Y_1^T Y_2)T_2$	$k_1 k_2(2m - k_1)$

The cost $k_1k_2(4m-k_1-1)$ is the cost of calling DQTC, i.e., the $A(1:m, k_1+1:k)$ $= Q_1^T A(1:m, k_1+1:k)$ computation of Figure 1. Also $Y(m,1) = 0$.

To solve equation (15) we compute an approximate solution. Define $Y = W - \Delta W$, where

$$W(m,k) = W(m,k_1) + W(m-k_1,k_2) + k_1k_2(8m + k_2 - 3k_1 - 2)/2, \tag{16}$$

with $W(m,1) = 0$. $\Delta W(m,k)$ is a function of k only and satisfies,

$$\Delta W(k) = \Delta W(k_1) + \Delta W(k_2) + (k - 2k_1)(k_1k_2)/2, \tag{17}$$

with $\Delta W(1) = 0$.

The solution to (16) is

$$W(m,k) = k(k-1)(4m-k)/2,$$

This can be verified by substituting $W(m,k)$ into (16). Similarly, $Y = W - \Delta W$ can be verified by substitution and using (15), (16) and (17). Now we turn to the solution of (17). Note that $k - 2k_1 = 0$ when k is even and equals 1 when k is odd. Thus, there is no additional count change unless k is odd and then it is $k_1(k_1+1)/2$. In particular, $\Delta W(n) = 0$ when $n = 2^k$. To find the general solution we need some notation. Let $N = \sum_{i\geq 0} b_i 2^i$ be the base two representation of N. Let $N_j = N/2^j = \sum_{i\geq j} b_i 2^{1-j}$ and let $n_j = \sum_{i=0}^{j} b_i 2^i$. The binary tree associated with N has 2^i nodes at level i. At the root there is one node of size N_0. Let $n_{-1} = 0$.

There are n_i nodes of size $N_{i+1} + 1$ and $2^{i+1} - n_i$ nodes of size N_{i+1} at level $i+1$. As we saw above only odd nodes add to the count by the amount $N_{i+1}(N_{i+1}+1)/2$. Now N_i is odd if and only if $b_i = 1$. Thus $N_i + 1$ is odd if and only if $b_i = 0$. Using these facts we see that at level i the increase in the count is

$$((1-b_i)n_i + b_i(2^{i+1} - n_i))(N_{i+1}(N_{i+1}+1)/2)$$

Since $\Delta W(1) = \Delta W(2) = 0$ we get

$$\Delta W(N) = \sum_{i=0}^{\lfloor \log_2 N \rfloor - 1} (n_i + 2b_i(2^i - n_i))(N_{i+1}(N_{i+1}+1)/2).$$

To obtain a bound on $\Delta W(N)$ we note that the contribution at each node at level $i+1$ is four times smaller than a node at level i. Since the number of nodes at level $i+1$ is double that of level i we find the increase at level $i+1$ is at most two times smaller than at level i. Thus an upper bound for $\Delta W(N)$ is two times the value at level 0, i.e., $2(N_1(N_1+1)/2)$. But when N is even, the contribution at level 0 is 0. Hence, we write $N = 2^i n$ where n is odd. Contributions only start showing up at level i and there are 2^i instances of the same positive contribution. Thus an upper bound is

$$\Delta W(N) \le 2^{i+1}(n/2)(n/2+1)/2$$

In our implementation we found $k = 48$ to be an optimal blocking factor for RGEQRF. The corresponding best factor for DGEQRF was $nb = 40$. For $k = 48$, $W(m,k) = 4512m - 54144 - \Delta W(48)$, where $\Delta W(48) = 32$.

Next we compute the FLOP count difference between RGEQRF and DGEQRF during one block iteration of size k, i.e., the call to RGEQR3 minus the calls to DGEQR2 and DLARFT. This difference, *Diff*(m,k), is $Y(m,k)$ minus $F(m,k)$.

$$\mathit{Diff}(m,k) = k(k-1)(k-8)/6 - \Delta W(k), \tag{18}$$

Note that *Diff* is independent of m and the cubic growth in k is only $1/6$, where k is the blocking factor and k is around 50 for practical problems. For $k = 48$, *Diff* is 15008 FLOPS. A realistic problem might have $m = 1000$. Summing $FG(m,k)$, $Y(m,k)$, and $T(m,k)$ we get the FLOP count of RGEQR3:

$$R(m,k) = 3k^2m - k(5k^2+3k-38)/6 - \Delta W(k).$$

For $m = 1000$ and $k = 48$, $R(m,k) \approx 6.8 \cdot 10^6$. The ratio *Diff*$/R(m,k)$ is 0.0022, about 0.2%. For tall thin matrices the difference is negligible, i.e., the FLOP counts are essentially equal. Thus we have quantified a remark we made on page 10.

Now compare *Diff*(k) to $T(m,k) = FT(m,k)$. $T(m,k)$ is at least four times more costly to compute and this occurs when $m = k$. For tall thin matrices (say $m/k = 20$) the ratio is over 100. The point we make here is that the additional cost for T's use in $k-1$ calls to DQTC of RGEQR3 is tiny compared to producing it.

Although the point is very minor, the cost $T(m,k)$ is actually less than $FT(m,k)$. We listed the cost of the first DTRMM computation as $k_1k_2^2$. However, Y_{12} is unit lower triangular; thus, the cost is actually $k_1k_2(k_2-1)$. We used the more expensive cost because the results were equal. Thus using Level 3 BLAS to replace a series of Level 2 computations can be slightly cheaper because a constant feature in the higher Level BLAS can be exploited whereas in the lower Level BLAS it is only one operation and usually can not be taken advantage of.

As mentioned, there is a modification of RGEQR3 which can save FLOPs. Recall that the computation $-T_1(Y_1^TY_2)T_2$ is done after the second recursive call. The computation is only necessary when $Q = I - YTY^T$ is needed to update a submatrix to the right of the currently factored submatrix. If the RGEQR3 task is to factor the entire matrix there is never a right submatrix. In RGEQRF, RGEQR3 is called as a subroutine $\tilde{n} = n/k$ times where k is the block size. On the first $\tilde{n} - 1$ calls there is a right submatrix to be updated. On the last call there is none. So in these cases, in RGEQR3 we omit the T_3 computation when the right boundary of T_3, a submatrix of T, is k. If the right boundary of T_3 is less than k, then T_3 must be computed (see Figure 4). Let $M(m,k)$ be the modified FLOP count and $S(m,k)$ be the saved FLOP count of computing the T_3 matrix. We have

$$\begin{aligned} M(m,k) &= T(m,k_1) + M(m-k_1,k_2) && (19)\\ S(m,k) &= S(m-k_1,k_2) + k_1k_2(2m-k_1), && (20)\end{aligned}$$

where $M(m,1) = S(m,1) = 0$. It is clear by induction that $M(m,k)+S(m,k) = T(m,k)$ as the sum of the right hand side of (19) and (20), using the induction hypothesis, gives the right hand side of (14). We can show that

$$M(m,k) = f(k)m - f_1(k) \qquad (21)$$

$$S(m,k) = g(k)m - g_1(k). \qquad (22)$$

Furthermore,

$$\begin{aligned} f(k) &= f(k_2) + k_1(k_1-1)\\ f_1(k) &= f_1(k_2) + k_1(f(k_2) + (k_1-1)(2k_1-1)/6), && (23)\end{aligned}$$

and

$$\begin{aligned} g(k) &= g(k_2) + 2k_1k_2\\ g_1(k) &= g_1(k_2) + k_1(g(k_2) + k_1k_2), && (24)\end{aligned}$$

with $f(1) = f_1(1) = 0$ and $g(1) = g_1(1) = 0$ define the functions f, f_1, g, and g_1, for $k = 1, 2, \ldots$.

Consider

$$\begin{aligned} af(2x) &= af(x) + x(x-1)\\ af_1(2x) &= af_1(x) + x(f(x) + (x-1)(2x-1)/6) && (25)\\ ag(2x) &= ag(x) + 2x^2\\ ag_1(2x) &= ag_1(x) + x(ag(x) + x^2). && (26)\end{aligned}$$

The prefix a means approximate. A solution to (25), (26) is:

$$af(x) = \frac{1}{3}(x-1)(x-2); \quad af_1(x) = (x-1)(4x^2 - 17x + 18)/42 \qquad (27)$$

$$ag(x) = \frac{2}{3}(x+1)(x-1); \quad ag_1(x) = (x-1)(5x^2 + 5x - 9)/21. \qquad (28)$$

These functions approximate the true functions (23) and (24). For $k = 2^j$, they are identical. Note that

$$f(k) + g(k) = af(k) + ag(k) \equiv k(k-1) \qquad (29)$$

and

$$f_1(k) + g_1(k) = af_1(k) + ag_1(k) \equiv k(k-1)(2k-1)/6. \tag{30}$$

We can also show directly that

$$\begin{aligned}
\Delta g(2k) &= \Delta g(k) && (31)\\
\Delta g(2k+1) &= \Delta g(k+1) + \frac{2}{3}k && (32)\\
\Delta g_1(2k) &= \Delta g_1(k) + k\Delta g(k) && (33)\\
\Delta g_1(2k+1) &= \Delta g_1(k) + k(\Delta g(k+1) + (4k-1)/21), && (34)
\end{aligned}$$

where $\Delta g(k) \equiv g(k) - ag(k)$ and $\Delta g_1(k) \equiv g_1(k) - ag_1(k)$.

Now define

$$\Delta ag(x) \equiv ag(x+1) - ag(x) = \frac{2}{3}(2x+1), \tag{35}$$

and

$$\Delta ag_1(x) \equiv ag_1(x+1) - ag_1(x) = (5x^2 + 5x - 3)/7. \tag{36}$$

It turns out that

$$\begin{aligned}
ag(k) &\leq g(k) < ag(k+1) && (37)\\
ag_1(k) &\leq g_1(k) < ag_1(k+1). && (38)
\end{aligned}$$

We can show that (37) and (38) are true using induction. These proofs are straight forward. A sketch of the proofs is given in Appendix A.

Note that in each region $2^j < k \leq 2^{j+1}$ equations (37) and (38) bound $g(k)$ and $g_1(k)$ from below and above. Thus $\lim_{k\to\infty}(g(k)/k^2) = \lim_{k\to\infty}(ag(k)/k^2) = 2/3$. Similarly, $\lim_{k\to\infty}(g_1(k)/k^3) = \lim_{k\to\infty}(ag_1(k)/k^3) = 5/21$. From (29) and (30) we get $\lim_{k\to\infty}(f(k)/k^2) = 1/3$ and $\lim_{k\to\infty}(f_1(k)/k^2) = 2/21$. Let $r_m(k) = f(k)/g(k)$ and $r(k) = f_1(k)/g_1(k)$. Figure 5 gives log plots of r_m and r vs. k. The plots verify experimentally that r_m and r have the correct limits of 0.5 and 0.4.

We now use the above results to compare RGEQR3 with the switch on to DGEQR2. The cost of RGEQR3 in this case becomes $R(m,k) - S(m,k)$. Using $ag(k)$ and $ag_1(k)$ for $g(k)$ and $g_1(k)$ and keeping only terms in k^2m and k^3, $R(m,k) - S(m,k) \approx k^2(7/3m - 40/21k)$. The FLOP count of DGEQR2, keeping the same terms, is $k^2(2m - k/3)$. For large m and small k (tall thin matrix case) the FLOP ratio of RGEQR3 to DGEQR2 is about 7/6; i.e., the FLOP count is about 17% higher. Assuming a 3 to 1 FLOP rate ratio, it follows that RGEQR3 should execute $2\frac{4}{7}$ times faster. This loose analysis quantifies the remarks we made on page 10.

We close this section with some remarks on why the hybrid algorithm RGE-QRF should not be replaced with RGEQR3. Let $m = n$ and set $k = n$ in $R(m,k)$. The cubic term is $(13/6)n^3$. For DGEQRF and RGEQRF using fixed $nb \ll n$ the extra FLOP cost is marginally higher than DGEQR2 which has a cubic term of $(4/3)n^3$.

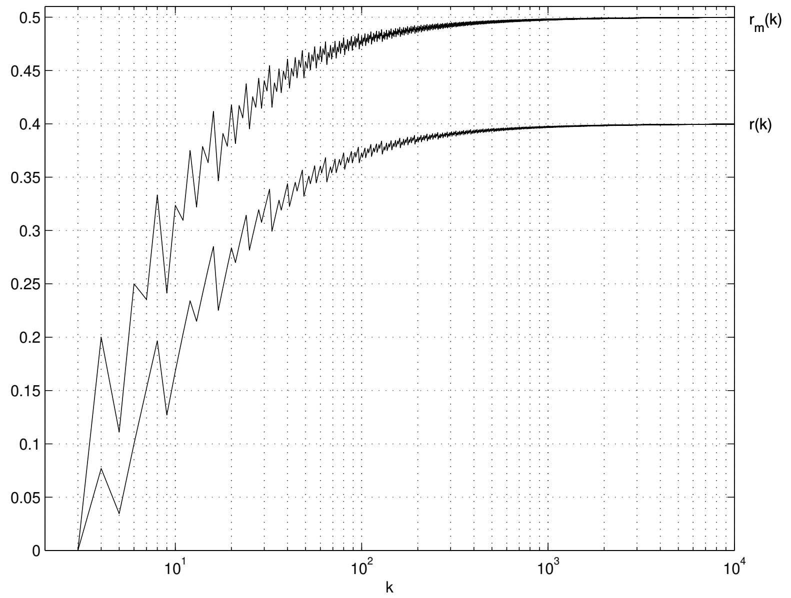

Figure 5: Log plots of $r_m(k)$ and $r(k)$ vs. k.

Even if we replace $T(m,k)$ with $M(m,k)$ to compute $R(m,k)$ (which one should do) the reduction in the cubic term would only be $3/7$ and the new cubic term would be $73/42$. In summary, the preceding FLOP count analysis of this section shows that the hybrid algorithm RGEQRF should be chosen. RGEQRF should exhibit Level 3 performance even for relatively small values of k. For large m and n, and using $k = nb \ll n$, the additional FLOP counts is marginally higher than for DGEQRF.

2.3 The routine DQTC

The matrix multiplications in the update operation

$$\hat{C} \longleftarrow Q^T C = C - YT^T Y^T C,$$

can be performed in several different orders. Depending on the sizes of the matrices involved, the total number of floating point operations may be quite different for the different approaches. In Table 3 we show the five possible alternatives and the number of floating point operations required for each of them, assuming C is $n_1 \times n_2$, Y is $n_1 \times k$ unit lower trapezoidal, and T is $k \times k$ upper triangular. Since we here consider each individual update through the whole factorization of an m by n matrix, we have for clarity chosen to use the new variables n_1 and n_2 to denote the size of the matrix to be updated.

For comparisons with results in Section 2.2, where we express the operations in terms of the first update, n_1 and n_2 should be replaced by m and $n-k$,

Table 3: Alternative approaches for updates, $\hat{C} \longleftarrow Q^T C = C - YT^T Y^T C$, in QR factorization.

	Method	# floating point operations
C_1:	$C - Y * (T^T * (Y^T * C))$	$kn_2(4n_1 - k - 1)$
C_2:	$C - Y * ((T^T * Y^T) * C)$	$kn_2(4n_1 - k - 1) + k(n_1 k - k + 1/3 - k^2/3)$
C_3:	$C - (Y * (T^T * Y^T)) * C$	$n_1^2(2n_2 + 2k - 1) + k(1/3 - k - k^2/3)$
C_4:	$C - ((Y * T^T) * Y^T) * C$	$n_1^2(2n_2 + 2k - 1) + k(n_1 - kn_1 - 2k + 1)$
C_5:	$C - (Y * T^T) * (Y^T * C)$	$kn_2(4n_1 - 2k) + k(n_1 k - k - 2k^2/3 + 2/3)$

respectively. In the update performed in a QR factorization of a large $m \times n$ matrix where $m \geq n$, we know that $n_1 \geq n_2 + k$ and that k is normally much smaller than n_1 and n_2 for almost all updates performed.

It is apparent that methods C_1 and C_5 require significantly less floating point operations than the other alternatives for the typical case in the QR factorization, i.e., for n_1 large and k relatively small. The method, C_1 requires the smallest number of floating point operations of these methods for all updates that occur in RGEQRF for a factorization of an $m \times n$ matrix if $m \geq n$. For the case $m = n$, the method C_5 is almost as cheap if $m = n$ is large and k is small.

The larger the block size is or the smaller n_2 is compared to n_1, the larger is the gain from using method C_1 compared to C_5. From this we conclude that method C_1 definitely is to be preferred by the recursive algorithm RGEQR3 where n_2 typically never exceed half of the block size *nb* used in the calling routine RGEQRF. For updates in RGEQRF, the choice of method is less obvious. However, in order to reduce the software complexity we have chosen to use the same method for all updates. Therefore, the routine DQTC implements method C_1. The routine DLARFB called by the LAPACK algorithm DGEQRF also implements the C_1 method, even though the implementation is slightly different. In both routines, the operations are performed as a series of calls to the Level 3 BLAS routines DGEMM and DTRMM.

3 Uniprocessor Performance Analysis

The uniprocessor performance of RGEQRF has been evaluated on a 120 MHz IBM POWER2 node and on one processor of a 4-way 332 MHz IBM PPC 604e SMP-node. The performance results are compared to those of the LAPACK Level 3 QR factorization routine DGEQRF. The performance has been analyzed both for square matrices ($m = n$) and for tall thin matrices ($m \gg n$), that are important in many applications. Results are presented for large as well as for small matrices.

Both RGEQRF and DGEQRF have parameters that need to be set properly in order to obtain optimal performance. The only parameter that needs to be set in RGEQRF is nb; i.e., nb is the number of columns to be factorized by the pure recursive algorithm RQEQR3. In DGEQRF, nb is the number of columns to be factorized by the Level 2 algorithm DGEQR2. DGEQRF also uses a parameter nx as a cross-over point, so that when the remaining matrix to be factorized consist of less than nx columns, it is completely factorized by the Level 2 algorithm. By using an appropriately chosen cross-over point, the overhead (extra FLOPS) of the Level 3 algorithm is avoided for matrices too small to benefit from the Level 3 operations.

One could argue that a similar cross-over point should be used also in RGEQRF in order to improve performance, but in our effort to make the routine as close to self-tuned as possible, we have chosen not to use this approach. We note, however, that the ISW switch being turned on has the effect of automatically introducing a second type of blocking. In the following sections, all serial and parallel performance results for the recursive algorithm have been obtained with the ISW switch turned on.

Of course, the optimal settings of nb for RGEQRF and nb and nx for DGEQRF vary depending on the size and the shape of the matrix to be factorized. Since we believe that most users of linear algebra library software do not change these parameters between different calls to the QR factorization routines, we have chosen to find a good general setting for these parameters for each machine used for performance testing. The settings have been made to give the best overall performance for square matrices varying from $m = n = 100$ to $m = n = 2000$.

In DGEQRF, the parameters nb and nx are obtained from the LAPACK routine ILAENV, which always should be modified to return the "optimal" parameters at the time of the installation (as we have done for the two machines considered here). We fear that some users do not notice this, or that the users do not take the time to determine what values to choose for these parameters. One should also note that the values for the parameters nb and nx used in DGEQRF are the same as the ones used in routines for RQ, QL and LQ factorization. Of course, there are default values to be used ($nb = 32$ and $nx = 128$ in the version we used), but use of them will lead to suboptimal performance on many computer systems. We also note that significantly more effort is needed to find optimum parameter settings for DGEQRF than for RGEQRF since the two parameters (nb and nx) used in DGEQRF defines a 2-dimensional parameter space, as the parameter space for RGEQRF is 1-dimensional (nb only). Based on this, we believe that each step towards an automatic tuning is just as important as the optimum performance achieved.

A defect in the recursive approach is the overhead cost of handling the recursion (e.g., the calling overhead). Thus for small m and n the direct method will perform faster. A way to avoid this defect is to introduce a recursive blocking parameter rb. Pure recursion has $rb = 1$. When n, the recursion variable, satisfies $n > rb$ a recursion step is taken, otherwise a direct method is used. For QR the value of rb can be quite small, say $rb = 4$. This is so because we are interested in achieving Level 3 performance via register blocking; typically an unrolling of 4 works well. However, we only used pure recursion, $rb = 1$, in our performance studies.

3.1 IBM POWER2 performance results

The performance results are obtained on a 120 MHz IBM POWER2 node, using IBM XL Fortran 5.1 [13]. The overall best block sizes nb for square matrices were found to be 48 for RGEQRF and 40 for DGEQRF. The overall best cross-over point nx for using Level 2 algorithm only in DGEQRF was found to be 112. These parameters were used in all performance tests for this node reported in this section, including tests for tall thin matrices. The BLAS routines used are from ESSL version 2.2 [14].

3.1.1 IBM POWER2 — square matrices

The performance results in *Mflops/s* for RGEQRF and DGEQRF on square matrices are shown in Figure 6. RGEQRF clearly reaches high performance much earlier than DGEQRF for increasing $m = n$. For very small matrices RGEQRF does not give optimal performance mainly because of overhead from handling the recursion and because we do not use any cross-over point similar to nx in DGEQRF to avoid the extra FLOPS performed in the T computations for matrices too small to benefit from Level 3 operations.

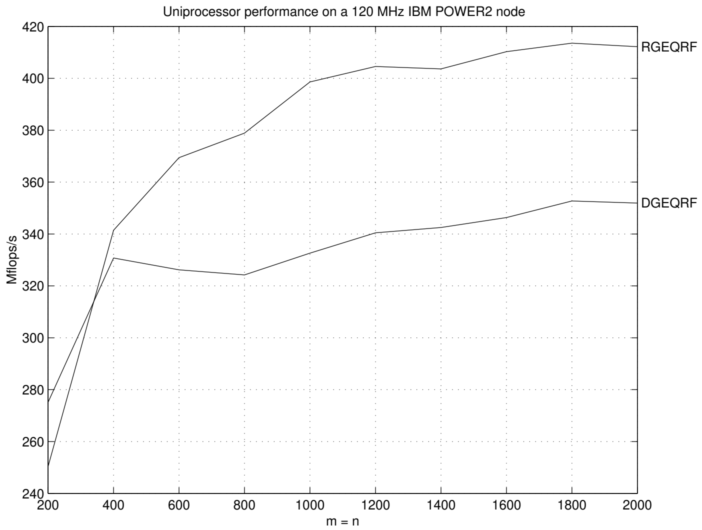

Figure 6: Uniprocessor performance results in *Mflops/s* for the hybrid recursive algorithm RGEQRF and DGEQRF of LAPACK on a 120 MHz IBM POWER2 node.

We note that the highest performance obtained for RGEQRF is 415.4 *Mflops/s* which is 90.3% of the theoretical peak performance of the POWER2 node (460

Mflops/s). DGEQRF reaches 353.4 *Mflops/s*, which is 76.8% of the theoretical peak performance.

In Table 4 is shown that the performance of RGEQRF is nearly 20% better than that of DGEQRF for square matrices with $n \geq 800$. This difference in performance seems to stabilize for large matrices. A similar behavior was also observed in the first results for the hybrid recursive algorithm presented for a 112 MHz IBM PPC 604 node in [10].

$m = n$	200	400	600	800	1000	1200	1400	1600	1800	2000
Ratio	0.91	1.03	1.13	1.17	1.20	1.19	1.18	1.18	1.17	1.17

Table 4: Ratio of uniprocessor performance results for RGEQRF and DGEQRF on a 120 MHz IBM POWER2 node.

3.1.2 IBM POWER2 — tall thin matrices

The performance results for RGEQRF and DGEQRF on tall thin matrices are shown in Table 5 and Table 6, respectively.

Table 5: Uniprocessor performance on tall thin matrices for RGEQRF on 120 MHz IBM POWER2 processor.

$m \backslash n$	50	100	150	200	250	300	350	400	450	500
200	152.3	214.6	231.4	249.7	-	-	-	-	-	-
400	198.6	270.6	288.6	306.7	324.8	333.4	336.9	341.3	-	-
600	218.4	294.4	302.2	322.3	333.0	340.3	356.0	355.6	357.9	362.8
800	218.8	295.3	301.1	319.7	337.8	346.2	362.9	368.5	364.1	369.1
1000	232.1	296.6	305.0	326.8	344.1	354.2	369.0	374.2	369.2	374.7
1200	235.8	297.0	310.4	328.8	346.3	356.1	370.3	375.9	372.4	377.4
1400	220.3	298.6	307.7	327.6	341.9	348.0	366.2	372.3	368.7	373.3
1600	220.7	297.5	307.6	328.2	339.1	352.7	369.6	375.2	370.7	376.3
1800	225.7	301.0	308.2	327.6	345.0	355.3	372.7	378.9	372.6	377.8
2000	225.0	295.0	301.8	322.2	341.6	351.7	362.8	371.0	368.9	371.4

Here, RGEQRF shows better performance than DGEQRF for all but four cases, and the difference is sometimes much larger than for the tests with square matrices. RGEQRF reaches a performance near or over 300 *Mflops/s* already for $n = 100$ for $m \geq 600$. For DGEQRF to obtain similar performance for large m, the number of columns n must be significantly larger (around 400).

In Figure 7 the performance ratio of RGEQRF to DGEQRF is shown for different values of n.

Notably, for $m \geq 800$ and $n \leq 150$, RGEQRF outperforms DGEQRF by a factor 1.5 – 2.3 and for $n = 100$, RGEQRF shows more than twice the performance of DGEQRF for $m \geq 800$. It confirms that the benefits of the recursive

Table 6: Uniprocessor performance on tall thin matrices for DGEQRF on 120 MHz IBM POWER2 processor.

$m\backslash n$	50	100	150	200	250	300	350	400	450	500
200	217.4	210.3	242.9	278.0	-	-	-	-	-	-
400	215.8	155.6	206.4	291.0	303.0	314.1	313.6	328.9	-	-
600	177.2	151.5	197.5	271.1	286.5	301.4	310.0	329.9	330.7	331.2
800	145.4	133.7	175.9	246.0	261.4	260.3	286.4	308.9	311.1	305.4
1000	147.5	146.1	186.9	248.3	264.8	279.8	289.2	309.0	311.7	314.9
1200	146.5	144.0	184.9	246.9	263.7	278.3	288.0	308.3	311.4	315.0
1400	146.6	145.0	186.6	247.0	263.8	276.6	286.2	305.9	305.0	313.2
1600	131.7	130.5	170.6	232.8	250.2	264.3	274.3	296.2	299.5	305.5
1800	142.0	144.3	185.2	244.1	260.5	275.0	284.9	304.2	307.2	312.1
2000	139.8	140.7	181.8	241.2	257.9	272.2	281.2	301.4	301.6	309.3

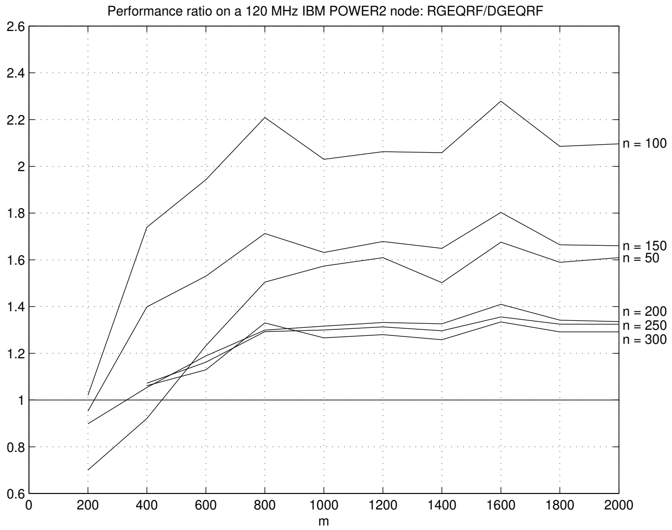

Figure 7: Performance ratio of RGEQRF to DGEQRF on a 120 MHz IBM POWER2 node for $m = 100, \ldots, 2000$ and $n = 50, \ldots, 300$.

algorithm is bigger for tall thin matrices than for square matrices. However, one should note that the cross-over point *nx* used in DGEQRF could be changed to give better performance for the LAPACK routine also for tall thin matrices. As mentioned above, *nx* is here chosen to give a good overall performance. In RGE-QRF the recursion leads to an automatic blocking that apparently performs well for a wider range of matrix sizes.

3.2 IBM PPC 604e uniprocessor performance results

The performance results are obtained on one processor of a 4-way 332 MHz IBM PPC 604e SMP-node, using IBM XL Fortran 5.1 [13]. The overall best block sizes *nb* for square matrices were found to be 56 for RGEQRF and 40 for DGEQRF. The overall best cross-over point *nx* for using Level 2 algorithm only in DGEQRF was found to be 72. We remark that this optimal cross-over point is very small, in fact less than twice the block size *nb*. These parameters were used in all performance tests for this node reported in this section, including tests for tall thin matrices. The BLAS routines used are from ESSL version 3.1 [12].

3.2.1 IBM PPC 604e — square matrices

The performance in *Mflops/s* for RGEQRF and DGEQRF on square matrices are shown in Figure 8. As for the POWER2 results, RGEQRF outperforms DGEQRF for all matrices except for the very small ones.

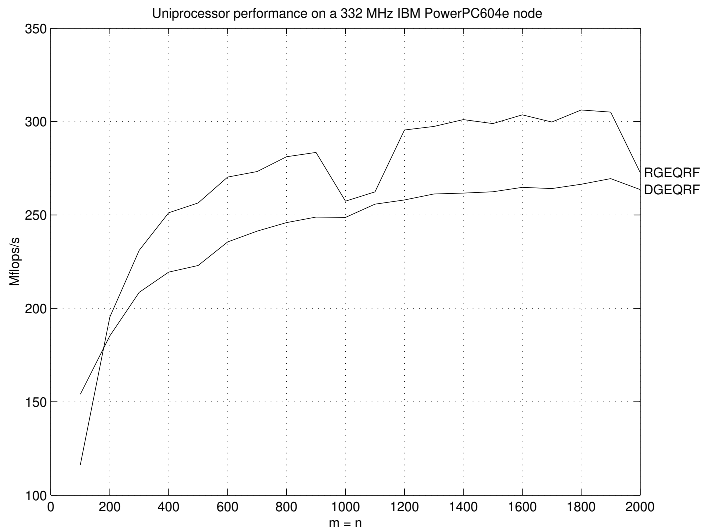

Figure 8: Uniprocessor performance results in *Mflops/s* for the hybrid recursive algorithm RGEQRF and DGEQRF of LAPACK on a 332 MHz IBM PPC 604e node.

$m=n$	200	400	600	800	1000	1200	1400	1600	1800	2000
Ratio	1.05	1.14	1.15	1.14	1.04	1.15	1.15	1.15	1.15	1.03

Table 7: Ratio of uniprocessor performance results for RGEQRF and DGEQRF on a 332 MHz IBM PPC 604e node.

The performance ratio of RGEQRF to that of DGEQRF is shown in Table 7. Except for drops in performance for $m = n = 1000$, 1100, and 2000, RGEQRF shows around 15% better performance than DGEQRF for large matrices, which is almost as good as what was observed for the POWER2 node.

However, there seems to be some irregular drops in performance for $m = n = 1000$, 1100, and 2000 in Figure 8. In order to get a better picture of the critical matrix sizes, the performance results for $m = n = 900, 901, 902, \ldots, 1200$ are presented in Figure 9. From this figure, it is clear that the performance is around 10% lower than what one would expect for $925 < m = n < 1130$.

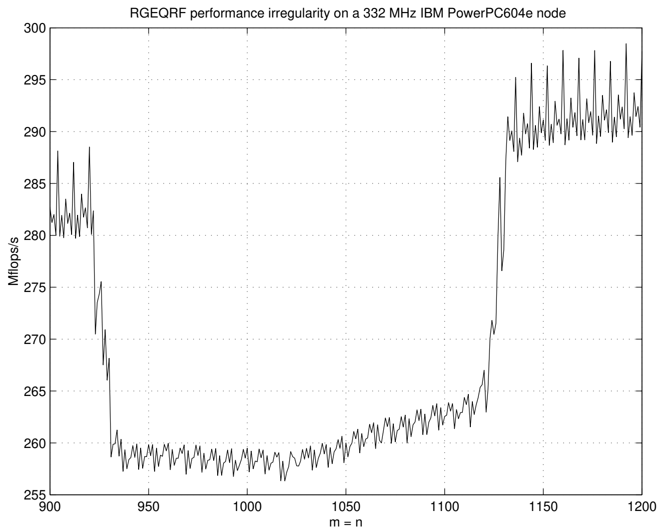

Figure 9: Closeup of performance results for RGEQRF on a 332 MHz IBM PPC 604e node for $m = n = 900$ to 1200.

Additional tests indicate that this performance problem depends on the leading dimension of the matrix. All performance results presented here are performed with a leading dimension equal to $m + 1$. If the leading dimension, for example, is increasing by a constant 300, a similar performance drop is observed for $m = n$ around 700 to 800.

This behavior is currently not fully understood, but one possible reason is that this is an effect explained by the fact that the Level 2 cache memory on the 332 MHz PPC 604e node is direct mapped, i.e., it is not a set associative cache memory. We remark that no indications of similar problems were observed on the POWER2 node, which indeed has a 4 way set associative Level 2 cache memory.

3.3 IBM PPC 604e — tall thin matrices

The performance results for RGEQRF and DGEQRF on tall thin matrices are shown in Table 8 and Table 9, respectively.

Table 8: Uniprocessor performance on tall thin matrices for RGEQRF on 332 MHz IBM PPC 604e processor.

$m\backslash n$	50	100	150	200	250	300	350	400	450	500
200	125.2	169.3	181.4	190.4	-	-	-	-	-	-
400	143.4	188.4	206.9	218.9	234.0	239.3	249.6	251.3	-	-
600	138.2	180.6	202.7	218.1	239.7	246.5	253.5	255.8	261.9	265.5
800	126.5	176.9	199.7	216.8	235.8	243.7	249.4	252.3	262.4	267.0
1000	114.2	152.5	168.1	176.8	195.0	199.2	211.5	213.4	232.8	237.4
1200	115.7	164.3	193.5	210.5	233.6	242.6	247.9	251.4	260.3	266.5
1400	119.9	172.6	194.6	211.9	233.6	243.2	250.6	254.0	263.3	269.4
1600	121.8	172.8	194.7	212.5	233.3	242.8	249.4	252.6	261.9	268.3
1800	116.0	168.5	190.9	208.9	229.2	239.4	245.9	250.2	258.6	265.5
2000	104.7	148.1	164.4	174.1	191.0	196.0	209.7	212.7	230.3	235.9

Table 9: Uniprocessor performance on tall thin matrices for DGEQRF on 332 MHz IBM PPC 604e processor.

$m\backslash n$	50	100	150	200	250	300	350	400	450	500
200	87.3	123.2	151.3	175.2	-	-	-	-	-	-
400	80.0	111.7	132.8	169.6	185.3	198.2	208.0	217.6	-	-
600	72.0	96.5	128.4	163.7	179.5	191.4	197.3	209.0	216.2	222.6
800	63.0	88.2	108.5	152.6	168.3	180.6	188.4	202.5	210.3	216.8
1000	50.2	76.9	96.4	135.9	151.7	162.5	169.5	187.4	196.4	203.8
1200	46.8	71.2	91.4	130.5	144.8	156.6	165.4	184.5	193.1	200.1
1400	44.1	66.1	86.1	120.9	135.8	147.4	157.4	176.0	184.5	191.7
1600	42.4	63.6	82.7	116.5	130.3	141.9	151.4	169.6	177.8	184.9
1800	41.1	61.6	80.1	111.6	125.4	136.9	146.6	163.8	172.1	179.1
2000	39.7	60.2	78.2	108.0	121.6	133.0	141.8	158.3	166.4	173.6

As for the POWER2 results, RGEQRF seems to handle tall thin matrices significantly better than DGEQRF. In fact, RGEQRF outperforms DGEQRF on all matrix sizes tested, and the performance ratios approach nearly a factor of three. As seen in Figure 10, the gain from using RGEQRF is bigger the smaller the n is.

We also remark that for larger n, a performance degradation around $m = 1000$ similar to that of square matrices can be observed.

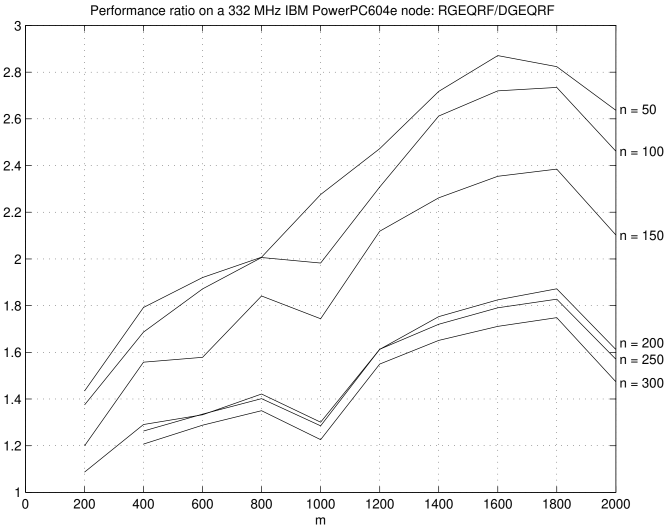

Figure 10: Performance ratio of RGEQRF to DGEQRF on a 332 MHz IBM PPC 604e node for $m = 100, \ldots, 2000$ and $n = 50, \ldots, 300$.

3.3.1 General comments to the uniprocessor performance

For both square and tall thin 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.

For RGEQRF we observed that the performance on the POWER2 node for tall thin matrices is close to constant for increasing m and fixed n. On the PPC 604e the performance decreases for increasing m and fixed n.

Since the number of floating point operations is $\mathcal{O}(mn^2)$ (for $m \geq n$) and the amount of data is $\mathcal{O}(mn)$, the number of floating point operations per matrix element is constant for fixed n. That the performance decreases as the number of floating point operations per matrix element remains the same indicates that the memory system is the bottleneck. Hence, it seems like the POWER2 memory system is much better suited to serve the POWER2 processor with data than is the corresponding system on the PPC 604e.

We also make similar observations from the results of the LAPACK algorithm DGEQRF, even though this routine shows a slight decrease in performance for increasing m on the POWER2 system. This indicates that the data locality in DGEQRF is not quite as good as in the recursive hybrid algorithm RGEQRF.

Even though we have seen in Section 3 that RGEQRF shows significantly better performance than DGEQRF, we believe that a possible bigger advantage of the recursive approach is the fact that the automatic variable blocking leads to an automatic tuning which ensures good performance. Also, very little time and effort was spent on adjusting tuning parameters, compared to the effort we gave to tuning DGEQRF.

4 The Parallel Algorithm PRGEQRF

In the parallel version of RGEQRF, several execution threads are created by a `parallel do` construction. The number of threads created is equal to the number of available processors, and a routine THRQR (standing for THRead QR) is called once for each parallel thread. In the THRQR routine, each processor repeatedly performs three major 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 that block through a call to RGEQR3, if required. The processor returns from THRQR when the QR factorization is completed. A brief description of routine THRQR is given in Figure 11.

THRQR($m, n, A, lda, \tau, work$)
if ($me = 0$) **then**
    call RGEQR3 $A(1:m, 1:firstjb) = (Y, R, nextT)$
endif
do while (There is still work enough for me)
    call GETJOB(j, *first*, *last*, jb, T, $nextT$, Y, $nextY$, R, *dofact*, $\ldots$)
    compute $(I - YT^TY^T)A(j:m, first:last)$ using *work* and T via
    a call to DQTC
    if (*dofact*) **then**
        call RGEQR3 $A(first:m, first+jb-1) = (nextY, R, nextT)$
    endif
enddo

Figure 11: A brief description of routine THRQR.

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 [6] for dynamically assigning work to the processors. The GETJOB algorithm is sketched in Figure 12, 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

```
do 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
enddo
```

Figure 12: Algorithm GETJOB does the pool-of-tasks implementation.

adaptive blocking is used [3]. 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, i.e., $last = first + jb - 1$ in Figure 11. 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. For example, one processor may still be performing updates with respect to the factorization in iteration $i - 1$ while another processor is updating with respect to iteration i and factorizing for iteration $i + 1$. In this example two T matrices are used read only and a third one is being produced. In practice, at least three such T matrices are required to avoid contention, and this is what is used in this implementation.

We remark that this pool-of-tasks algorithm implements a general strategy for dynamic load balancing that may be used for parallel implementations of other matrix factorizations algorithms as well. It is related to the strategies used for the LU factorization algorithm described in [1] and the dynamic load balancing versions of LU and *Cholesky* factorization in [8].

5 Parallel Performance Results

The performance results for the 4-way 332 MHz IBM PPC 604e SMP node were obtained using the same compiler and ESSL library as in Section 3.2. In order to perform tests on 1, 2, 3, and 4 processors, the environment variable `XLSMPOPTS parthds` is set to 1, 2, 3, and 4, respectively.

The performance in *Mflops/s* and the parallel speedup, S, for PRGEQRF for 1–4 processors on square matrices are shown in Figure 10.

Table 10: Parallel performance for PRGEQRF on a 4-way 332 MHz IBM PPC 604e SMP node.

	#proc = 1	#proc = 2		#proc = 3		#proc = 4	
$m = n$	*Mflops/s*	*Mflops/s*	S	*Mflops/s*	S	*Mflops/s*	S
100	114.5	146.6	*1.28*	137.5	*1.20*	133.5	*1.17*
200	192.7	318.6	*1.65*	393.7	*2.04*	398.0	*2.07*
300	225.1	399.3	*1.77*	541.0	*2.40*	612.2	*2.72*
400	243.0	452.9	*1.86*	615.9	*2.53*	759.2	*3.12*
500	251.4	477.0	*1.90*	653.4	*2.60*	762.7	*3.03*
600	265.6	517.5	*1.95*	713.7	*2.69*	886.1	*3.34*
700	268.7	522.7	*1.95*	741.1	*2.76*	895.1	*3.33*
800	277.0	541.4	*1.95*	766.7	*2.77*	942.6	*3.40*
900	273.7	533.3	*1.95*	772.9	*2.82*	955.0	*3.49*
1000	250.4	493.9	*1.97*	735.3	*2.94*	913.5	*3.65*
1100	257.3	505.9	*1.97*	768.4	*2.99*	964.0	*3.75*
1200	289.0	556.7	*1.93*	823.8	*2.85*	1039.8	*3.60*
1300	295.3	568.1	*1.92*	851.8	*2.88*	1061.2	*3.59*
1400	300.7	577.8	*1.92*	860.5	*2.86*	1090.8	*3.63*
1500	299.0	572.4	*1.91*	863.1	*2.89*	1087.2	*3.64*
1600	303.9	584.5	*1.92*	873.0	*2.87*	1116.1	*3.67*
1700	299.2	573.4	*1.92*	860.4	*2.88*	1090.6	*3.65*
1800	306.1	592.3	*1.94*	879.3	*2.87*	1135.5	*3.71*
1900	301.3	579.5	*1.92*	868.0	*2.88*	1119.8	*3.72*
2000	270.4	524.5	*1.94*	805.3	*2.98*	1072.9	*3.97*

It should be noted that the uniprocessor performance for the parallel algorithm is almost the same as the results presented for the serial algorithm in Figure 8. For some matrix sizes, the parallel algorithm shows negligibly higher *Mflops/s* figures; for others, it shows up to 3% lower figures.

In order to facilitate the interpretation of the parallel speedup, Figure 13 shows the speedup for 1, 2, and 3 processors for varying matrix sizes. The largest speedups observed are 1.97, 2.99, and 3.97, for 2, 3 and 4 processors, respectively.

It should be noted that the ESSL library used here (version 3.1) [12] in fact is parallelized and it is not clear to the authors how the parallel BLAS routines are designed to work when multiple threads simultaneously are calling BLAS routines. When each parallel thread in PRGEQRF performs a call to a BLAS routine, that routine should ideally not create additional threads itself. However, it is not clear if this is actually what is done or if four parallel threads each

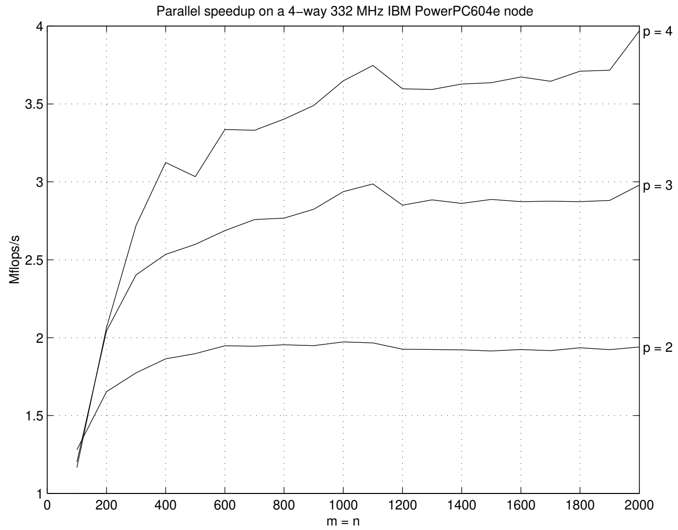

Figure 13: Relative parallel speedup of PRGEQRF for 2, 3, and 4 processors on a 4-way 332 MHz IBM PPC 604e node for $m = n = 100, \ldots, 2000$.

calling a BLAS routine eventually may lead to 16 parallel threads in total executing inside the BLAS. Such construction would of course introduce additional software overhead, taking into account also that each BLAS routine is called several times for each processor. However, from the parallel speedup figures, we can conclude that if additional threads are created, the extra cost for that is close to negligible for large matrices. How much these extra costs will effect the performance for very large number of processors is still to be investigated.

6 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. Another important advantage is the automatic tuning obtained from the automatic variable blocking following from the recursion.

Recursion has previously been successfully applied to LU factorization [11, 16]. For the LU case, no extra FLOPS are introduced from recursion and the performance exceeds or significantly exceeds that of LAPACK block algorithms. In this contribution we have shown 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

[8] and the Alliant fx2816 in [7] and it is similar to what is presented for IBM 3090 VF/600J in [7].

7 Acknowledgements

We are grateful to Bo Kågström and Carl Tengwall who were instrumental in setting up our fruitful collaboration. We thank Isak Jonsson for providing coefficient of m for S when $k = 2^j$ and for the plots in Figure 5. We are also grateful to the anonymous referees for carefully reading the manuscript, and to the second referee for pointing out an alternative approach for deriving $Y(m,k)$ in (15) for the simplified case when $n = 2^k$.

This research was conducted using the resources of High Performance Computing Center North (HPC2N) in Umeå, Sweden, and UNI•C in Lyngby, Denmark.

References

[1] R.C. Agarwal and F.G. Gustavson. A parallel implementation of matrix multiplication and LU factorization on the IBM 3090. In M. Wright, editor, *Aspects of Computation on Asynchronous and Parallel Processors, IFIP*, pages 217–221, Amsterdam, 1989. North-Holland.

[2] 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.

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

[4] 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.

[5] R. K. Brayton, F.G. Gustavson, and R.A. Willoughby. Some results on sparse matrices. *Math. of Computation*, 24:937–954, 1970.

[6] A. Chalmers and J. Tidmus. *Practical Parallel Processing.* International Thomson Computer Press, UK, 1996.

[7] 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., editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 330–338, Philadelphia, 1993. SIAM Publications.

[8] 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.

[9] 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.

[10] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In B. Kågström et al., editors, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, No. 1541, pages 120–128, 1998.

[11] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[12] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference.* Version 3, Release 1.

[13] IBM. *XL Fortran for AIX, Language Reference.* Version 5, Release 1.

[14] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference.* Version 2, Release 2, 1994.

[15] 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.

[16] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix. Anal. Appl.*, 18(4), 1997.

A Sketch of proof for equations (37) and (38)

Equations (37) and (38) are the same as

$$\begin{aligned} 0 &\leq \Delta g(k) < \Delta ag(k) \quad and & (39) \\ 0 &\leq \Delta g_1(k) < \Delta ag_1(k) & (40) \end{aligned}$$

The following relations are true:

$$\begin{aligned} ag(x) &= \frac{2}{3}(x^2-1), \quad and & (41) \\ \Delta ag(x) &\equiv ag(x+1) - ag(x) = \frac{2}{3}(2x+1), & (42) \end{aligned}$$

$$\begin{aligned} ag_1(x) &= \frac{1}{21}(5x^3 - 14x + 9), \quad and & (43) \\ \Delta ag_1(x) &\equiv ag_1(x+1) - ag_1(x) = \frac{1}{7}(5x^2 + 5x - 3), & (44) \end{aligned}$$

$$\begin{aligned} \Delta ag(2k) &= \Delta ag(k) + \frac{4}{3}k, \quad and & (45) \\ \Delta ag_1(2k) &= \Delta ag_1(k) + \frac{5}{7}k(3k+1), & (46) \end{aligned}$$

$$\begin{aligned} \Delta ag(2k+1) &= \Delta ag(k+1) + \frac{4}{3}k, \quad and & (47) \\ \Delta ag_1(2k+1) &= \Delta ag_1(k+1) + \frac{15}{7}k(k+1), & (48) \end{aligned}$$

$$\begin{aligned} g(2k) &= g(k) + 2k^2, \quad and & (49) \\ g_1(2k) &= g_1(k) + k(g(k) + k^2), & (50) \end{aligned}$$

$$\begin{aligned} g(2k+1) &= g(k+1) + 2k(k+1), \quad and & (51) \\ g_1(2k+1) &= g_1(k+1) + k(g(k+1) + k(k+1)), & (52) \end{aligned}$$

$$\begin{aligned} \Delta g(2k) &= \Delta g(k), \quad and & (53) \\ \Delta g_1(2k) &= \Delta g_1(k) + k\Delta g(k), & (54) \end{aligned}$$

$$\begin{aligned} \Delta g(2k+1) &= \Delta g(k+1) + \frac{2}{3}k, \quad and & (55) \\ \Delta g_1(2k+1) &= \Delta g_1(k+1) + k(\Delta g(k+1) + \frac{(4k-1)}{21}), & (56) \end{aligned}$$

where $\Delta g(k) \equiv g(k) - ag(k)$ and $\Delta g_1(k) \equiv g_1(k) - ag_1(k)$.

There are four induction proofs, two for (39) and two for (40). Using (45), (47), (53), and (55) we get

$$\begin{aligned} \Delta ag(2k) - \Delta g(2k) &= \Delta ag(k) - \Delta g(k) + \frac{4}{3}k, \quad and & (57) \\ \Delta ag(2k+1) - \Delta g(2k+1) &= \Delta ag(k+1) - \Delta g(k+1) + \frac{2}{3}k. & (58) \end{aligned}$$

Using (46), (48), (54), and (56) along with (45), (47), (53), and (55) we get

$$\begin{aligned}
\Delta ag_1(2k) - \Delta g_1(2k) &= (\Delta ag_1(k) - \Delta g_1(k)) \\
&\quad + k(\Delta ag(k) - \Delta g(k)) + \frac{k}{21}(17k+1), \ \textit{and} \quad (59) \\
\Delta ag_1(2k+1) - \Delta g_1(2k+1) &= (\Delta ag_1(k+1) - \Delta g_1(k+1)) \\
&\quad + k(\Delta ag(k+1) - \Delta g(k+1)) + \frac{k}{21}(13k+4). \quad (60)
\end{aligned}$$

Lemma 1. $\Delta g(k) \geq 0$.
Proof: $\Delta g(1) = 0$. Assume result is true for $0 < k \leq 2^j$. Let $2^j < l \leq 2^{j+1}$. For $l = 2k$ the result follows from (53) and the induction hypothesis (I.H.). For $l = 2k+1$, the results follows from (55) and the I.H. □

Lemma 2. $\Delta ag(k) > \Delta g(k)$.
Proof: $\Delta ag(1) = 2$ and $\Delta g(1) = 0$. Assume result is true for $0 < k \leq 2^j$. Let $2^j < l \leq 2^{j+1}$. For $l = 2k$ the result follows from (57) and the I.H. For $l = 2k+1$, the results follows from (58) and the I.H. □

Lemma 3. $\Delta g_1(k) \geq 0$.
Proof: $\Delta g_1(1) = 0$. Assume result is true for $0 < k \leq 2^j$. Let $2^j < l \leq 2^{j+1}$. For $l = 2k$ the result follows from (54), Lemma 1 and the I.H. For $l = 2k+1$, the results follows from (56), Lemma 1 and the I.H. □

Lemma 4. $\Delta ag_1(k) > \Delta g_1(k)$.
Proof: $\Delta ag_1(1) = 1$ and $\Delta g_1(1) = 0$. Assume result is true for $0 < k \leq 2^j$. Let $2^j < l \leq 2^{j+1}$. For $l = 2k$ the result follows from (59), Lemma 2 and the I.H. For $l = 2k+1$, the results follows from (60), Lemma 2 and the I.H. □

Finally some notes. Equations (53)–(54) follow from equations (49)–(50) and (41)–(43). Equations (55)–(56) follow from equations (51)–(52) and (41)–(43). Equations (45)–(46) and (47)–(48) follow from (41)–(42) and (43)–(44).